Agradecimientos

Este trabajo no habría sido posible sin el apoyo y el estímulo de mi colega y amigo, Doctor Rudolf Fliesning, bajo cuya supervisión escogí este tema y comencé la tesis. Sr. Quentin Travers, mi consejero en las etapas finales del trabajo, también ha sido generosamente servicial, y me ha ayudado de numerosos modos, incluyendo el resumen del contenido de los documentos que no estaban disponibles para mi examen, y en particular por permitirme leer, en cuanto estuvieron disponibles, las copias de los recientes extractos de los diarios de campaña del Vigilante Rupert Giles y la actual Cazadora la señorita Buffy Summers, que se encontraron con William the Bloody en 1998, y por facilitarme el pleno acceso a los diarios de anteriores Vigilantes relevantes a la carrera de William the Bloody.

También me gustaría agradecerle al Consejo la concesión de Wyndham-Pryce como Compañero, el cual me ha apoyado durante mis dos años de investigación, y la concesión de dos subvenciones de viajes, una para estudiar documentos en los Archivos de Vigilantes sellados en Munich, y otra para la investigación en campaña en Praga. Me gustaría agradecer a Sr. Travers, otra vez, por facilitarme la acreditación de seguridad para el trabajo en los Archivos de Munich, y al Doctor Fliesning por su apoyo colegial y ayuda en ambos viajes de investigación.

No puedo terminar sin agradecer a mi familia, en cuyo estímulo constante y amor he confiado a lo largo de mis años en la Academia. Estoy agradecida también a los ejemplos de mis difuntos hermano, Desmond Chalmers, Vigilante en Entrenamiento, y padre, Albert Chalmers, Vigilante. Su coraje resuelto y convicción siempre me inspirarán, y espero seguir, a mi propio y pequeño modo, la noble misión por la que dieron sus vidas. Es a ellos a quien dedico este trabajo.

Índice general

Agradecimientos **1**

1. Introducción **4**
1.1. Objetivos 4
1.2. Estructura de la tesis 4
1.3. El método de dinámica molecular 5
1.3.1. Fundamentos 5
1.3.2. Esquema general del método 6
1.3.3. La función potencial 7
1.3.4. Condiciones de frontera 12
1.3.5. Estado del arte 13

2. La arquitectura GPU **16**
2.1. Introducción 16
2.2. Organización de procesadores 19
2.3. Organización de la memoria 22
2.4. Esquema de paralelismo 25
2.5. Diferencias entre Tesla, Fermi, Kepler 27
2.6. CUDA, Herramientas de desarrollo, profiling, exploración 28
2.7. Requerimientos de un problema para GPGPU 29
2.8. Diferencia entre CPU y GPU - Procesadores especulativos 31
2.9. Idoneidad para la tarea 33

3. Algoritmos **35**
3.1. Esquema general del algoritmo 36

3.2. Calculo usando tablas de valores del potencial 37
3.3. Calculo usando tablas de valores de la derivada 38

4. Implementaciones **40**
4.1. Consideraciones previas 40
4.2. Esquema general de la implementación sobre GPU 41
4.3. Implementación usando tabla de valores potenciales 42
4.4. Implementación usando tabla de derivadas 42
4.5. Implementacion sobre CPU 42
4.6. Implementación usando tabla de derivadas 43

5. Resultados **44**
5.1. Propiedades de los sistemas de evaluación 44
5.2. Performance 45
5.2.1. Evaluación de las implementaciones 45
5.2.2. Efectos del tamaño de bloque 45
5.3. Calidad numérica 45

6. Conclusiones **46**

Bibliografía **47**

Capítulo 1

Introducción

1.1. Objetivos

El principal objetivo de la tesis es obtener una implementación del método de dinámica molecular basada en GPU, que siga un modelo de interaccion de Lennard-Jones (potencial de no-unión), y que resuelva el cálculo de las fuerzas utilizando valores tabulados en el sistema de memoria provisto por la arquitectura.

La forma funcional que describe el potencial de Lennard-Jones permitiría obtener una buena aproximación del valor utilizando una tabla de resultados precalculados. Además, el sistema de memorias de la arquitectura, que ha evolucionado considerablemente desde su creación, permite recuperar de forma eficiente el valor asociado en un esquema de tablas. De esta forma, se espera que la modificación resulte en una mejora de la performance manteniendo la correctitud en los resultados de la simulación.

1.2. Estructura de la tesis

En las siguientes secciones de este capítulo se exponen los fundamentos teóricos necesarios para comprender el marco en el que se insertan los distintos cálculos que se buscan resolver. Se detalla el método que se implementa y se exhibe una representació abstracta y generalizada del algoritmo.

En el capítulo 2, se analiza en detalle la arquitectura sobre la cual se trabaja. Se

tienen en cuenta tanto las características físicas del hardware como las del modelo de programación asociado, poniendo énfasis en las características que serán utilizadas para este trabajo. Finalmente, se presentan de manera concisa los distintos aspectos de performance que se deben tener en cuenta a la hora de trabajar sobre este tipo de procesadores.

Más adelante, en el capítulo 3, se exponen los algoritmos que se desarrollaron a partir de los trabajos existentes en la literatura, y las modificaciones planteadas con respecto a estos. Se explican, además, las propiedades de estos algoritmo que permiten obtener una implementación eficiente haciendo uso de arquitecturas altamente paralelas.

Luego, en el capítulo 4, se explica el funcionamiento del código desarrollado para el procesador gráfico, justificando las decisiones que se tomaron en cada caso y detallando cómo se tuvieron en cuenta los aspectos de performance previamente mencionados.

Todos los resultados obtenidos, que incluyen la búsqueda de parámetros óptimos, análisis de performance de la implementación y de la calidad numérica de los cálculos realizados, se presentan en el capítulo 5.

Por último, en el capítulo número 6, se discuten todos los resultados obtenidos y el trabajo a futuro que deja esta tesis.

1.3. El método de dinámica molecular

1.3.1. Fundamentos

Los sistemas químicos de interés como las proteínas suelen ser complejos de estudiar debido a su gran tamaño. La superficie de energía libre, que describe la energía del sistema y a partir de la cual se podrían derivar todas las propiedades termodinámicas y cinéticas de interés, es una función 3N dimensional (siendo N el número de partículas del sistema), lo que hace imposible derivar analíticamente las propiedades por el altísimo costo computacional asociado.

Otra forma de analizar estos sistemas es mediante una aproximación numérica, a través de simulaciones computacionales. La simulación computacional de biomoléculas involucra la exploración de su superficie de energía libre, la cual, debido a la

complejidad de estos sistemas, es altamente accidentada, contiene una gran cantidad de mínimos locales y de barreras de energéticas. De esta forma, si los parámetros de la simulación están correctamente definidos, el conjunto de conformaciones adoptadas durante la ejecución será representativo del ensamble real de conformaciones posibles del sistema, lo que permite estimar las propiedades de interés.

Los métodos de simulación molecular nos permiten, entonces, obtener una serie de configuraciones representativas del sistema, de modo que las propiedades termodinámicas extraídas del mismo se correspondan de manera precisa con los valores reales. Una de las formas de obtener estas configuraciones es mediante el método de dinámica molecular(DM). Este método implica simular la progresión temporal -eal"del sistema, obteniendo distintas conformaciones a medida que avanza el tiempo de simulación.

La técnica de simulación de dinámica molecular se basa en resolver las ecuaciones de movimiento de Newton para cada átomo del sistema; así, en cada "paso"de la simulación se calcula la energía potencial entre las particulas y, de ésta, se derivan las fuerzas que actúan sobre cada átomo. Las ecuaciones de Newton relacionan las fuerzas resultantes del potencial con la aceleración que tendrá cada particula y, por lo tanto, el cambio en la velocidad y en la posicion con el tiempo. Dado que el potencial es una función continua dependiente de la posición, los cambios en las velocidades y posiciones resultan de una integración a lo largo de la trayectoria de la partícula.

1.3.2. Esquema general del método

En la práctica, la integración de las ecuaciones de Newton se calcula computacionalmente mediante el llamado Algoritmo de Verlet, el cual resuelve la integración con una ecuación discreta donde las posiciones estan separadas por intervalos de tiempo ("dt") . Usando las fuerzas resultantes, junto con las posiciones y velocidades de las partículas correspondientes a la iteración, el algoritmo de Verlet calcula las nuevas posiciones y velocidades en un intervalo de tiempo posterior. De este modo se genera una trayectoria, determinada por las posiciones de las particulas en cada paso de la simulación. Esta trayectoria describe como cambia la conformación espacial del sistema a lo largo del tiempo. La elección del "dt.[es] una situación de compromiso, ya que un valor muy chico, si bien representa la propagación del movimiento de forma

mas precisa(mas cercana al valor real de la integración), requiere una mayor cantidad de cálculos para un mismo tiempo total de simulación.

El esquema general del método se puede ver en la figura 1.1.

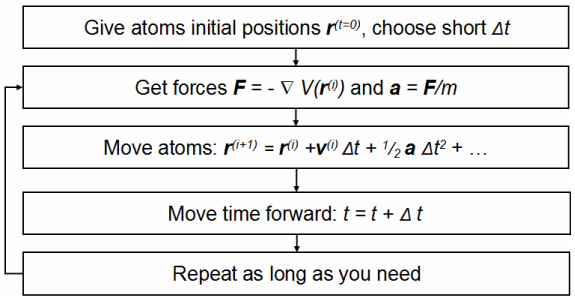

Figura 1.1: Esquema general del algoritmo de dinámica molecular

El resultado de la simulación es, fundamentalmente, una trayectoria representada por las posiciones de cada una de las partículas a lo largo de un cierto tiempo. No se va a detallar acerca del análisis posterior de este resultado pero es relevante decir que la longitud de trayectoria que podamos obtener en un tiempo de cómputo razonable es lo que limita el tipo de proceso químico que podremos estudiar. Para poder estudiar un proceso de interés, entonces, es necesario que la simulación alcance la escala de tiempo en la cual éste ocurre. Un mayor tiempo de simulación equivale a realizar más pasos y por lo tanto más cálculos totales, aumentando así el tiempo de cómputo requerido.

Por su parte, simular sistemas más grandes (con mayor cantidad de partículas) implica un incremento en la cantidad de cálculos por cada paso y por lo tanto un mayor costo computacional. Además, la función que describe las propiedades del sistema tiene por naturaleza una mayor cantidad de variables, debido a esto es necesario correr la simulación durante más tiempo para poder llegar a un conjunto de conformaciones que sea representativo del ensamble real.

1.3.3. La función potencial

En la utilización del método de dinámica molecular, el potencial(V) que permite derivar las fuerzas entre partículas se obtiene modelando al sistema molecula a través

de la mecánica clásica (método conocido como mecánica molecular o MM). Usando un método de MM, se ignoran los electrones y la naturaleza cuántica de estos. La energía potencial del sistema, entonces, depende exclusivamente de las posiciones de los núcleos atómicos. Se modela cada molécula como un conjunto de sitios -que representan los átomos que la componen- y resortes -que representan los enlaces químicos entre estos- junto con un potencial parametrizado ad hoc. Este potencial es una función matemática que depende exclusivamente de las posiciones de los átomos, la cual se ajusta a las interacciones observadas experimentalmente entre los componentes del sistema. Este tipo de repesentacion simplificada del sistema permite reducir la complejidad de los cálculos necesarios para la simulación y, por lo tanto, el costo computacional asociado. La desventaja de este tipo de modelos es que limita el conjunto de procesos que se pueden estudiar. Por ej. no es posible hacer análisis de reacciones quimicas que impliquen ruptura o formación de enlaces ya que estos no son considerados con suficiente detalle. Mas allá del método MM, existen otras formas de representar al sistema con mayor detalle(referencia QM) y aproximaciones que utilizan esquemas híbridos de representación(referencia QM/MM) que buscan balancear el costo computacional y el detalle que se obtiene del proceso estudiado.

Las interacciones representadas en el método de MM se pueden agrupar en dos tipos:

De unión: Describen las interacciones entre dos átomos unidos entre si directamente o hasta dos enlaces de distancia. Consisten en aquellos términos del potencial cuya energía se ve afectada por los estiramientos de los enlaces, las flexiones de los ángulos entre dos átomos, y la rotación de dos átomos adyacentes sobre un eje (ángulos dihedros). Los estiramientos y las flexiones angulares son modeladas mediante un oscilador armónico, mientras que las rotaciones de los enlaces en el plano son modeladas mediante una función trigonométrica.

De no unión: describen la interacción entre átomos ubicados a más de 3 enlaces de distancia de la misma molécula, o bien entre átomos de moléculas distintas, y consisten en un término para la contribución electrostática calculada mediante la Ley de Coulomb, y un término para la contribución de Van Der Waals modelada por un potencial de Lennard-Jones 12-6.

El esquema general de la función potencial suele tomar una forma similar a la que sigue:

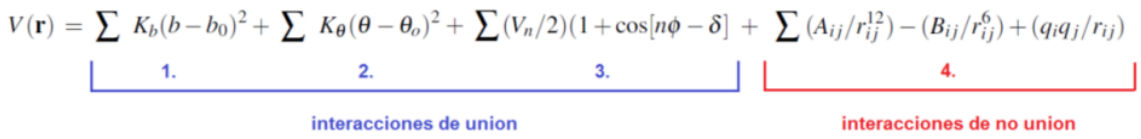

Figura 1.2: Ejemplo de potencial de interacción del paquete de software Amber

En azul se representan los términos matemáticos que modelan las interacciones de unión (1. Término de estiramiento de enlaces; 2. Término de flexión angular; 3. Término de rotación de ángulos diedros), y en rojo las interacciones de no unión (Potencial de Lennard-Jones 12-6 e interacción Coulómbica).

Los parámetros que se observan en esta función potencial (las constantes K_b, K_θ y V_n , los valores de equilibrio b_0, θ_0 y δ , las constantes A y B correspondientes al potencial Lennard-Jones, y las cargas q) deben ser ajustados especialmente para cada molécula individual que se desee incluir en la simulación. Estos parámetros se suelen obtener tanto de datos experimentales, como así también de rigurosos cálculos cuánticos, y en ocasiones emplean correcciones empíricas. Al conjunto o "set"de parámetros derivados para utilizarse de forma conjunta en una simulación se lo conoce como campo de fuerzas.

Esta parametrización del modelo de interacciones representa la principal diferencia entre los distintos paquetes de software para simulaciones de dinámica molecular. Las distintas versiones suelen tener uno o mas campos de fuerzas propios para utilizar en las simulaciones. Estos campos de fuerzas estan parametrizado específicamente para el tipo de sistema que se intenta simular, logrando que se ajuste lo mejor posible a la realidad. De esta forma, existen versiones específicas para simular distintos sistemas biologicos (proteínas, ácidos nucleicos), otros que intentan abarcar a sistemas de química orgánica en general, etc. La eleccion del campo de fuerzas es un aspecto importante de la simulacion de DM ya que éste define como se relacionan distintas partes de una misma molécula, como cada átomo se ve afectado por el entorno y como las condiciones contribuyen a la estructura molecular.

Potencial de Lennard-Jones

El cálculo de las interacciones de no unión es una paso importante en el algoritmo ya que implica la mayor parte del costo computacional asociado a la simulación. Esto se debe a que es necesario calcularlo entre todos los elementos del sistema. En particular, es importante el término correspondiente al potencial de Lennard-Jones porque este existe siempre, independientemente de la carga neta en las partículas que interaccionan.

La expresión mas común que representa este potencial es:

$$V_{LJ}(r) = 4\epsilon[(\sigma/r)^{12} - (\sigma/r)^{6}] \tag{1.1}$$

Esta ecuación está compuesta por dos términos representando fuerzas opuestas: El primer término ($(\sigma/r)^{12}$) representa fuerzas de repulsión que actúan a corta distancia. El segundo término ($(\sigma/r)^{6}$) representa fuerzas de atracción que actúan en un rango mayor de distancias.

Es usual reagrupar los términos σ y ϵ usando $A = 4\epsilon\sigma^{12}$ y $B = 4\epsilon\sigma^{6}$ para obtener una forma simplificada de la ecuación:

$$V_{LJ}(r) = [(A/r^{12}) - (B/r^{6})] \tag{1.2}$$

En la figura 1.3 se puede ver el comportamiento físico que describe este modelo de interacción:

Se pueden observar dos regiones separadas que representan las fuerzas opuestas de la ecuación. Estos segmentos están separados por un valor mínimo que se corresponde con el equilibrio entre el componente repulsivo y el de atracción. La distancia de equilibrio (que toma un valor de $2^{1/6} \approx 1{,}12$) se obtiene derivando el potencial e igualándolo a 0.

En el primer segmento de la curva ($r < 1{,}12\sigma$), el valor del potencial esta gobernado por el componente repulsivo. Este valor aumenta exponencialmente al disminuir la distancia, toma un valor $V = 0$ cuando $r = \sigma$, y se hace positivo para valores de $r < \sigma$ indicando una fuerza repulsiva entre los átomos.

En el punto de equilibrio ($r = 1{,}12\sigma$) el potencial toma el valor mínimo $V = -\epsilon$.

En el segundo segmento de la curva ($r > 1{,}12\sigma$), el valor del potencial esta

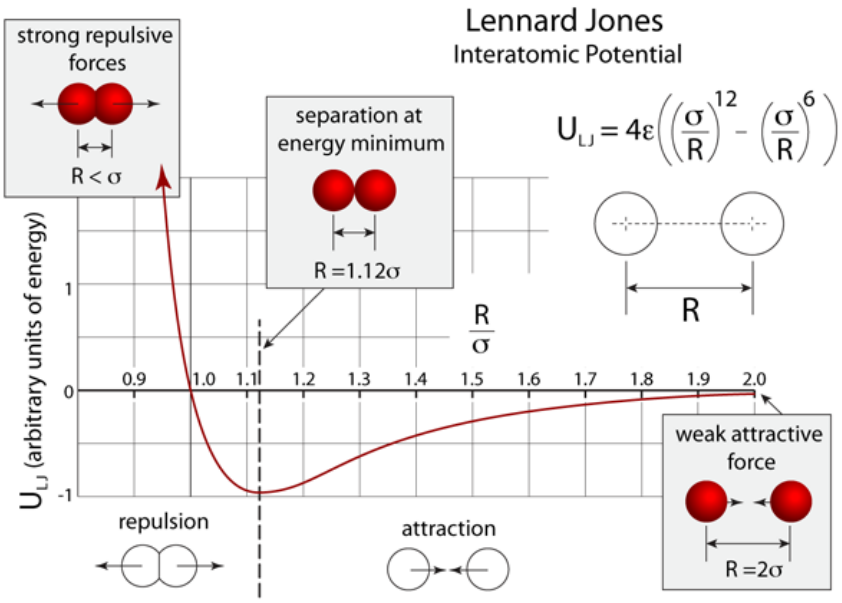

Figura 1.3: Comportamiento del potencial de Lennard-Jones

gobernado por el componente atractivo. El valor del potencial (siempre negativo, indicando una fuerza de atracción) se acerca a 0 a medida que la distancia se acerca a infinito.

A partir de la forma que toma el potencial de L-J resulta razonable pensar que, si se desprecia el de este potencial para interacciones entre partículas a una distancia mayor que cierto valor de corte, se podrá obtener una buena aproximación del comportamiento a la vez que se reduce considerablemente la cantidad de cálculos necesarios en cada paso de la simulación. Esta es una aproximación que proveen varias de las versiones implementadas del método, permitiendo que el usuario defina un valor de corte (cutoff). En cada paso, si dos partículas están separadas por una distancia mayor que el cutoff se les asigna automáticamente un valor de potencial $V_{LJ} = 0$ asumiendo que se está realizando una buena aproximación, lo que elimina el costo de tener que calcular el valor real. Depende del usuario definir un cutoff que sea coherente con el sistema y simulación a realizar.

Como se mencionó previamente, el potencial de L-J ofrece una aproximación generalmente buena de las interacciones de no unión en modelos moleculares y, dada su simplicidad, es incluido en gran parte de los campos de fuerzas utilizados en

dinámica molecular. Es por esto que la optimización del calculo asociado tendría un gran impacto en este métdodo de simulación. Además, la forma funcional implica que existe, al menos débilmente, entre todos los pares de particulas del sistema, lo que se traduce en una gran cantidad de cálculos por paso. El cálculo de este potencial representa así la mayor parte del costo computacional asociado a la simulación.

1.3.4. Condiciones de frontera

Hasta aquí hemos definido al sistema como el conjunto de átomos sobre los cuales realizamos la simulación. Sin embargo, los sistemas reales tienen, en términos practicos, un tamaño infinito. Es decir, independientemente de cuántos elementos podamos incluir en nuestro conjunto de simulación, el número de parículas(N) será siempre despreciable con respecto al nuúero de componentes del sistema macroscópico real.

Si no se implementa ninguna condición particular a los límites de nuestro sistema, se estaría simulando un conjunto de particulas que se encuentra en el vacío. De esta forma, las interacciones ocurrirían solo entre las particulas del conjunto de simulación que definimos, sin tener en cuenta ningún elemento en el contexto. Además, no habría limites definidos para las posibles posiciones de las partículas, las cuales pueden, por lo tanto, difundir libremente durante la ejecución.

Este tipo de simulaciones raramente es útil ya que no representan situaciones realistas. Se debe tener en cuenta, entonces, condiciones especiales que tendrán los limites en la simulación para que estos representen de forma adecuada a los sistemas de interés. Existen distintas aproximaciones para implementar estas condiciones en los límites manteniendo un conjunto reducido de particulas.

Utilizar condiciones periódicas de borde es la forma mas común de simular un sistema infinito. Aplicando esta condición, las partículas están contenidas en una caja de simulación con un tamaño definido. Esta caja es replicada virtualmente al infinito en las tres direcciones del eje cartesiano, llenando completamente el espacio. Todas las imágenes de las partículas se mueven de forma idéntica a la partícula central. De hecho, sólo una de ellas(la que se encuentra en la caja central) es la que efectivamente se está simulando. Cuando una partícula entra o sale de la caja central, una imagen de ella entra o sale por la cara opuesta de esta región, de manera tal que el número de partículas contenidas en la región de simulación siempre se conserva.

Se puede ver que, de esta forma, los límites del sistema que se está simulando son virtualmente eliminados y no tienen ningún efecto sobre la simulación. Como resultado de esto, cada partícula perteneciente al conjunto de simulación interactúa no solo con las demás partículas de este, sino también con las imágenes virtuales que estamos teniendo en cuenta.

Al parecer, el número de interacciones (y por lo tanto la cantidad de cálculos a realizar) se incrementará enormemente con esta nueva condición. Sin embargo, esto puede evitarse si se utiliza un potencial con un rango finito, dado que en este caso se pueden ignorar las interacciones entre partículas separadas por una distancia mayor a cierto valor de cutoff. En este caso, teniendo un valor de cutoff R_c y definiendo una caja de simulaciún cuyo lado menor tenga una longitud L_m: Si hacemos cumplir que $L_m > 2R_c$, entonces se verá que, como máximo, solo una de las imágenes de cada particula quedará dentro del rango del cutoff y por lo tanto debera ser considerada. Este método, conocido como el criterio de imagen mínima, permite limitar el número de cálculos necesarios considerando solo la imagen mas cercana e ignorando el resto. De esta forma se puede cumplir con las características infinitas del sistema real y las propiedades impuestas por el potencial, logrando una mejor aproximación en la simulación. La figura 1.4 muestra una representación gráfica de este método.

1.3.5. Estado del arte

Desde que fue desarrollado, el método ha sido de una gran utilidad y ha demostrado una posiblidad de aplicación a futuro mucho mayor. Esta posiblidad de aplicarlo a sistemas cada vez mas grandes y procesos en escalas de tiempo mayores es lo que impulsó una constante búsqueda de mejoras sobre el algoritmo inicial. Se han desarrollado entonces una gran cantidad de estrategias que apuntan a optimizar el costo computacional asociado al método, sin sufrir una pérdida considerable en la precisión.

Para sistemas típicos de biomoleculas en donde el sistema está compuesto por una macromolécula(ej. proteínas) rodeada de un solvente líquido, es posible utilizar una aproximación conocida como solvente implicito. En este modelo, el solvente se representa como un medio continuo alrededor del sistema central en lugar de representar explícitamente todas las unidades que lo componen, reduciendo considerablemente

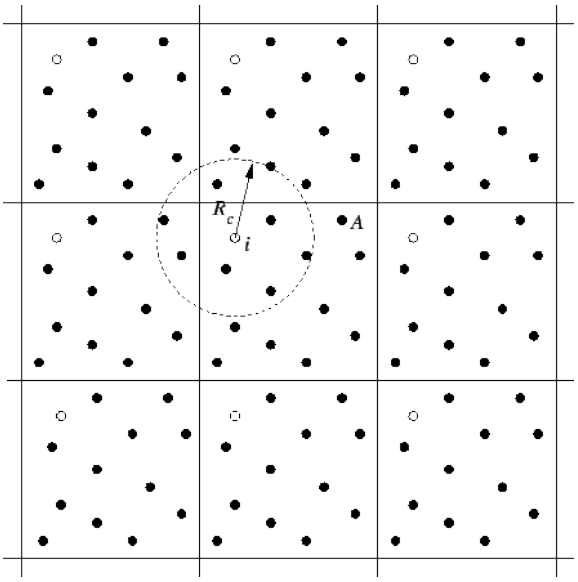

Figura 1.4: Criterio de imagen mínima para condiciones periódicas de frontera. Los círculos sin relleno representan todas las imágenes de la misma particula i

el número de cálculos necesarios para evaluar la interacción con las moléculas de solvente. Al ser una aproximación, tiene ciertas limitaciones en cuanto a la precisión y al tipo de sistemas en los que puede ser aplicado.

El calculo asociado al componente electroestático puede ser optimizado usando el método conocido como sumatoria Ewald . Este método permite aproximar el cálculo de interacciones que actúan en un amplio rango de distancias sobre sistemas periódicos.

Otra adaptación muy utilizada es la implementación de una lista de vecinos para cada partícula . Esta lista mantiene las moleculas que estan dentro del rango de interaccion(distancia menor al cutoff) y son los unicos que se tendran en cuenta a la hora de hacer el calculo. La optimización se basa en que la lista solo se actualiza cuando se ejecuta una cierta cantidad de pasos definida como parámetro. Este valor se deriva experimentalmente a partir de simulaciones de prueba y es el parámetro que balancea la optimización y la precisión: cuanto mas se demora la actualizacion mejor es el tiempo de ejecución pero menos representativa es la lista acerca de la realidad del entorno y, por lo tanto, menos preciso es el cálculo.

Existe una gran cantidad de variantes del algoritmo, solo se han mencionado

algunos de los caminos en los que se ha avanzado realizando modificaciones con respecto a la descripción inicial.

En cuanto a las arquitecturas de cómputo, las implementaciones sobre GPU, que son el centro de este trabajo, se han establecido como estándares para las implementaciones actuales del método. Esto se debe al gran poder de cómputo que proveen en relación al costo económico, teniendo en cuenta que el algoritmo es altamente paralelizable y este tipo de hardware provee las condiciones necesarias para una implementación eficiente.

Con este contexto , las capacidades de cálculo actuales permiten realizar simulaciones de dinámica molecular en sistemas de hasta XXX átomos aproximadamente, sobre un intervalo de tiempo del orden de YYYYYYY micro/pico/******segundos).

Si bien se ha avanzado mucho desde el desarrollo del método, existen múltiples procesos de interés que podrían ser abordados mediante experimentos de simulación computacional pero que el intervalo de tiempo en el cual ocurren no es posible de ser alcanzado con los recursos disponibles hoy en dia. Esta situación hace que se destine una gran cantidad de esfuerzo para continuar las mejoras en el método y mantener las implementaciones a la par de las arquitecturas disponibles, las cuales han avanzado considerablemente teniendo en cuenta que hace un largo tiempo que el método comenzó a ser aplicado al estudio de proteinas (Referencia Dynamics of folded proteins McCammon, J. Andrew; Gelin, Bruce R.; Karplus, Martin Nature (London, United Kingdom) (1977), 267 (5612), 585-90CODEN: NATUAS; ISSN:0028-0836. The dynamics of a folded globular protein (bovine pancreatic trypsin inhibitor). El presente trabajo intenta, entonces, evaluar posibles modificaciones que permitan adecuar las implementaciones más actuales del método para explotar al máximo todas las características que ofrece la arquitectura de GPU.

Capítulo 2

La arquitectura GPU

2.1. Introducción

Este trabajo se centra en la arquitectura GPU desarrollada por Nvidia, conocida como CUDA por las siglas en ingles de *Compute Unified Device Architecture*. CUDA surge naturalmente de la aplicación del hardware desarrollado para problemas gráficos, pero aplicados al cómputo científico.

Las placas de vídeo aparecen en 1978 con la introducción de Intel del chip `iSBX 275`. En 1985, la Commodore Amiga incluía un coprocesador gráfico que podía ejecutar instrucciones independientemente del CPU, un paso importante en la separación y especialización de las tareas. En la década del 90, múltiples avances surgieron en la aceleración 2D para dibujar las interfaces gráficas de los sistemas operativos y, para mediados de la década, muchos fabricantes estaban incursionando en las aceleradoras 3D como agregados a las placas gráficas tradicionales 2D. A principios de la década del 2000, se agregaron los *shaders* a las placas, pequeños programas independientes que corrían nativo en el GPU, y se podían encadenar entre sí, uno por pixel en la pantalla [10]. Este paralelismo es el desarrollo fundamental que llevó a las GPU a poder procesar operaciones gráficas órdenes de magnitud más rápido que el CPU.

En el 2006, Nvidia introduce la arquitectura G80, que es el primer GPU que deja de resolver únicamente problemas de gráficos para pasar a un motor genérico donde cuenta con un set de instrucciones consistente para todos los tipos de operaciones que realiza (geometría, vertex y pixel shaders) [23]. Como subproducto de esto, la

GPU pasa a tener procesadores simétricos más sencillos y fáciles de construir. Esta arquitectura es la que se ha mantenido y mejorado en el tiempo, permitiendo a las GPU escalar masivamente en procesadores simples, de baja frecuencia de reloj y con una disipación térmica manejable.

Los puntos fuertes de las GPU modernas consisten en poder atacar los problemas de paralelismo de manera pseudo-explícita, y con esto poder escalar "fácilmente" si solamente se corre en una placa con más procesadores [16].

Técnicamente, esta arquitectura cuenta con entre cientos y miles de procesadores especializados en cálculo de punto flotante, procesando cada uno un *thread* distinto pero trabajando de manera sincrónica agrupados en bloques. Cada procesador, a su vez, cuenta con entre 63 a 255 registros [14, 17]. Las GPU cuentas con múltiples niveles de cache y memorias especializadas (subproducto de su diseño fundamental para gráficos). Estos no poseen instrucciones SIMD, ya que su diseño primario esta basado en cambio, en SIMT (*Single Instruction Multiple Thread*), las cuales se ejecutan en los bloques sincrónicos de procesadores. De este modo, las placas modernas como la Nvidia Tesla K40 alcanzan poder de cómputo de 4,3 TFLOPs (4300 mil millones de operaciones de punto flotante por segundo) en cálculos de precisión simple, 1,7 TFLOPs en precisión doble y 288 GB/seg de transferencia de memoria, usando 2880 CUDA Cores [15]. Para poner en escala la concentración de poder de cálculo: una computadora usando solo dos de estas placas posee una capacidad de cómputo comparable a la supercomputadora más potente del mundo en Noviembre 2001 [22]. Una comparativa del poder de cómputo teórico entre GPUs y CPUs puede ver en la figura 2.1.

Para poder explotar la arquitectura CUDA, los programas deben ser diseñados de manera de que el problema se pueda particionar usando el modelo de grilla de bloques de *threads*. Para este propósito es que Nvidia desarrolló el lenguaje CUDA.

Hoy en día, poder aprovechar la potencialidad de las GPU requiere una reescritura completa de los códigos ya existentes desarrollados para CPU y un cambio de paradigma importante, al dejar de tener vectorización, paralelización automática y otras técnicas tradicionales de optimización en CPU. Sin embargo, este trabajo ha rendido sus frutos en muchos casos: en los últimos seis años, la literatura de HPC con aplicaciones en GPU ha explotado con desarrollos nuevos basados en la aceleración

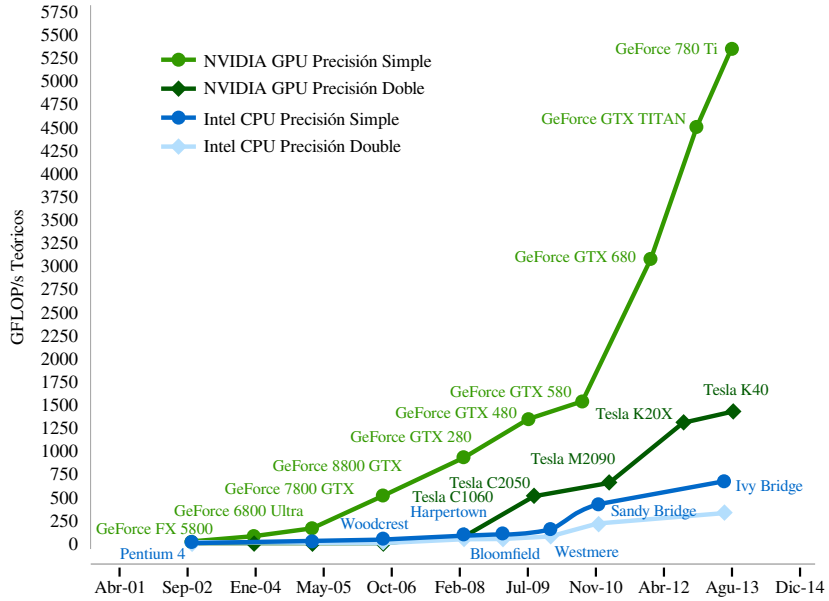

Figura 2.1: Picos teóricos de *performance* en GFLOPS/s. Tomado de [16].

de algoritmos numéricos (su principal uso). Por este motivo, este trabajo no ahondará en las particularidades del lenguaje CUDA y su modelo de paralelismo, más allá de lo estrictamente necesario para analizar *performance*. Para más información se puede consultar la bibliografia [4, 21, 23].

Además, no todas las aplicaciones deben reescribirse de manera completa. Con la introducción de las bibliotecas `CuBLAS` y `CuFFT`, se ha buscado reemplazar con mínimos cambios las históricas bibliotecas `BLAS` y `FFTw`, piedras fundamentales del cómputo HPC [12, 13].

Nuevas soluciones para la portabilidad se siguen desarrollando: las bibliotecas como `Thrust` [3], `OpenMP` 4.0 [2] y `OpenACC` 2.0 [1] son herramientas que buscan generar código que puedan utilizar eficientemente el acelerador de cómputo que se haya disponible. Estas herramientas permiten definir las operaciones de manera genérica y dejan el trabajo pesado al compilador para que subdivida el problema de manera que el acelerador (CPU, GPU, MIC) necesite. Obviamente, los ajustes finos siempre quedan pendiente para el programador especializado, pero estas herramientas representan un avance fundamental al uso masivo de técnicas de paralelización automáticas, necesarias hoy día y potencialmente imprescindibles en el futuro.

2.2. Organización de procesadores

Los procesadores GPGPU diseñados por Nvidia han sido reorganizados a lo largo de su existencia múltiples veces pero conservan algunas líneas de diseño a través de su evolución. A continuación se describe la organización definida en la arquitectura `Fermi` y luego analizaremos las diferencias con `Kepler`.

Las arquitecturas de las GPUs se centran en el uso de una cantidad escalable de procesadores *multithreaded* denominados *Streaming Multiprocessors* (SMs). Un multiprocesador esta diseñado para ejecutar cientos de threads concurrentemente, usando sus unidades aritméticas llamadas *Streaming Processors* (SPs). Las instrucciones se encadenan para aprovechar el paralelismo a nivel instrucción dentro de un mismo flujo de ejecución, y funcionando en conjunto con el paralelismo a nivel de *thread*, usado de manera extensa a través del hardware. Todas las instrucciones son ejecutadas en orden y no hay predicción de saltos ni ejecución especulativa, todo se ejecuta solamente cuando se lo necesita [21].

Los SMs (figura 2.2) son unidades completas de ejecución. Cada uno de ellos tiene 32 SPs interconectados entre sí que operan sobre un *register file* de 64 KB común a todos. Los SMs cuentan con múltiples unidades de *Load/Store*, que permiten realizar accesos a memoria independientes. Existen cuatro unidades de SFU (*Special Function Unit*) por SM, para realizar rápidamente operaciones matemáticas trascendentales (trigonométricas, potencias, raíces, etc.). Cada SM ejecuta simultáneamente una cantidad fija de threads, llamado *warp*, con cada uno de estos corriendo en un SP. Las unidades de despacho de warps se encargan de mantener registro de qué *threads* están disponibles para correr en un momento dado y permiten realizar cambios de contexto por hardware eficientemente ($< 25\mu$s) [18]. Con esto, se pueden ejecutar concurrentemente dos warps distintos para esconder la latencia de las operaciones. En precisión doble, esto no es posible, así que hay solamente un warp corriendo a la vez.

Un SM cuenta con una memoria común de 64 KB que se puede usar de forma automática tanto como memoria compartida común a todos los *threads* como cache L1 para todos los accesos a memoria.

Por como funciona un pipeline gráfico clásico, los SM se agrupan de a cuatro en GPCs (*Graphics Processing Cluster*) y no interactúa con el modelo de cómputo de

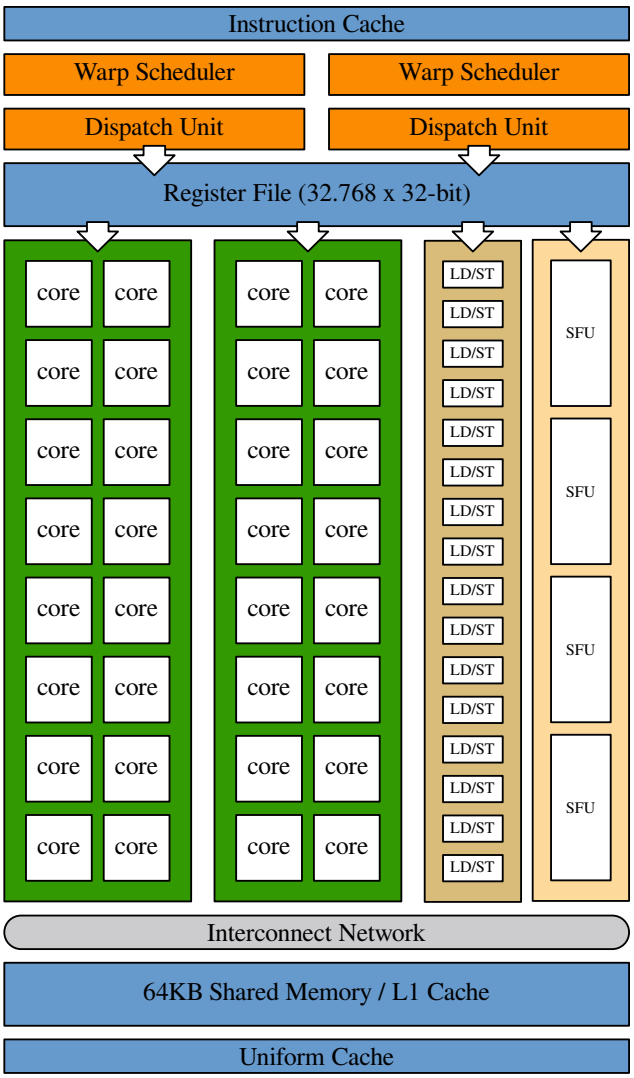

Figura 2.2: Diagrama de bloques del SM de GF100 Fermi. Basado en [17].

Figura 2.3: Diagrama de bloques de GF100 Fermi. Tomado de [17].

CUDA. Un esquema de esta división global de los SM y cómo se comunican puede verse en la figura 2.3.

Todos los accesos a memoria global (la memoria por fuera del procesador) se realizan a través de la cache L1 de cada SM y a través de la L2 del todo el procesador. Esta L2 consiste de seis bancos compartidos de 128 KB. Estas caches se comunican de manera directa tanto con la DRAM propia de la placa como con el bus PCI Express por el cual pueden comunicarse dos placas entre sí, sin pasar por CPU, y son *write-through*, es decir cada escritura se hace tanto en la DRAM como en la memoria cache.

Como estos procesadores implementan el estándar IEEE754-2008, cuentan con operaciones de precisión simple y doble acorde al mismo, por lo cual los cálculos intermedios en operaciones como FMA (*Fused Multiply-Add*), que toma tres operandos y devuelve el producto de dos de ellos sumado al tercero, no pierden precisión por redondeo.

2.3. Organización de la memoria

La memoria de la GPU es uno de los puntos cruciales de esta arquitectura, un esquema gráfico puede observarse en la figura 2.4. Esta se subdivide entre memorias on-chip y memorias on-board, de acuerdo a su ubicación y latencia de acceso, en cuatro categorías distintas:

- Registros
- Memoria local
- Memoria compartida
- Memoria global

Cada *thread* de ejecución cuenta con una cantidad limitada de registros de punto flotante de 32 bits con latencia de un par de ciclos de clock. A su vez, existe una cantidad finita de registros totales que cuenta un SM (oscila entre 16535 y 65535 registros). Por su baja latencia son la clase principal de almacenamiento temporal.

La memoria local es una memoria propia de cada *thread*, y se encuentra almacenada dentro de la memoria global. Esta memoria es definida automáticamente por el compilador y sirve como área de almacenamiento cuando se acaban los registros: los valores anteriores se escriben a esta memoria, dejando los registros libres para nuevos valores en cálculos, y cuando se terminan estos cálculos se carga los valores originales nuevamente. Cuenta con las mismas desventajas que la memoria global, incluyendo su tiempo de acceso.

La memoria compartida, o *shared*, es una memoria que es visible para todos los *threads* dentro de un mismo SM. Cada *thread* puede escribir en cualquier parte de la memoria compartida dentro de su bloque y puede ser leído por cualquier otro *thread* de este. Es una memoria muy rápida, on-chip, y que tarda aproximadamente 40 ciclos de acceso [24]. Esta memoria es compartida con la cache L1, la cual tiene capacidad de entre 16 KB y 64 KB configurable por software. Esta memoria se encuentra dividida en 32 bancos de 2 KB de tamaño, permitiendo que cada uno de los 32 *threads* acceda independientemente a un float. Si hubiera conflicto, los accesos a ese banco se serializarían, aumentando la latencia de la llamada [4].

La memoria global es la memoria principal fuera del chip de la GPU. Esta es de gran tamaño (de entre 1 GB y 12 GB) y es compartida por todos los SM de la GPU y los CPU que integran el sistema. Es decir, tanto los GPU como los CPU pueden invocar las funciones de CUDA para transferir datos entre la memoria de la placa y la memoria RAM de *host*. La latencia de acceso a la memoria global es de cientos de ciclos [24], sumamente lenta en comparación con el procesador. La memoria global también puede ser mapeada, o *pinneada*, para que exista una copia de esa reserva tanto en la memoria en la placa como en la memoria principal del procesador. El driver de CUDA va a mantener la consistencia entre ambas de manera asíncrona, evitando la necesidad de hacer copias de memoria explícitas. No es ilimitada la cantidad de memoria mapeada posible, por lo que es importante saber elegir qué elementos se van a almacenar de esta manera.

Adicionalmente, la GPU cuenta con múltiples niveles de memorias cache para poder aminorar el hecho de que el principal cuello de botella del cómputo es la latencia en los accesos a memoria global. Estas se dividen en cuatro:

- Cache L1

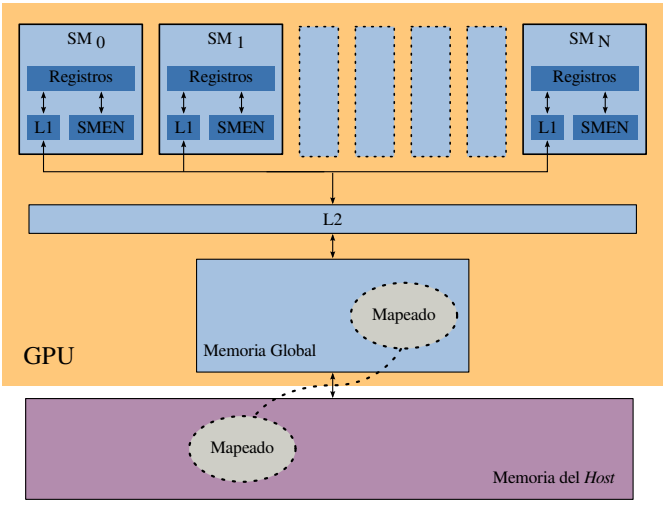

Figura 2.4: Esquema de la jerarquía de memorias en GPU, detallando de las disponibles en cada SM, la memoria compartida (SMEN), la L2 global, la memoria de la placa (global) y la mapeada entre el *host* y la placa de video. Tomado de [4].

- Cache L2
- Cache constante
- Cache de textura

La cache L1 es dedicada por SM. Esta cache fue introducida en Fermi y su diseño hace que también está dedicada a la memoria compartida, por lo que es posible en tiempo de ejecución darle directivas a la GPU que asigne más memoria cache o más memoria compartida, permitiendo a los bloques tener mayores espacios de memorias compartidas o mayores *hit rates* de caches.

La cache L2 es común a todos los SM de la GPU, donde, a partir de Fermi en Nvidia, todos los accesos de lectura y escritura a memoria global y textura pasan a través de esta [17].

La cache constante es una cache sobre la memoria global dedicada solamente a lecturas de memoria. Esta es muy reducida (solo cuenta con 64 KB) y está optimizada para muchos accesos a la misma dirección. Cuando un *thread* lee esta memoria, se retransmite a los demás *threads* del warp que estén leyendo esa misma dirección, reduciendo el ancho de banda necesario. Si, en cambio, los *threads* leen distintas

direcciones, los accesos se serializan. Cuando hay un *miss* de esta memoria, la lectura tiene el costo de una lectura de memoria global.

La cache de textura es una cache sobre la memoria global que presenta no solo localidad espacial, como la mayoría de las caches de procesadores normales (es decir, la cache contiene una porción consecutiva de la memoria principal), sino que se le puede agregar el concepto de dimensiones, para poder modelar datos en más de una dimensión. Esto se adapta de muy bien a los problemas de gráficos en 2D y 3D, y es una herramienta clave a la hora de minimizar los accesos a matrices no solo por filas sino por columnas. Esta cache se debe definir en momento de compilación en el código, ya que tiene límites espaciales (necesarios para poder definir áreas de memoria sobre la cual operar) y a su vez se debe acceder a los datos subyacentes a través de funciones específicas. Una característica adicional de esta cache es que como necesita resolver estos accesos no convencionales a la memoria, cuenta con una unidad propia de resolución de direcciones. Esta unidad tiene limitantes en cuanto a sus posibilidades, ya que no posee un ancho de banda suficiente como para resolver todos los accesos a memoria globales que podrían surgir, por lo cual su uso debe ser criterioso.

2.4. Esquema de paralelismo

Al ser una arquitectura masivamente paralela desde su concepción, CUDA presenta varios niveles de paralelismo, para agrupar lógicamente el cómputo y poder dividir físicamente su distribución. Los principales son:

- Bloques de *threads*
- Grilla de bloques
- Streams
- Múltiples placas

El paralelismo a nivel de bloque instancia una cantidad de *threads*, subdivididos lógicamente en 1D, 2D o 3D. Los *threads* internamente se agrupan de a 32, es decir, un *warp*. Cada uno de estos *threads* va a contar con una manera de identificarlos

unívocamente: un `blockId` y, dentro de cada bloque, su propio `threadId`. Además, van a correr simultáneamente en el mismo SM y van a ser puestos y sacados de ejecución de a un warp dinámicamente por el *scheduler* de hardware que cuenta cada SM. Para compartir información entre ellos, se puede utilizar la memoria compartida o las instrucciones de comunicación de *threads* intrawarp (solo disponibles a partir de Kepler [14]).

El paralelismo a nivel de grilla determina una matriz de bloques de ejecución que particiona el dominio del problema. El *GigaThread Scheduler* va a ejecutar cada bloque en un SM hasta el final de la ejecución de todos los *threads* de este. Los bloques no comparten información entre sí. Por esto, no pueden ser sincronizados mediante memoria global ya que no se asegura el orden en el que serán puestos a correr, y un bloque mantiene su SM ocupado hasta que termine de ejecutar, bloqueando a los demás (es decir, no hay *preemption* en los SM).

El paralelismo de stream es una herramienta empleada para hacer trabajos concurrentes usando una sola placa. Esta técnica permite que múltiples kernels (unidades de código en CUDA) o copias de memoria independientes estén encolados, para que el driver pueda ejecutarlas simultáneamente si se están subutilizando los recursos, de forma de minimizar tiempo ocioso del dispositivo. Los streams permiten kernels concurrentes pero cuentan con importantes restricciones que generan sincronización implícita, lo cual hay que tener presente si se desea mantener el trabajo de forma paralela.

El paralelismo a nivel de placa consiste en poder distribuir la carga del problema entre distintas GPUs dispuestas en un mismo sistema compartiendo una memoria RAM común como si fuera un software multithreaded tradicional. CUDA no cuenta con un modelo implícito de paralelismo entre distintas placas, pero es posible hacerlo manualmente eligiendo de manera explícita qué dispositivo usar. Las placas se pueden comunicar asíncronamente entre sí, tanto accediendo a las memorias globales de cada una como ejecutando código remotamente. En las versiones modernas del driver de CUDA, también pueden comunicarse directamente las placas entre sí a través de la red, permitiendo escalar multinodo fácilmente en un cluster de cómputo [4].

2.5. Diferencias entre Tesla, Fermi, Kepler

Hasta ahora se describió la arquitectura vista desde el punto de vista Fermi, que es la segunda arquitectura GPGPU diseñada por Nvidia. Fermi es la evolución de Tesla, construida para desacoplar aún más los conceptos de procesamiento gráfico de modo de lograr un procesador más escalable y de propósito general. La arquitectura sucesora a Fermi es Kepler, presentada en el 2012, con las metas de disminuir el consumo y aumentar la potencia de cálculo [14].

Características	Tesla (GT200)	Fermi (GF100)	Kepler (GK110)
Año introducción	2006	2010	2012
Transistores	1400 millones	3000 millones	3500 millones
Tecnología fabricación	65 nm	40 nm	28 nm
SMs	30	16	15
SP / SM	8	32	192
Caché L1	-	16 - 48 KB	16 - 32 - 48 KB
Caché L2	-	768 KB	1536 KB
Memoria Shared/SM	16 KB	16 - 48 KB	16 - 32 - 48 KB
Registros/Thread	63	63	255
Pico Precisión Simple	240 MAD / clock	512 FMA / clock	2880 FMA / clock
Pico GFLOPS Simple	933	1345	3977
GFLOPS/Watt	3,95	5,38	15,9

Cuadro 2.1: Tabla comparativa de las características más prominentes de las tres arquitecturas de CUDA.

En la tabla 2.1 se ve una comparación de las recursos que están más directamente relacionados a la *performance* de un dispositivo GPU. Se puede apreciar el crecimiento notable del poder de cómputo debido a las tecnologías de fabricación, que permitieron aumentar la cantidad de transistores por unidad de superficie. También se puede comprobar que, a diferencia de los CPU, las arquitecturas GPGPU decidieron utilizar esos nuevos transistores disponibles para más núcleos de procesamiento, en vez de dedicarlas a aumentar las memorias cache, que crecieron mínimamente (comparando contra las caches de CPU).

Una de las diferencias más notorias entre Tesla y Fermi es la presencia de FMA contra el MAD (*Multiply - Add*). El MAD realiza la multiplicación y la acumulación

en dos pasos, pero más rápidos que hacerlos independientemente por tener hardware dedicado. Debido a que debe redondear entre los pasos, pierde precisión y no respeta completamente el estándar IEEE754-2008. El FMA, en cambio, lo hace en una sola operación, y sin redondeos intermedios.

La métrica usada por Nvidia para publicitar la *performance* de estos dispositivos y poder compararlos entre sí, y contra CPU, son los GFLOPS. Esta unidad mide cuantas operaciones de punto flotante de precisión simple se pueden realizar por segundo. Los GFLOPs son utilizados también por los clusters en el ranking TOP500, donde se ordenan de acuerdo a la *performance* medida usando un software estandarizado, `LINPACK`. No solo es notable como se cuadruplicó la *performance* (teórica) en solamente seis años, sino que aún más importante es como mejoró la *performance* por Watt. Esto también se ve en que Kepler tiene menos SM que Fermi o Tesla, pero son mucho más poderosos y eficientes. La tecnología de fabricación ha ayudado a la disminución del consumo, un problema que acechaba a los diseños Fermi, ya que sus consumos superiores a 200W por dispositivo los hacían muy difíciles de refrigerar incluso en clusters de HPC. Se puede apreciar entonces la estrategia de mercado de Nvidia de introducirse en las supercomputadoras de todo el mundo, donde el consumo y la refrigeración son factores limitantes (mucho más aún que, por ejemplo, en computadoras de escritorio) [7].

2.6. CUDA, Herramientas de desarrollo, profiling, exploración

Para soportar una arquitectura masivamente paralela, se debe usar una ISA (*Instruction Set Architecture*) diseñada especialmente para el problema. En el caso de CUDA, esta ISA se denominada PTX y debe poder soportar conceptos fundamentales del cómputo GPGPU: grandes cantidades de registros, operaciones en punto flotante de precisión simple y doble, y FMA (fused multiply-add). Además, el código compilado para GPU debe ser agnóstico al dispositivo que lo va a correr, por lo cual la paralelización no debe estar demasiado atada a este, sino que el dispatching lo debe poder determinar el driver de la placa en tiempo de ejecución. Un último requerimiento clave de esta ISA es que debe soportar hacer ajustes manuales, para

poder construir partes claves de ciertas bibliotecas frecuentemente usadas (como las rutinas de BLAS de álgebra lineal) [17].

El lenguaje CUDA es una extensión de C++, con ciertas características agregadas para poder expresar la subdivisión de las rutinas en *threads* y bloques, junto con mecanismos para especificar qué variables y funciones van a ejecutarse en la GPU y en el CPU. Una característica de CUDA es que todas las llamadas a los kernels de ejecución son asincrónicas, por lo que es relativamente sencillo solapar código en GPU y CPU. A su vez se cuenta con múltiples funciones opcionales, con distinta granularidad, que permiten esperar a que todas las llamadas asíncronas a GPU finalicen, agregando determinismo en forma de barreras de sincronización al lenguaje.

El código CUDA compila usando `nvcc`, una variante del `GNU gcc` que se encarga de generar el código PTX para las funciones que se van a ejecutar en las GPU. Este código objeto después se adosa normalmente con el resto del código que corre en CPU y se genera un binario ejecutable.

Nvidia, además, provee herramientas de profiling para explorar cómo se están utilizando los recursos durante la ejecución. Éstas son esenciales para optimizar, puesto que los limitantes de GPU son sumamente distintos a los de CPU, presentando dificultades conceptuales incluso para programadores experimentados. Las herramientas de profiling no solo muestran *runtime*, sino que sirven para ver dónde hay accesos a memoria excesivos, puntos de sincronización costosos, limitantes en los registros y cómo se superponen las llamadas asincrónicas.

El uso de todas estas herramientas fue vital en este trabajo para poder entender cómo funciona la arquitectura en detalle, cómo medir *performance* y utilización, y cómo los cambios realizados impactaron en las distintas generaciones de dispositivos.

2.7. Requerimientos de un problema para GPGPU

Dada la organización de un procesador GPU, un problema debe exhibir al menos las siguientes características para que tenga potencialidad para poder aprovechar las características y recursos disponibles en esta arquitectura:

1. El problema debe tener una gran parte paralelizable.
2. El problema debe consistir, mayormente, de operaciones numéricas.
3. El problema debe poder ser modelado, en su mayor parte, utilizando arreglos o matrices.
4. El tiempo de cómputo debe ser muy superior al tiempo de transferencia de datos.

El ítem 1 se refiere a que debe existir alguna forma de partir el problema en subproblemas que puedan realizarse simultáneamente, sin que haya dependencias de resultados entre sí. Si el problema requiere partes seriales, lo ideal es que se las pueda dividir en partes independientes que sean etapas de una cadena de procesos, donde cada una de éstas exhiban características fuertemente paralelas. Como las arquitecturas masivamente paralelas tienen como desventaja una menor eficiencia por núcleo, si el problema no se puede dividir para maximizar la ocupación de todos los procesadores disponibles, va a resultar muy difícil superar en eficiencia a los procesadores seriales.

El ítem 2 habla acerca de que el método de resolución de los problemas debe provenir de una aplicación numérica o de gran carga aritmética. El set de instrucciones de las arquitecturas GPGPU están fuertemente influenciados por las aplicaciones 3D que las impulsaron en un principio. Éstas consisten mayormente de transformaciones de álgebra lineal para modelar iluminación, hacer renders o mover puntos de vistas. Todos estos problemas son inherentemente de punto flotante, por lo cual el set de instrucciones, las ALUs internas y los registros están optimizados para este caso de uso.

El ítem 3 menciona que los problemas que mejor se pueden tratar en esta arquitectura se pueden representar como operaciones entre arreglos o matrices de dos, tres o cuatro dimensiones. Las estructuras de datos no secuenciales en memoria incurren en múltiples accesos a memoria para recorrerlas y, en las arquitecturas GPGPU, generan un gran cuello de botella. Además, suelen ser difíciles de paralelizar en múltiples subproblemas. Tener como parámetros de entrada matrices o arreglos que se puedan partir fácilmente producen en overheads mínimos de cómputo y permiten aprovechar mejor las memorias caches y las herramientas de prefetching que brinda el hardware.

Item 4 ataca uno de los puntos críticos de esta arquitectura. Para poder operar con datos, se requiere que estén en la memoria de la placa, no en la memoria de propósito general de la computadora. Se debe, entonces, hacer copias explícitas entre las dos memorias, ya que ambas tienen espacios de direcciones independientes. Esta copia se realiza a través de buses que, a pesar de tener un gran *throughput*, también tienen una gran latencia (del orden de milisegundos). Por lo tanto, para minimizar el tiempo de ejecución de un programa usando GPGPUs, se debe considerar también el tiempo de transferencia de datos a la hora de determinar si el beneficio de computar en menor tiempo lo justifica. Las nuevas versiones de CUDA buscan brindar nuevas herramientas para simplificar este requerimiento, proveyendo espacio de direccionamiento único y memoria unificada [4], pero siguen siendo copias de memoria a través de los buses (aunque asincrónicas).

Estas características limitan enormemente la clase de problemas que una GPGPU puede afrontar, y suelen ser una buena heurística para determinar de antemano si vale la pena invertir el tiempo necesario para la implementación y ajuste fino.

2.8. Diferencia entre CPU y GPU - Procesadores especulativos

Hasta ahora, solo se consideraron a los GPUs de forma aislada, observando las prestaciones del hardware y una aproximación a la manera en que se escriben los programas para esta arquitectura. La esencia de GPGPU se puede apreciar mejor comparándola contra los motivos de la evolución de CPU, y los problemas que se fueron enfrentando los diseños siguiendo la historia de los componentes que fueron apareciendo en estos. Esto se mostró en la tabla **??** (página **??**), que detalla algunos de los eventos más importantes que aceleraron la *performance* de los CPU.

Lo clave es observar el siguiente patrón: *"no desechar algo que pudiéramos necesitar pronto"*, *"intentar predecir el futuro de los condicionales"*, *"intentar correr múltiples instrucciones a la vez porque puede llegar a bloquear en alguna de ellas"*.

Todos estos problemas han convertido al CPU en un dispositivo que gira alrededor de la especulación, de los valores futuros que pueden tener las ejecuciones, del probable reutilización de datos. En un CPU moderno (por ej. Intel Xeon E7-8800 [8]) las

unidades que verdaderamente realizan las operaciones lógico-aritméticas (las ALU) son muy pocas en comparación con la cantidad utilizadas para las operaciones de soporte.

En contraste, los dispositivos GPU son verdaderos procesadores de cómputo masivo. Están diseñadas para resolver constantemente operaciones muy bien definidas (instrucciones de punto flotante en su mayoría). Comparativamente con un CPU, las ALU de las GPU son bastante pobres y lentas. No funcionan a las mismas velocidades de clock (rara vez superan 1,1 GHz) y sus SP deben estar sincronizados entre sí. Pero la gran ventaja esta en la cantidad.

Un CPU cuenta con pocas ALU por core, dependiendo de la cantidad de cores y del tamaño de sus operaciones SIMD (alrededor de 16 cores por *die* de x86 es el tope de línea ofrecido actualmente, procesando de a 32 bytes simultáneamente). Un GPU cuenta con miles de ALUs en total (más de 2500 CUDA cores en una Tesla K20 [15]). El diseño de esta arquitectura concibe la escalabilidad cuantitativa de las unidades de cómputo como la característica esencial a tener, tanto por su énfasis fundamental, las aplicaciones gráficas, como para su aspecto de coprocesador numérico de propósito general.

Por contrapartida, los GPUs disponen de pocas unidades de soporte del procesamiento. Éstos no disponen de pipelines especulativos, el tamaño de las caches están a órdenes de magnitud de las de CPU, la latencia a las memorias principales de la GPU están a centenas de clocks de distancia, etc. La arquitectura supone que siempre va a tener más trabajo disponible para realizar, por lo cual en vez de intentar solucionar las falencias de un grupo de *threads*, directamente pone al grupo en espera para más adelante y continúa procesando otro warp de *threads*. Se puede notar que durante del diseño de la arquitectura CUDA, buscaron resolver el problema del cómputo masivo pensando en hacer más cuentas a la vez y recalcular datos, si fuera necesario. Esto es una marcada diferencia con respecto a los CPU, que están pensados en rehacer el menor trabajo posible e intentar mantener todos los datos que pueda en las memorias caches masivas.

Nuevamente, en este punto se puede apreciar el legado histórico de los CPU. Al tener que poder soportar cualquier aplicación, no pueden avocarse de lleno a una sola problemática. Para las arquitecturas GPGPU, el hecho de no tener que diseñar

un procesador de propósito general compatible con versiones anteriores, permitió un cambio radical a la hora de concebir una arquitectura de gran throughput auxiliar al procesador, no reemplazándolo sino más bien adicionando poder de cómputo [19].

Las arquitecturas Tesla, Fermi y Kepler conciben el diseño de un procesador de alto desempeño. Su meta principal es poder soportar grandes cantidades de paralelismo, mediante el uso de procesadores simétricos, pero tomando la fuerte restricción de *"no siempre tiene que andar bien"*. Es decir, los diseñadores suponen que el código que van a ejecutar esta bien adaptado a la arquitectura y no disponen casi de mecanismos en el procesador para dar optimizaciones post-compilación. Relajar esta restricción permite romper con el modelo de cómputo de CPU y definir nuevas estrategias de paralelismo, que no siempre se adaptan bien a todos los problemas, pero para el subconjunto de los desafíos que se presentan en el área de HPC y de vídeo juegos han probado ser un cambio paradigmático.

2.9. Idoneidad para la tarea

El problema de QM/MM enfrentado en este trabajo cuenta con múltiples operaciones matemáticas de gran volumen de cálculos. En particular, las operaciones matriciales constituyen los principales cuellos de botella en esta aplicación. Estas operaciones se realizan para varios grupos dentro de una grilla de integración (ecuación **??**), los cuales se pueden realizar de manera independiente (y por lo tanto en paralelo).

Para obtener los valores numéricos de densidad buscados en los puntos, se deben obtener las derivadas primeras y segundas, lo cual implica hacer múltiples operaciones de multiplicación matricial. Este tipo de problemas está estudiado fuertemente en la literatura debido a la multiplicidad de aplicaciones de diferentes campos que requieren de operaciones de álgebra lineal.

En nuestro caso, para un sistema se requieren miles de estas multiplicaciones entre matrices, algunas con matrices de más de 500^2 elementos. Como LIO es un proyecto de resolución numérica de QM/MM, los problemas enfrentados son casi, en su totalidad, operaciones de punto flotante. Luego, dadas las características de contar con un fuerte nivel de paralelismo en los cuellos de botella y de ser opera-

ciones mayormente de punto flotante, se determinó que el uso de GPGPU para este problema era promisorio, en comparación con arquitecturas de propósito general con menos poder de cómputo. La exploración original de esta arquitectura trajo buenos resultados, por lo que se prosiguió su análisis como un camino prometedor [11].

Capítulo 3

Algoritmos

Este capítulo, los conceptos del método de dinámica molecular presentados en la introducción son explicados en términos del algoritmo que los representa, haciendo énfasis en el paralelismo intrínseco que poseen, intentando abstraerse de cualquier implementación.

En la primera sección se describen loa conceptos del algoritmo base, describiendo todos los pasos de este, en particular todas las ecuaciones q no fueron detalladas en la introducción.

En las siguientes secciones del capítulo se explican las modificaciones que este trabajo introduce y las propiedades que se deben tener en cuenta para implementarlas utilizando arquitecturas paralelas.

Es importante destacar que las adaptaciones que se plantean y analizan en este trabajo constituyen una modificación puntual para realizar el calculo del potencial de Lennard-Jones y, por lo tanto, tienen la capacidad de ser incluidas directamente en cualquier algoritmo existente y combinadas junto con cualquier otro método de optimización. Por otro lado, como se vió en el capitulo 1, existen pocas variantes para optimizar el calculo de este potencial de no-unión y, dado que comprende el mayor aporte al costo computacional del método, esta implementación tendría un gran impacto en distintas implementaciones del método.

3.1. Esquema general del algoritmo

El algoritmo inicial sigue la especificacion que se detallo en el capitulo 1....

El proceso de actualización de coordenadas deberá calcular para cada partícula la nueva posición, la cual estará dada en funcion de la posicion y velocidad previa. La ecuación que se debe calcular en este paso es: Claramente este cálculo es totalmente independiente entre las partículas y por lo tanto puede hacerse la actualización totalmente en paralelo.

El proceso para actualizar la velocidad implica obtener un nuevo valor de velocidad en función del valor previo de esta y de la fuerza resultante calculada que actúa sobre la partícula acelerándola. La ecuación que se debe aplicar es:

A partir de las distancias entre las partículas se puede obtener la fuerza total resultante que actúa sobre cada partícula.

El cálculo de la fuerza entre un par de partículas responde a la siguiente ecuación:

$$\vec{F}(r) = \frac{\partial U(r)}{\partial (r)} =$$

Este valor da la fuerza resultante en la dirección de $\vec{r}$, es decir, en la dirección del vector distancia entre ambas partículas. Para obtener la suma tota de las fuerzas que actúan sobre una partícula es necesario sumar el aporte de la interacción contra todas las demás partículas del sistema, esto implica sumar fuerzas con distintas direcciones, lo que implica sumar los componentes x,y,z de los distintos vectores. Por lo tanto, el cálculo de la fuerza resultante que actúa sobre una partícula suele dividirse en tres pasos:

- En un primer paso se calcula el valor de la fuerza entre cada par de partículas de acuerdo a la ecuación anterior.

- En un segundo paso se descompone esta fuerza en sus 3 componentes (x,y,z). Para esto se calcula: $F_i = \frac{\partial U(r)}{\partial r}\frac{\partial r}{\partial i}$ para i= x,y,z

- Por último se realiza la suma de los componentes para todas las fuerzas que actúan sobre una partícula. $F_i = \sum F_i$

El calculo del potencial de interacción tiene una particularidad y es que este valor(y por lo tanto su derivada) es simétrico??? Es decir, la interacción entre una partícula a y b tiene el mismo modulo pero signo opuesto q el valor entre b y a. Esto tiene una implicación relevante en el calculo q debe realizarse ya que solo deberán calcularse interacciones para la mitad de los pares de partículas...reduciendo así considerablemente el número de cálculos.

Las limitaciones en este conjunto de pasos se deben a que tiene un orden de ejecucion N^2. Aun en arquitecturas altamente paralelas como las GPUs, este calculo es extremadamente costoso. Aun cuando el numero de procesadores en estas arquitecturas siga creciendo, es poco probable que se supere el numero de calculos que es necesario hacer para simular sistemas grandes. Es necesario, entonces, realizar aproximaciones extra para superar este limite.

El objetivo de esto es, en primer lugar, facilitar la implementacion de modificaciones/optimizaciones en el algoritmo relacionadas con el calculo del potencial de L-J. En segundo lugar, utilizando una implementacion propia se puede evaluar las ventajas y desventajas del uso de una tabla numerica de forma independiente a cualquier otra implementacion existente sobre GPU. Se podrá verificar facilmente que las variaciones en el tiempo y en la precision de los resultados son producto exclusivo de las modificaciones implementadas en cada paso.

3.2. Calculo usando tablas de valores del potencial

El algoritmo utiliza este potencial para obtener la interaccion entre cada par de particulas y derivar la fuerza resultante de esto. Ademas, esta la funcion permite evaluar la energia potencial total asociada a la conformacion. Para esto se realiza la suma del valor de la funcion entre todos los pares de particulas.

Basandose en la forma funcional del potencial, es razonable pensar que, utilizando una tabla conteniendo los resultados para un rango de valores definido y, aun sin tener ésta un tamaño excesivo, se podria obtener una buena aproximacion del resultado para cualquier distancia.

La primera aproximacion que involucra el uso de tablas se deduce de forma bastante directa a partir de la descripción general del algoritmo, se basa en utilizar una

tabla que tenga almacenados valores del potencial para un rango definido de distancias. Teniendo este tipo de valores almacenados es posible utilizarlos para calcular tanto las fuerzas resultantes (necesarias para la evolucion de la simulacion) como el valor de la enegia potencial total, el cual puede ser requerido por el usuario como resultado de la simulacion.

si bien el objetivo de este trabajo es optimizar la ejecucion de las implementaciones sobre GPU, el

Utilizando esta tabla, para obtener el valor de la energia potencial de interaccion entre un par de particulas simplemente se recupera el valor asociado a la distancia(r) que existe entre las particulas(o el valor tabulado mas cercano). Para poder obtener la fuerza resultante de esta interaccion es necesario conocer el valor de la derivada de la funcion potencial para la distancia correspondiente. Utilizando la tabla descita previamente se puede obtener el valor de la derivada mediante la tecnica de derivacion numerica. Para esto se obtienen primero dos valores de potencial a partir de la tabla, correspondientes a distancias de r+-x. Donde x es un valor discreto definido en base a..... Como se ve, esto implica una doble fuente de error, debido a una aproximacion tanto en la utilizacion de la derivada numerica como en la obtencion de los valores de potencial a partir de una tabla.

El esquema general del algoritmo se mantiene en todas las variantes. Las modificaciones estan centradas en la etapa de calculo del potencial, el contexto de este se mantiene siempre igual.

3.3. Calculo usando tablas de valores de la derivada

Como se mencionó en el capitulo 1, el resultado mas importante de la simulación es la trayectoria. Además, es la base de la simulacion y se debe calcular siempre. Dado que el valor mas importante que se obtiene a partir del potencial de L-J es la fuerza resultante de la interacción, se podría pensar que sería mas conveniente mantener en una tabla directamente los valores de la derivada. De esta forma, para obtener la fuerza correspondiente se busca el valor en la tabla que se encuentre mas cercano a la distancia de interacción.

Se debe tener en cuenta que, aún cuando sea posible realizar una buena aproximacion utilizando tablas, el algoritmo de dinamica molecular implica utilizar este valor para obtener las fuerzas y movimientos resultantes de la interaccion, lo cual afecta la posicion (y por lo tanto el valor del potencial asociado) en la proxima iteracion. De esta forma, los errores resultantes de la aproximacion por utilizar tablas de valores seran propagados a lo largo de una simulacion que involucra miles de pasos.....

Capítulo 4

Implementaciones

En este capítulo se describen los kernels implementados, incluyendo las variantes evaluadas. Además, se detalla cómo se tuvieron en cuenta los aspectos de performance expuestos en el capítulo 2.

4.1. Consideraciones previas

Una de las primeras decisiones que se deben considerar es la precisión que se utilizará para realizar los cálculos y acumular resultados. Dadas las caracterísiticas de la arquitectura GPU, esta decisión afectará la eficiencia de cualquier cálculo a realizar y por lo tanto es uno de los factores mas importantes a analizar.

Si bien es un factor importante, este ya ha sido estudiado usando otras implementaciones del método. En particular, se ha estudiado usando Amber[9]. El trabajo mencionado analiza un esquema mixto que implica almacenar los calculos de fuerzas entre particulas utilizando precisión simple y acumular en variables de precisión doble la suma de todos los aportes que componen la fuerza sobre una partícula. Asumiendo los resultados en performance y correctitud encontrados en Amber y, dado que es el software con el cual evaluaremos luego la calidad numérica lograda, utilizaremos, entonces, un esquema similar de precisión para nuestra implementación.

4.2. Esquema general de la implementación sobre GPU

El esquema general utilizado para implementar el algoritmo sobre GPU es el explicado en [5]. Sin embargo, en el trabajo mencionado, el potencial incluye componentes de interacción de unión entre partículas, por lo que los calculos requeridos son algo distintos. En esta sección se explica, entonces, cuales son las variantes que utilizamos para realizar cada paso.

Para el primer paso de calculo de distancias entre partículas, la implementación es bastante directa. La granularidad usada es de 1 thread por cada par de partículas y se usaron tamaños de bloques fijos dd 1024x1. Usando este esquema, con cualquier placa actual es posible lanzar en paralelo el calculo de todas las distancias para sistemas de hasta Xxx partículas. Teniendo en cuenta que el cálculo a realizar es muy simple, no presenta posibilidades de divergencia y dado el esquema de acceso a memoria, el kernel hace un uso óptimo de los recursos.

El paso correspondiente al calculo de las fuerzas entre particulas es quizas el mas complicado de implementar, y el numero de implementaciones existentes en la literatura es muy diversa, principalmente debido a que los calculos a realizar dependen de las interacciones que se estan utilizando y estás dan lugar a una gran variedad de posibles optimizaciones [5][6][20] (citar los papers de amber y el de Accelerating Molecular Dynamic Simulation on Graphics Processing Units)

En el caso de interacciones non-bond, como las que se analizan en nuestro trabajo, la principal optimizacion deriva de la propiedad simetrica que presenta el potencial de interaccion, y por lo tanto la fuerza resultante, son simetricas

En nuestra implementación, esta parte es central ya que da lugar luego a diversas modificaciones. El presente trabajo, entonces, intenta seguir una implementacion estandar de forma tal que las modificaciones evaluadas tengan un carácter mas general y no dependan de ninguna

Para hacer uso de la propiedad de simetria, entonces, el calculo de la fuerza resultante de la interaccion solo se realiza sobre N2/2 pares, y el resto se deriva negando este valor.

Para descomponer las componentes el cálculo se realiza usando un thread por

cada par de partículas. Y para hacer la suma de la fuerza resultante, este se hace directamente sumando todos los valores de una fila. El acceso en este caso es coalescente ????? Y si hago una reducción ???

Los pasos siguientes son, bastante directos. La aceleración se calcula en base a la fuerza resultante usando un thread por cada partícula. La actualización de coordenadas también se realiza usando un thread para cada partícula. En este caso el único valor q se lee es el de

4.3. Implementación usando tabla de valores potenciales

4.4. Implementación usando tabla de derivadas

4.5. Implementacion sobre CPU

Con el fin de poder hacer una comparación real de los aspectos asociados a la performance se desarrolló además una implementación que realiza el cálculo de fuerzas(o solo el cálculo de derivadas???) sobre CPU.

Esta implementación solo modifica la forma en que se calcula la derivada del potencial, la cual se ejecuta ahora sobre la CPU, por lo tanto los datos de distancia y potenciales deben estar accesibles en memoria para que este cálculo pueda ser realizado.

Se implementaron variantes equivalentes para el calculo usando la ecuación derivada, usando una tabla de valores potenciales, y usando una tabla de valores de derivada. Deberán tenerse en cuenta a la hora de usar esta implementación todos los aspectos asociados a la transferencia de datos entre la memoria del dispositivo y la memoria del host. En el próximo capitulo se verá como esta implementación es utilizada para demostrar las ventajas q aporta la arquitectura GPU.

Para implementar este mecanismo hay varias propiedades del sistema de memoria que se puede usar y que se deben analizar..... la memoria de textura provee el ajuste automatico en los limites de la tabla. Además,

permite obtener el valor resultante de la interpolacion, lo cual si bien no es necesario, ayuda considerablemente en la precision.

4.6. Implementación usando tabla de derivadas

Capítulo 5

Resultados

5.1. Propiedades de los sistemas de evaluación

Claramente, el factor mas importante del sistema que afectará a la ejecución es el número total de partículas de éste. Los tamaños de sistemas utilizados fueron tres, y para

Además del tamaño, hay otros factores del sistema que se tuvieron en cuenta para realizar una correcta evaluación y que deben ser Para tener en cuenta cualquier posible diferencia relacionada a los tipos de partículas, la asignación de tipos se hizo de forma aleatoria entre un total de 37 tipos existentes.

Además de las posiciones y tipos de particulas, es posible especificar velocidades iniciales....

En cuanto a los parámetros de ejecución se debe tener en cuenta que a medida que la simulación avanza, las partículas se deben tener en cuenta, entonces, dos aspectos que asocian la ejecucion con los parametros del sistema: -las velocidades iniciales(ya descrito antes) -el largo de la simulación y el uso de periocidad: es importante tener en cuenta que, si no se usa periodicidad, luego de cierto punto las particulas .

Este efecto se verá en cualquier simulación independientemente de las propiedades del sistema inicial y, claramente es algo a tener en cuenta ya que afecta la correcta evaluación, la aproximación que usamos es aplicar condiciones periódicas de contorno en todas las simulaciones, independientemente del tamaño del sistema. Al usar condiciones periodicas El tamaño de la caja es, entonces, otro factor importante

. De esta forma, a menos que se especifique lo contraro, se define como tamaño de la caja al valor de cutoff/2.

Todos estos factores deben ser tenidos en cuenta a la hora de evaluar las implementaciones. , las situaciones mencionadas que afectan a la ejecucion sirven para demostrar propiedades de las implementaciones y como ejecuciones de control, para En las próximas secciones, cuando sea relevante, se especificarán las propiedades del sistema y de la ejecucion que se utilizaron.

5.2. Performance

Los sistemas detallados en la sección anterior se utilizaron para evaluar las distintas implementaciones en cuanto a performance..

5.2.1. Evaluación de las implementaciones

5.2.2. Efectos del tamaño de bloque

5.3. Calidad numérica

Capítulo 6

Conclusiones

Bibliografía

[1] *OpenACC 2.0 Specification*, . URL http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf.

[2] *OpenMP 4.0 Specification*, . URL http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[3] Nathan Bell y Jared Hoberock. Thrust: A productivity-oriented library for cuda. *GPU Computing Gems*, 7, 2011.

[4] Rob Farber. *CUDA application design and development*. Elsevier, 2011.

[5] Mark S Friedrichs, Peter Eastman, Vishal Vaidyanathan, Mike Houston, Scott Legrand, Adam L Beberg, Daniel L Ensign, Christopher M Bruns, y Vijay S Pande. Accelerating molecular dynamic simulation on graphics processing units. *Journal of computational chemistry*, 30(6):864–872, 2009.

[6] Andreas W Götz, Mark J Williamson, Dong Xu, Duncan Poole, Scott Le Grand, y Ross C Walker. Routine microsecond molecular dynamics simulations with amber on gpus. 1. generalized born. *Journal of chemical theory and computation*, 8(5):1542–1555, 2012.

[7] John L. Hennessy y David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th ed$^{\text{ón}}$., 2011. ISBN 012383872X, 9780123838728.

[8] Intel. Intel xeon e7 8800 specifications. http://ark.intel.com/products/53580/Intel-Xeon-Processor-E7-8870-30M-Cache-2_40-GHz-6_40-GTs-Intel-QPI. Accessed: 2014-08-22.

[9] Scott Le Grand, Andreas W Götz, y Ross C Walker. Spfp: Speed without compromise—a mixed precision model for gpu accelerated molecular dynamics simulations. *Computer Physics Communications*, 184(2):374–380, 2013.

[10] William R. Mark, R. Steven Glanville, Kurt Akeley, y Mark J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003. ISSN 0730-0301. doi:10.1145/882262.882362.

[11] Matias A Nitsche. *Aceleración de cálculos de estructura electrónica mediante el uso de Procesadores Gráficos Programables.* Proyecto Fin de Carrera, Universidad de Buenos Aires, 2009.

[12] Nvidia. Cuda cublaslibrary. *NVIDIA Corporation, Santa Clara, California*, 2008.

[13] Nvidia. Cuda cufft library. *NVIDIA Corporation, Santa Clara, California*, 2010.

[14] NVIDIA. Kepler GK110 whitepaper. 2012. URL http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[15] NVIDIA. Kepler GK110 family. 2013. URL http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf.

[16] Nvidia. Cuda programming guide. Inf. téc., Nvidia, 2015.

[17] Nvidia Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Inf. téc., Nvidia Corporation, 2009. URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf.

[18] David Patterson. The top 10 innovations in the new nvidia fermi architecture, and the top 3 next challenges. Inf. téc., NVIDIA, 2009. URL http://www.nvidia.com/content/pdf/fermi_white_papers/d.patterson_top10innovationsinnvidiafermi.pdf.

[19] Peter Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. Inf. téc., Nvidia Corporation, 2009. URL http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIAFermi-TheFirstCompleteGPUComputingArchitecture.pdf.

[20] Romelia Salomon-Ferrer, Andreas W Götz, Duncan Poole, Scott Le Grand, y Ross C Walker. Routine microsecond molecular dynamics simulations with amber on gpus. 2. explicit solvent particle mesh ewald. *Journal of Chemical Theory and Computation*, 9(9):3878–3888, 2013.

[21] Alan Tatourian. Nvidia gpu architecture and cuda programming environment. 2013. URL http://tatourian.com/2013/09/03/nvidia-gpu-architecture-cuda-programming-environment.

[22] Top500. November 2001 list. http://www.top500.org/lists/2001/11/. Accessed: 2014-08-22.

[23] Nicholas Wilt. *The CUDA Handbook: A comprehensive guide to GPU Programming*. Pearson Education, 2013.

[24] Henry Wong, M-M Papadopoulou, Maryam Sadooghi-Alvandi, y Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. En *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, págs. 235–246. IEEE, 2010.