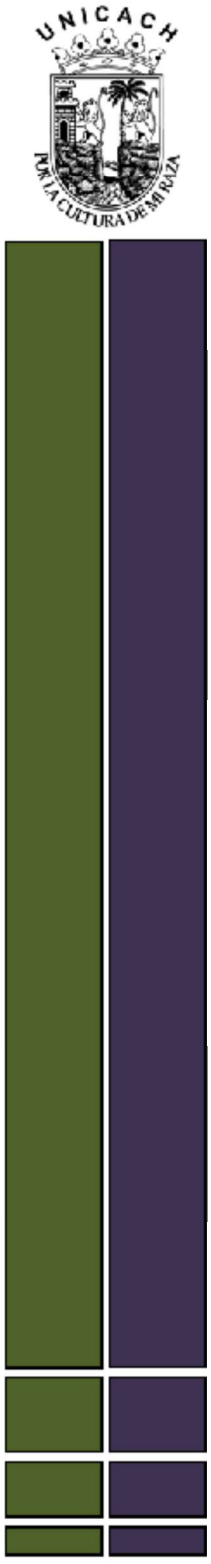


Universidad de Ciencias y Artes de Chiapas

FACULTAD DE INGENIERÍA

T E S I S

"IMPLEMENTACIÓN DE DINÁMICA MOLECULAR EN UN PROGRAMA PARA GPU"

QUE PARA OBTENER EL GRADO DE

Maestro en Ciencias en Desarrollo Sustentable

PRESENTA:

Agustín Escobar López

Comité Tutoral

Director

Dr. Juan Andrés Reyes Nava

Co-Director

Dr. Feliú Sagols Troncoso

Asesores

Dr. Alejandro Nettel

Dr. Carlos Manuel Lara García

Tuxtla Gutiérrez, Chiapas. Marzo, 2016.

Agradecimientos

Este documento es el resultado de una ardua pero fascinante labor de investigación, representa la culminación de una etapa en gran medida enriquecedora tanto en lo personal como en lo académico. Sin embargo, estas dinámicas carecen de importancia si no generan un aporte real a la comunidad a la que pertenecemos, por ello es necesario darle un sentido a lo que se está investigando. Este trabajo de investigación tiene como objetivo contribuir a las investigaciones referentes al área de materiales dentro de la Universidad. Para que pudiera concretarse con éxito, muchos estuvieron involucrados, primeramente agradezco a Dios que nos permitió llegar hasta donde hemos llegado hoy; agradezco a mi familia el apoyo y la paciencia, así también a Gaby que estuvo ahí a lo largo de este extenueante período. Así mismo, agradezco a la Universidad de Ciencias y Artes de Chiapas, que me brindó el espacio y lo necesario para llevar a cabo este trabajo, que no hubiera sido posible en otras circunstancias. Por la disposición y el interés, a mi comité revisor: los Doctores Alejandro Nettel y Carlos Lara García, de esta universidad, por su detallada revisión del manuscrito y por la solidez de sus críticas; especialmente agradezco al Dr. Feliú Salgols, del Departamento de Matemáticas del CINVESTAV, quien contribuyó a la realización del presente trabajo incluso impartiendo en nuestra universidad un curso sobre programación, su disposición fue uno de los motores para que este trabajo se llevara a cabo. Finalmente, el Dr. Andrés Reyes Nava, no habría sido posible desarrollar esta tesis sin su experiencia en el desarrollo de códigos de dinámica molecular, usando el paradigma de programación Paso de Mensajes (modelo complementario al empleado en esta Tesis). Gracias por todas las enseñanzas a lo largo de este tiempo. Terminar este tipo de proyectos no es sencillo, pero gracias al esfuerzo compartido, fue posible. Gracias.

Resumen

Los laboratorios de cómputo de alto desempeño son una herramienta clave para el estudio de materiales a nivel molecular, permitiendo investigar propiedades muy difíciles o costosas en laboratorios de experimentación [1] [2] [3]. Sin embargo, estos laboratorios de cómputo presentan altos costos de adquisición y mantenimiento, un alto gasto energético y - cuando se carece de una capacidad instalada considerable - los tiempos de ejecución son muy largos en la mayoría de las simulaciones. Recientemente, el uso de tarjetas gráficas en investigaciones científicas ha permitido evitar los problemas mencionados, aumentando el rendimiento de los cálculos y simulaciones haciendo posible investigar fenómenos a escala real. Para su programación, la compañía NVIDIA ha desarrollado el modelo de programación en paralelo CUDA, el cual permite eficientar la ejecución de programas dentro de las tarjetas gráficas. Sin embargo, el uso de esta tecnología es difícil pues es de reciente creación, y aunque en el mercado es posible encontrar códigos implementados en tarjetas gráficas, la inexistencia de códigos específicos que determinen propiedades no estudiadas con anterioridad en el área de estudio de materiales es un problema que dificulta la generación de aportes científicos [1] [3]. Así, en este trabajo se presenta la construcción y codificación en CUDA-C de un programa semilla que presenta los cálculos fundamentales para la determinación del movimiento de los átomos de una nanopartícula. Este código serviría como base para la generación de cualquier otro código partiendo de éste. De esta forma, se crea un puente que permite asimilar esta nueva tecnología e implementarla en investigaciones futuras. Al analizar, se realiza una comparación de rendimiento y velocidad de procesamiento entre una GPU y una CPU para validar su utilización.

Keywords: CPU, GPU, tarjetas gráficas, CUDA, dinámica molecular,

paralelismo, paradigma de programación.

Abstract

High Performance Computing (HPC) laboratories are an important tool in the study of materials in a molecular level, allowing to investigate properties that are impossible to study in conventional laboratories [1] [2] [3]. These HPC laboratories have a high cost of acquisition y maintenance, a high energy comsumption and a long runtimes of many simulations if the installed capacity of the laboratory isn't big enough. Recently, the use of graphics cards avoids these problems, increasing the performance of the calculations and simulations and making possible the research of real-scale phenomena. To facilitate the programming of graphics cards, NVIDIA Corporation has developed the CUDA parallel programming model, which enables more efficient parallel programs implementation inside the graphics cards. But, the use of this technology is very hard due to it is newly created and, although in the market it is possible to find codes and softwares for simulations using graphics cards, the lack of specific codes that determine properties not previously studied in the area of study of materials is a latent problem that hinders the generation of scientific contributions [1] [3]. Thus, it is presented the construction and codification in CUDA of a seek program that contains the fundamental calculations for the determination of the movement of the atoms of a nanoparticle. This code is the basis of the construction of any other code inside the research area. In other words, this reseach build a bridge for the assimilation of this new technology and its implementation in future researches. At the end, a comparison of performance and processing speed between a GPU and a CPU is done, in order to validate the use of GPUs.

Keywords: CPU, GPU, graphic card, CUDA, molecular dynamic, paralelism, programming paradigm.

Índice general

Índice de figuras

Capítulo 1

El problema

1.1. Planteamiento del problema

La investigación de los materiales requiere frecuentemente de códigos de cómputo de alto desempe no que calculen propiedades nunca antes determinadas por los programas existentes [4] [5]. Las investigaciones que requieren tales códigos exigen la solución del problema de su inexistencia. Este trabajo de investigación resuelve este problema mediante la creación de un código semilla propio, usando el modelo de programación CUDA (*Compute Unified Device Arquitecture*, Arquitectura Unificada de Dispositivos de Cómputo), para generar partiendo de él, cualquier otro programa para GPU (*Graphic Processing Unit*, Unidad de Procesamiento Gráfico). Específicamente, se codificó un programa para determinar el movimiento de los átomos de una nanopartícula. La generación de nuevos programas a partir del código desarrollado está garantizada por el hecho de que la investigación de cualquier material necesita de la descripción de su movimiento atómico.

1.2. Justificación

El análisis de las interacciones entre los elementos de un sistema (físico, químico o biológico) en un período definido de tiempo, es la base de una gran variedad de investigaciones en el estudio de materiales. Para llevar a cabo estos estudios es necesario contar con los laboratorios para realizar experimentación. Sin embargo, factores como la infraestructura adecuada, el costo del mantenimiento de los equipos de laboratorio, la administración y costo de insumos y el manejo de personal, dificultan la construcción y puesta en marcha de estos laboratorios. Los centros tradicionales de supercómputo (nodos de CPUs) son el recurso más utilizado actualmente para cubrir las necesidades de capacidad de cálculo requerida, pero la adquisición de equipos, mantenimiento, refrigeración y administración de las tareas a ejecutar también representan un costo elevado.

Por otro lado, como tecnología emergente, las GPUs muestran capacidades útiles par investigar teóricamente sistemas y procesos en la escala de tama no y tiempo en que se manifiestan en el laboratorio y en la naturaleza [1] [3]. Su uso para realizar simulaciones de fenómenos físicos presenta ciertas ventajas en comparación con el uso de CPUs:

1. El costo de adquisición de una GPU es mucho menor en aproximadamente 30 % el costo de un equipo CPU.

2. El mantenimiento de una estación de nodos de CPU que cuente con la misma capacidad de cálculo de una tarjeta Kepler K20 -la utilizada para esta investigación- es mucho mayor.

3. El rendimiento de una GPUs en operaciones de punto flotante es mucho mayor

[6] (ver capítulo 3, figura 3.2).

Sin embargo, las dificultades del uso de tarjetas gráficas surgen en la codificación de programas para su implementación en paralelo. La generación de códigos nuevos requiere de conocimientos avanzados tanto de programación tradicional como de las instrucciones para su ejecución en paralelo, es decir, la curva de aprendizaje es muy alta. Así mismo, la modificación de códigos existentes -cuando es posible- resulta una tarea muy compleja. A partir de los factores anteriores, surge la necesidad de un código semilla que simule el componente básico de interacción entre los elementos de un cúmulo, es decir, las fuerzas interatómicas, y que además utilice los elementos fundamentales de programación en paralelo en tarjetas gráficas. Teniendo como base un código semilla, la identificación de los elementos de programación a modificar o a nadir es clara y sencilla. De este modo, el código construído representa el punto de partida para la generación de cualquier otro código en el paradigma de CUDA.

Por otro lado, el uso adecuado de recursos energéticos es una de las problemáticas actuales más graves a nivel global. El estudio de materiales que permiten el funcionamiento de los sistemas energéticos es una pieza clave para la generación de soluciones a esta situación. Con el uso de cómputo de alto rendimiento como herramienta, la generación de contribuciones científicas se acelera (por el uso de procesamiento en paralelo) y se vuelve accesible para instituciones que no pueden contar con un laboratorio experimental. Así, la generación de conocimento científico en este tema se ve acelerada.

Del mismo modo, en aspectos como los sistemas de enfriamiento, uso frecuente y mantenimiento de un laboratorio de GPU's se aprecia el menor consumo de energía en comparación con un laboratorio tradicional de supercómputo con la misma capacidad de procesamiento. Además, el uso de simulación en lugar de experimentos reales -cuando es posible- propicia un ahorro, no sólo en consumo de energía, sino también en el uso de

reactivos y materiales. Aunado a esto, los costos económicos de puesta en marcha, uso y mantenimento de equipos con GPU es también considerablemente menor comparado con centros de cómputo con CPU y laboratorios experimentales. En el ámbito social, las investigaciones en el área de materiales repercutirán en la generación de sistemas energéticos más eficientes [7] y que usen energía renovable como alternativa al uso de recursos fósiles para producción de energía. Así mismo, las simulaciones llevadas a cabo con GPUs son una herramienta fundamental en la creación de elementos que puedan ayudar a solucionar más eficientemente diversos problemas sociales. Se pueden mencionar aportes en el área de salud [8] [9], en problemas ambientales [10] [11], entre otros. De este modo, el uso de GPU's es justificado en el campo de la sustentabilidad. El asimilar y utilizar adecuadamente esta tecnología es clave para lograr un desarrollo sustentable.

1.3. Hipótesis

Un programa de Dinámica Molecular paralelizado para GPU contiene los elementos básicos del lenguaje de programación CUDA-C.

La elección del modelo de interacción es vital para la generación de un código que, paralelizado para GPU, contenga las elementos básicos del paradigma de programación de CUDA. Estos elementos se engloban en tres categorías que describen a CUD [6]: la jerarquía de elementos de procesamiento, la jerarquía de memoria y la comunicación entre CPU y GPU.

1.4. Objetivos

General:

Construir un código semilla propio, en lenguaje CUDA-C, para generar partiendo de él, cualquier otro programa para GPU.

Específicos:

1. Construir un programa de cómputo para GPU en lenguaje CUDA-C que determine el movimiento de los átomos en un cúmulo libre de un gas noble.

2. Describir los elementos básicos y el procedimiento para codificar fácilmente un programa cualquiera en lenguaje CUDA-C.

Capítulo 2

Antecedentes

2.1. Arquitectura de Supercómputo Disponible

En el Centro de Investigación y Desarrollo Tecnologico en Energías Renovables de la UNICACH se cuenta con un equipo de supercómputo llamado *Tzolkin*, vocablo de origen maya que en términos generales significa *poner orden al caos*. *Tzolkin* posee dos Procesadores AMD Opteron 6272 instalados, cada uno con velocidad de ejecución de 269 GFLOPS, 16 núcleos de procesamiento cada una y un caché L3 de 16 MB y dos tarjetas gráficas Kepler K20 cada una con 2496 núcleos de procesamiento y una velocidad de ejecución de $1,17$ TFLOPS. Para la instalación de cada AMD 6272 se requiere de una tarjeta madre, elementos de entrada y salida, gabinete para su colocación, entre otros dispositivos. Mientras cada GPU K20 necesita úncicamente de una ranura PCIe para su utilización. Para fines de comparación entre la CPU y la GPU, es necesario hacer notar que cada núcleo de la tarjeta gráfica Kepler K20 tiene un rendimiento de $0,469$ GFLOPS (ya que en total, 2496 núcleos poseen un rendimiento de $1,17$ GFLOPS) siendo el rendimiento de un único núcleo de CPU AMD 6272 es de alrededor de $15,5$ GFLOPS, es decir, cada núcleo de la GPU en cuestión es aproximadamente 33 veces más lento. Sin embargo, al comparar los equipos, el desempe no de la GPU K20 es $4,3$

veces mayor.

2.2. Códigos de Dinámica Molecular

Actualmente existe una gran variedad de programas para simulación y análisis de fenómenos a nivel molecular. Uno de los métodos más utilizados en estos programas es Dinámica Molecular. Naturalmente, la mayoría de estos programas están codificados para su implementación en nodos de CPUs con algunas alternativas para su ejecución en GPUs. Así, las aplicaciones de Dinámica Molecular destinadas a ser utilizadas en tarjetas gráficas no son tan numerosas como las elaboradas para CPUs, y son aún menos las que se elaboran utilizando CUDA. A continuación se mencionan algunos ejemplos de los programas para Dinámica Molecular más utilizados.

- LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) - Programa de Dinámica Molecular desarrollado para ser implementado en nodos de CPU, con soporte para utilizar GPU, usando la biblioteca MPI para la comunicación entre los núcleos. El potencial del uso de LAMMPS se encuentra en el estudio de metales y semiconductores, así como de biomoléculas y polímeros [12]. Actualmente está manejado por Sandia National Laboratories, un laboratorio del Departamento de Energía de Estados Unidos.

- NAMD (NAnoscale Molecular Dynamics) - El desarrollo de este programa se centró en simulaciones de estructuras biológicas en escalas de millones de átomos [13]. NAMD escala del orden de 100 hasta 500000 núcleos de acuerdo a las necesidades de la simulación; además trabaja en conjunto con otros programas como VMD y AMBER.

- VMD (Visual Molecular Dynamics) - Este programa es propiamente un visualizador de moléculas, pero incluye herramientas para el análisis de los resultados de simulaciones de Dinámica Molecular y de datos de naturaleza diversa. VMD es desarrollado por el grupo creador de NAMD y tiene soporte para GPUs.

- AMBER (Assisted Model Building with Energy Refinement) - Es un paquete de simulación de biomoléculas con Dinámica Molecular usando campos de fuerza. Está escrito en FORTRAN 90 y C, y actualmente, tiene soporte para ser implementado en GPUs.

- GROMACS (GROningen MAchine for Chemical Simulations) - Este paquete de códigos de simulación de Dinámica Molecular utiliza específicamente las ecuaciones de movimiento de Newton para determinar el movimiento atómico en sistemas compuestos por millones de partículas, ya sean biomoléculas (lípidos y proteínas) o de sistemas no-biológicos (polímeros). GROMACS tiene soporte para trabajar con CUDA en GPUs de NVIDIA; para la comunicación en paralelo usualmente manejan la biblioteca MPI. También incluye herramientas de análisis de datos.

- Abalone - Este programa realiza la simulación de la dinámica de biopolmeros, utiliza tanto Dinámica Molecular como Monte Carlo. Está dise nado para simular proteínas y DNA con campos de fuerza compatibles con el programa AMBER.

- TeraChem - Este paquete de códigos para química cuántica está dise nado para correr en tarjetas gráficas NVIDIA y para utilizar CUDA, siendo una de las caraterísticas más relevantes el uso de DFT y Dinámica Molecular ab initio para nanopartículas.

Existen otros programas y diferentes modificaciones a los que ya se mencionaron para adaptarse a las necesidades del usuario. Desde que las tarjetas gráficas se con-

virtieron en una herramienta de alto desempe no para la investigación científica, la mayoría de los creadores de estos programas, originalmente para nodos de CPU, están generando versiones que utilicen GPUs y eventualmente CUDA. Varios de estos paquetes de programas son código libre, aunque algunos de los más importantes tienen un costo ecónomico de adquisición, lo que podría significar un obstáculo para conseguirlos. Todos estos programas funcionan como cajas negras, es decir, son limitadas las caraterísticas que el usuario puede manipular al inicio o durante el proceso de simulación. Estas limitaciones son impuestas por el desarrollador del código. Incluso, de acuerdo a la investigación que se esté realizando, existe la posibilidad de que los resultados puedan no ser tan específicos como el usuario desearía. Una alternativa presentada por parte de los creadores, es el acceso al código fuente de los programas, con el fin de poder realizar las modificaciones que se necesiten. Sin embargo, el rastreo y modificación de las instrucciones adecuadas para tener los resultados esperados es una tarea laboriosa que requiere de un nivel alto de conocimientos de programación que, en mucho de los casos, no es el área de experiencia del usuario de los programas. Entonces las alternativas se limitan a ir adecuando los procesos de investigación y simulación de acuerdo a la gama de programas existentes o generar códigos propios que ataquen problemas específicos en las investigaciones en curso.

2.3. Códigos CUDA-C libres

El fabricante NVIDIA facilita y alienta el uso de sus GPUs distribuyendo de forma gratuita un conjunto de programas ejemplo codificados en CUDA-C [14]. Estos programas vienen incluídos en el paquete de instalación de CUDA en sus versiones más recientes con la finalidad de mostrar las diferentes capacidades de CUDA en problemas

muy específicos. Dentro de este paquete se encuentran códigos ejemplo de la forma de implementar diversos procesos en códigos paralelizados para GPUs, por ejemplo, operaciones con matrices de una enorme cantidad de elementos, el producto escalar entre dos vectores, uso de librerías propias, incluso códigos sencillos de visualización de movimiento de partículas, entre otras. Pese a que estos programas son muy sencillos, son clave para entender como se ejecutan las instrucciones en las tarjetas gráficas y las potencialidades de la organización de las unidades fundamentales de procesamiento y sus espacios de memoria. Cada uno de estos programas describe la implementación del paradigma de programación CUDA [14], y permite reconocer las diferencias con respecto a otras estructuras de paralelización que se han desarrollado para cálculo científico, como el modelo de programación MPI [15].

2.4. Código Producto Escalar

Uno de los códigos ejemplos que fueron significativos para el desarrollo de este trabajo fue scalarProd.cu. Este programa determina el producto escalar de dos vectores. Para ello, los vectores son enviados a la GPU, donde las unidades fundamentales de cálculo, denominados hilos, se organizan procurando que la carga de trabajo entre ellos sea lo más equitativa posible. Los hilos a su vez, son organizados en bloques. En la figura 2.1 se muestra la parte del código que se encarga de la distribución de la carga de trabajo y la acumulación por bloques de los resultados del producto escalar.

```
...
for(int iAccum = threadIdx.x; iAccum < ACCUM_N; iAccum += blockDim.x){
   float sum = 0;

   for(int pos = vectorBase + iAccum; pos < vectorEnd; pos += ACCUM_N)
      sum += d_A[pos] * d_B[pos];

      accumResult[iAccum] = sum;}
...
```

Figura 2.1: Sección del código scalarProd.cu para distribución de tareas entre hilos [14].

Dentro de cada bloque, se efectúa una acumulación de los resultados parciales en un arreglo. Mediante una reducción tipo árbol, se obtiene un sólo resultado parcial por bloque. Al finalizar, se genera un arreglo de resultados que es enviado a la CPU. Durante el proceso, todos los hilos tienen acceso a cualquier elemento de los vectores. Éstos se copian a cada bloque para que los hilos tengan acceso rápido a ellos. Los resultados parciales se almacenan en la memoria de cada bloque. Al final de todas las operaciones, el resultado de cada bloque se envía a la memoria global de la GPU, desde donde se envían los datos procesados de regreso a la CPU. Esta técnica de organización de la carga de trabajo y el manejo de las memorias fue adoptado por el código semilla para su implementación en el cálculo de las fuerzas. En la figura 2.2 se muestra la reducción tipo árbol realizada en el primer ciclo for y la recolección de datos por el hilo 0, tarea que se realiza mediante la sentencia condicional if.

```
...
for(int stride = ACCUM_N / 2; stride > 0; stride >>= 1){
    __syncthreads();

    for(int iAccum = threadIdx.x; iAccum < stride; iAccum += blockDim.x)
       accumResult[iAccum] += accumResult[stride + iAccum];}

    if(threadIdx.x == 0) d_C[vec] = accumResult[0];
...
```

Figura 2.2: Sección del código scalarProd.cu para reducción tipo árbol y recolección de resultados [14].

2.5. Composición de Dinámica Molecular

Todo programa que determine el movimiento de los átomos en un material tiene dos elementos básicos: el cálculo de las fuerzas atómicas y el cálculo del movimiento de los átomos. El primero depende de la naturaleza del material, por lo que su implementación en supercómputo, para la gran mayoría de los materiales, es inexistente. Así, una aportación importante de la tesis es la descripción de la implementación del cálculo de las fuerzas en GPUs. El segundo cálculo es el mismo para todos los materiales, la determinación numérica de la solución de las ecuaciones de movimiento de los átomos. Esta parte está documentada en detalle en libros [16] [17] [18]. Así, la implementación de este cálculo se redujo a su extracción de un programa previamente codificado GUPTA.F y en su traducción a lenguaje C. En virtud de la capacidad de las GPUs para realizar millones de cálculos se opta por la adquisición de seis tarjetas gráficas y se inicia con la codificación del programa de dinámica molecular, utilizando un potencial diferente al del código previo, con el fin de poner a prueba el rendimiento de las GPUs.

2.6. Dinámica molecular y gases nobles

Dinámica molecular es un método numérico para determinar el movimiento de los átomos de un sistema (físico, químico o biológico) durante un período de tiempo dado T. El movimiento de los átomos se determina resolviendo de forma numérica sus ecuaciones de movimiento sobre cada uno de los elementos de la partición T. Con dinámica molecular es posible calcular propiedades de sistemas físicos y químicos como la energía libre, entropía, presión, temperaturas de cambio de fase, etc. [19] [20] [21]. En sistemas biológicos es posible describir el comportamiento de proteínas y moléculas complejas bajo condiciones determinadas [22] [23]. Existen modelos de interacción atómica para cada tipo de sistema, cada uno de ellos se describe mediante la energía de interacción, frecuentemente denominada potencial de interacción. En este trabajo de investigación se determinaría el movimiento de los átomos de gases nobles o inertes, éstos son: helio, neón, argón, kriptón, xenón y radón. Las fuerzas interatómicas de los cúmulos de gases nobles son muy débiles, por consecuencia, tienen puntos de fusión y de ebullición muy bajos. Además los átomos de gases nobles presentan estabilidad química, es decir, estáan completos eléctricamente, así que no intercambian electrones. El estudio de los gases inertes, gracias a su estabilidad, es relativamente sencillo y existen resultados de experimentos y simulaciones realizadas bajo otros métodos. Para describir la interacción entre los átomos en el programa codificado se utilizaría el potencial de Lennard-Jones [16], el cual determina la energía potencial de dos aatomos separados a una distancia determinada. La existencia de reportes de la determinación experimental de las propiedades de sistemas mediante Lennard-Jones, proporciona una sólida base para evaluar la eficacia del modelo, al comparar los resultados obtenidos en el experimento y en la simulación.

Capítulo 3

Elementos Básicos

3.1. Computación paralela

La computación en paralelo se centra en dividir un proceso en subprocesos independientes que son posteriormente solucionados simultáneamente. Existen dos tipos generales:

- Paralelismo de datos. Consiste en subdividir el conjunto de datos de entrada de un problema, de manera que a cada procesador le corresponda un subconjunto de esos datos. Cada procesador efectuaría la misma secuencia de operaciones que los otros procesadores sobre su subconjunto de datos asignado. En resumen: se distribuyen los datos y se replican las tareas.

- Paralelismo de tareas. Consiste en asignar distintas tareas a cada uno de los procesadores de un sistema de cómputo. En consecuencia, cada procesador efectuaría su propia secuencia de operaciones.

Resolver los subproblemas independientes en paralelo genera la impresión de obtener mayor rapidez respecto a resolver todo el problema secuencialmente. Sin embargo, es necesario considerar algunos aspectos para poder validar dicha aseveración: La natu-

raleza del problema, la relación entre los datos, la posibilidad de dividir los datos o el problema y la independencia de los resultados. Este método de resolución se utiliza principalmente para evitar el problema del cuello de botella en la ejecución de instrucciones, que son muy recurrentes en simulaciones de modelos moleculares, climáticos[24] o económicos[25], es decir, los que tienen una amplia componente paralela.

3.2. Taxonomía de Flynn

Michael Flynn propuso en 1972 una clasificación de computadoras de acuerdo a su arquitectura. Flynn consideró dos aspectos: el flujo de instrucciones concurrentes y el flujo de datos. Las cuatro clasificaciones son las siguientes:

1. SISD (Single Instruction, Single Data): Computadora que no cuenta con paralelismo en el flujo de instrucciones o en el flujo de datos.

2. SIMD (Single Instruction, Multiple Data): Computadora en la cual se procesan varios flujos de datos dentro de un único flujo de instrucciones para realizar operaciones simultáneamente.

3. MISD (Multiple Instruction, Single Data): Este tipo de computadoras se utiliza en paralelismo redundante, por ejemplo, en navegación aérea, donde se necesitan varios sistemas de respaldo en caso de que uno falle.

4. MIMD (Multiple Instruction, Multiple Data): Consiste en varios procesadores autónomos que ejecutan simultáneamente diferentes instrucciones sobre varios flujos de datos.

En la imagen 3.1, se sintetizan las diferentes categorías de la clasificación anterior. El término *Bando de instrucciones* se refiere al flujo de instrucciones que debe ejecutarse

en la *unidad de procesamiento* (UP) y *Banco de datos* indica el flujo de datos que sería procesado en la unidad de procesamiento.

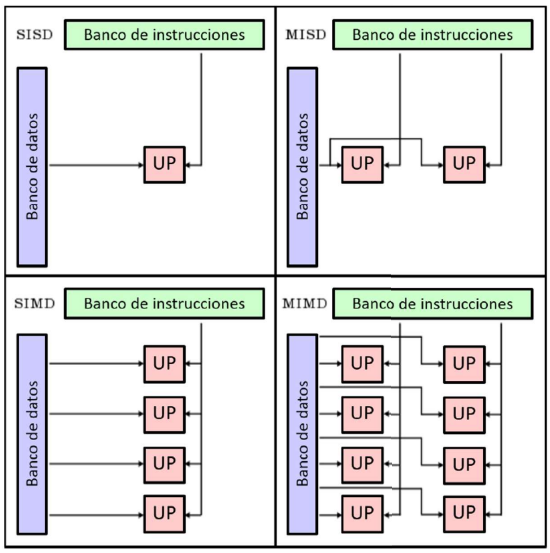


Figura 3.1: Taxonomía de Flynn. Fuente: `http://www.fing.edu.uy/inco/cursos/hpc/material/clases/Clase2-2009.pdf` [26].

En el caso de las tarjetas gráficas, el flujo de instrucciones se ejecuta simultáneamente en cada uno de los núcleos, los cuales realizan el procesamiento de manera secuencial sobre diferentes flujos de datos. De esta manera, gracias a este procesamiento en tandem, es posible manipular flujos de datos de gran tama no que sean susceptibles a dividirse en flujos independientes más peque nos.

3.3. Unidad de Procesamiento Gráfico

La Unidad de Procesamiento Gráfico es el coprocesador encargado del procesamiento de imágenes dentro de cualquier computadora. Este dispositivo surge con la finalidad de aligerar la carga de trabajo de la CPU. La GPU puede considerarse el cerebro

de la tarjeta gráfica. Debido a la fuerte demanda del mercado de procesamiento en tiempo-real y gráficos 3D de alta definición, la GPU se ha convertido en un procesador multi-núcleo de alto paralelismo con una enorme potencia computacional y un ancho de memoria muy alto, como es ilustrado en las 3.2 3.3. La razón detrás de la discrepancia en la capacidad de ejecución de operaciones de punto flotante entre una CPU y una GPU es que la GPU está especializada en computación intensiva de alto paralelismo. En su diseño, más transistores están dedicados al procesamiento de datos en lugar de tareas de almacenamiento en memoria caché o control de flujo.

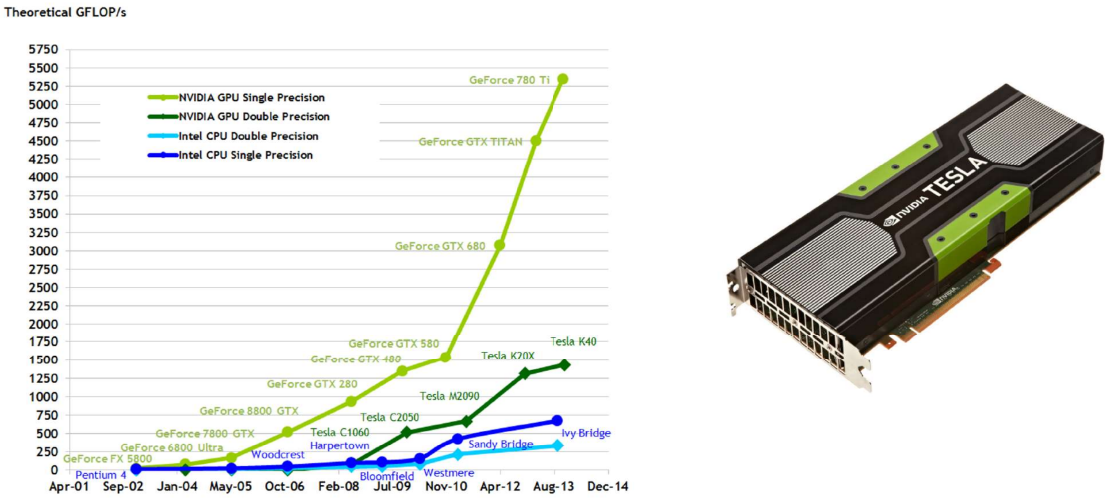


Figura 3.2: Gráfica comparativa de operaciones de punto flotante por segundo en CPU y GPU. A un lado, una tarjeta gráfica NVIDIA Kepler K20. Fuente: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/` [27].

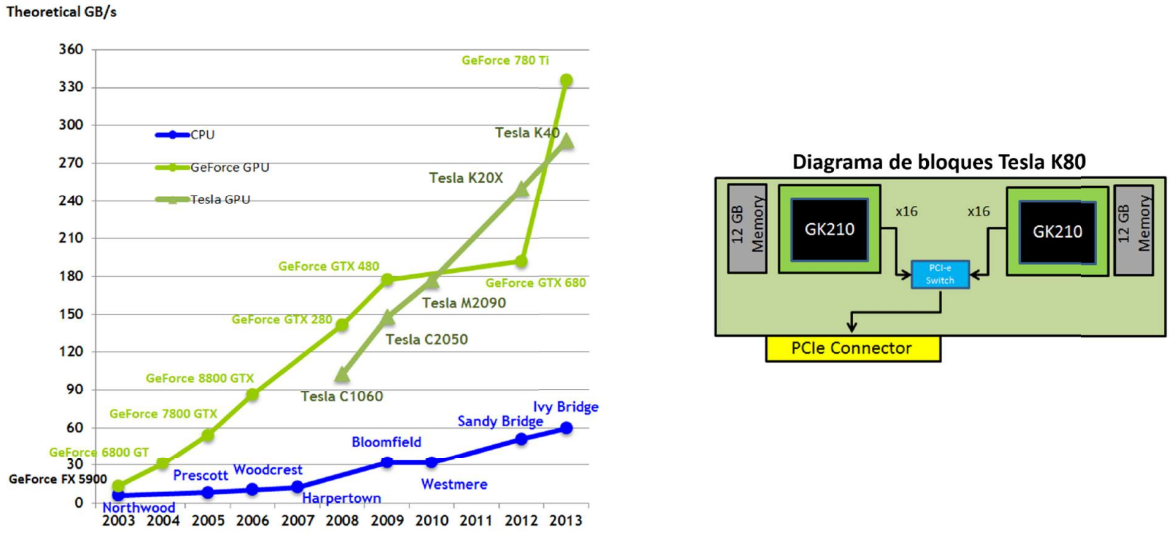


Figura 3.3: Gráfica comparativa de ancho de banda para CPU y GPU. Fuente: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`. A un lado, diagrama de bloques de la tarjeta Tesla K80 que cuenta con dos chips con sus respectivas memorias para el procesamiento de datos, es decir, es dos tarjetas en una. Fuente: `https://www.microway.com/hpc-tech-tips/introducing-nvidia-tesla-k80-gpu-accelerator-kepler-gk210/ \cite{comparison2}.`

La tarjeta Tesla K80 mostrada en la imagen 3.3 tiene un rendimiento de $8,74$ TFLOPS en operaciones de simple precisión. Para lograr estos resultados, esta tarjeta cuenta con dos chips GK210 con 2496 núcleos cada uno, con su espacio de memoria de 12 GB y se comunican a velocidades de $16\times$ con el bus PCIe mediante un interruptor, es decir, es dos tarjetas en una. Específicamente, la GPU es adecuada para abordar problemas que puedan ser expresados como cómputo de datos en paralelo, en otras palabras, el mismo programa ejecutado sobre varios elementos en paralelo con una alta intensidad aritmética (relación de las operaciones aritméticas con respecto a las operaciones en memoria). Debido a que el mismo programa se ejecuta sobre con-

juntos distintos de datos, existe una menor necesidad de control complejo de flujo. La latencia de acceso a memoria puede ocultarse con los cálculos debido a que el programa se ejecuta con alta intensidad aritmética. Así, los grandes cachés de datos resultan innecesarios. El procesamiento con paralelismo sobre datos mapea subconjuntos de datos en hilos de procesamiento en paralelo. Muchas aplicaciones que procesan grandes cantidades de información se basan en éste para acelerar el cómputo. En la generación de imágenes o videos partiendo de un modelo en 3D, grandes conjuntos de pixeles y vértices son mapeados en hilos de procesamiento en paralelo. Similarmente, aplicaciones de procesamiento tanto de imágenes como de multimedia mapean bloques de imágenes y pixeles a hilos de procesamiento en paralelo. Ejemplo de ellas son el post-procesamiento de imágenes renderizadas, la codificación y decodificación de video, el escalamiento de imágenes, la visión estereo, y el reconocimiento de patrones. Incluso, muchos algoritmos fuera del campo de renderización y procesamiento de imágenes son acelerados mediante el procesamiento de datos en paralelo, desde el tratamiento general de se nales o simulaciones físicas hasta finanzas computacionales o biologa computacional [27].

3.4. CUDA

La Arquitectura Unificada de Dispositivos de Cómputo (CUDA) es una plataforma y modelo de programación para cómputo en paralelo de propósito general. Aprovecha la maquinaria de cómputo en paralelo de las GPUs de NVIDIA para resolver problemas computacionales complejos de una forma más eficiente que en una CPU. Está dise nada para operar con varios lenguajes, interfaces de programación de aplicaciones, y estándares de código libre de directivas de compilacion. Por ejemplo: Fortran, DirectCompute, OpenACC [28]. Este entorno de programación fue liberado de forma gratuita por la

compa nía NVIDIA en noviembre de 2006. El flujo de procesamiento en CUDA tiene cuatro pasos principales: 1) Copia de la información de la memoria principal a la memoria de la GPU, 2) La CPU le da órdenes a la GPU de como realizar el procesamiento, 3) La GPU ejecuta las instrucciones en paralelo en cada núcleo y 4) Copia el resultado de la memoria de la GPU a la memoria principal. En la figura 3.4 se muestra gráficamente este proceso.

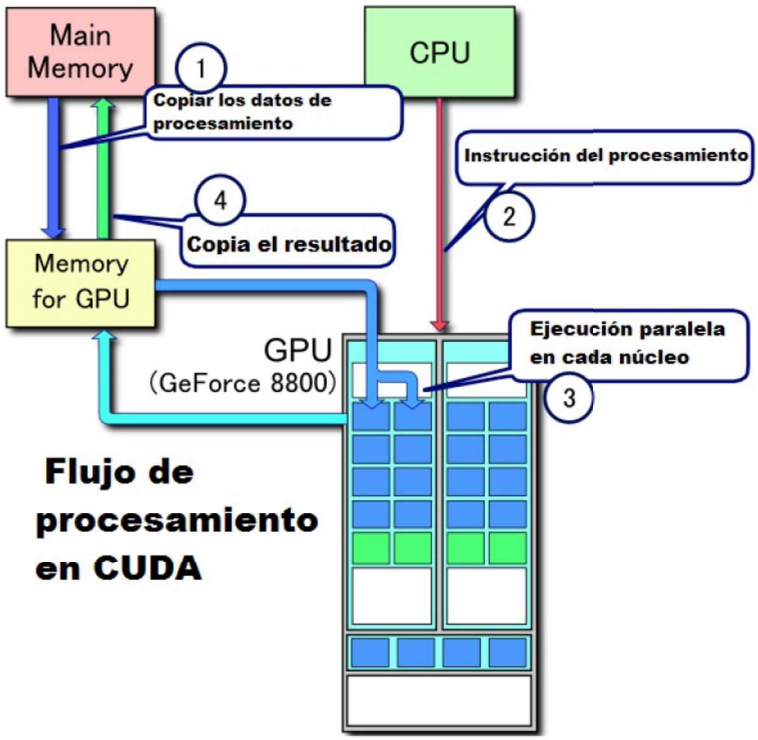


Figura 3.4: Flujo de procesamiento de CUDA. Fuente: `http://sabia.tic.udc.es/gc/trabajos\}202011-12/ATIvsCUDA/ventajas.html` [29].

3.5. Un modelo de programación escalable

La llegada de CPUs multinúcleos y GPUs de varios núcleos significó que los chips del procesador pasaran a ser sistemas paralelos. Actualmente, este paralelismo continúa siendo escalado de acuerdo a la ley de Moore (ver glosario). El reto es desarrollar programas cuyo paralelismo sea escalable de forma clara para aprovechar el creciente número de núcleos de procesamiento, tal y como las aplicaciones gráficas en 3D escalan su paralelismo en diferentes GPUs. El modelo de programación con paralelismo de CUDA está dise nado para superar este reto manteniendo una curva de aprendizaje suave para aquellos programadores que ya están familiarizados con lenguajes de programación estándares como C y FORTRAN. CUDA está estructurado con base en tres elementos fundamentales: Una jerarquía de grupos de hilos, memoria compartida y barreras de sincronización. Los cuales son expuestos al programador como un conjunto mínimo de extensiones de los lenguajes de programación estándar. Estos elementos proveen un paralelismo de grano fino de datos e hilos, anidado dentro de un paralelismo de grano grueso de datos y tareas. Así mismo, sirven de guía al programador para 1) dividir el problema en subproblemas gruesos que pueden ser resueltos simultáneamente en bloques de hilos y 2) dividir cada subproblema en partes más finas que pueden ser resueltas de manera cooperativa y simultánea por todos los hilos del mismo bloque. Esta descomposición permite tanto la incorporación automática de la escalabilidad, como la preservación de la expresibilidad del lenguaje [27]. Efectivamente, cada bloque de hilos puede ser programado en cualquiera de los multiprocesadores disponibles dentro de una GPU, en cualquier orden (concurrente o secuencialmente). Así, un programa compilado en CUDA puede ejecutarse en cualquier número de multiprocesadores, siempre que el entorno de ejecución conozca la cantidad disponible de éstos (Ver Fig. 3.5). Este mode-

lo de programación escalable permite a la arquitectura de la GPU abarcar un amplio rango de mercado simplemente escalando el número de multiprocesadores y particiones de memoria: desde las GPUs Tesla para investigaciones científicas y tecnológicas hasta una variedad de GPUs GeForce de bajo costo.

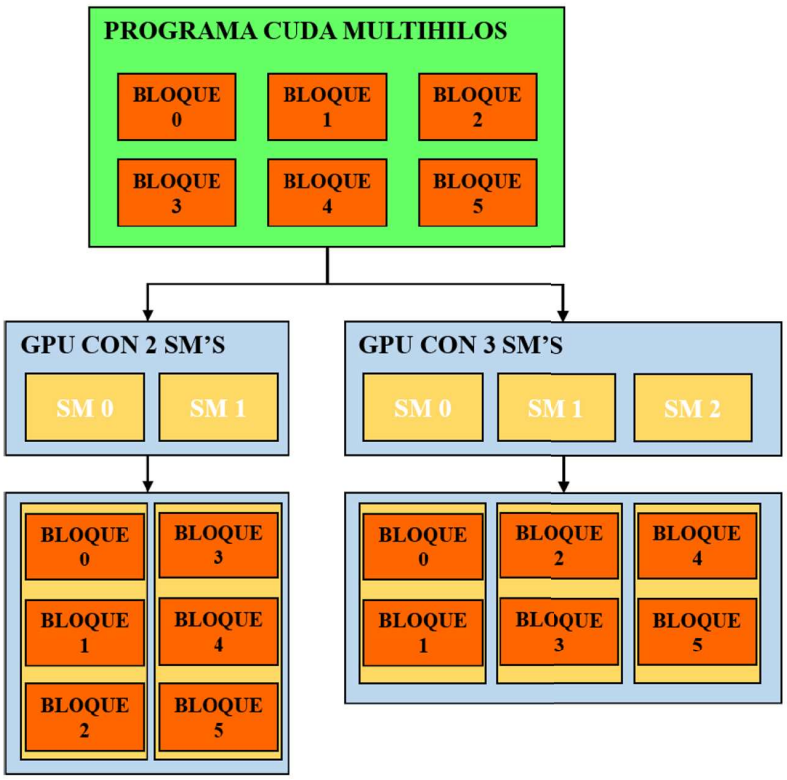


Figura 3.5: Escalabilidad en CUDA. [27].

Una GPU está construída en torno a un arreglo de Streaming Multiprocessors (SMs). Un programa multihilos es particionado en bloques de hilos que se ejecutan de manera independiente, de este modo, una GPU con más multiprocesadores va automáticamente a ejecutar el programa en menos tiempo que una GPU con menos multiprocesadores.

3.6. Modelo de Programación CUDA

La estructura interna de CUDA está basada en jerarquías de unidades de procesamiento que se ejecutan simultáneamente. El procesamiento secuencial, denominado hilo, es colocado como la unidad mínima de la estratificación. Los hilos son creados y destruídos de acuerdo a las definiciones que realiza el programador dentro del código, pero la cantidad máxima depende de la tarjeta gráfica utilizada.

3.6.1. Kernel

CUDA C permite que el programador defina funciones en C, llamadas *kernels*, que son ejecutadas N veces simultáneamente por N diferentes hilos CUDA. Un kernel es definido utilizando la declaración *__global__* y debe especificarse la cantidad de hilos CUDA que ejecutaría el kernel. La sintaxis para invocar el kernel es la siguiente:

$$mi\_kernel <<< N\_bloques, N\_hilos >>>;$$

En la siguiente imagen se muestra un código ejemplo de un kernel y su invocación.

```
// Declaración del kernel
__global__ void SumaVectores(float* A, float* B, float* C)
{
 int i = threadIdx.x;
 C[i] = A[i] + B[i];
}
int main()
{
 ...
 // Invocación del kernel con N hilos
 SumaVectores<<<1, N>>>(A, B, C);
 ...
}
```

Figura 3.6: Ejemplo de llamado a un kernel. [27].

3.6.2. Jerarquía de hilos

Cada hilo que ejecuta el kernel recibe un índice que permite acceder directamente a él. Este índice es asignado mediante la variable *threadIdx*. Esta variable está definida como un vector de tres componentes, es decir, los hilos pueden ser identificados con índices de una, dos o tres dimensiones, organizados, a su vez, en bloques de una, dos o tres dimensiones. Esto proporciona una paridad natural entre los elementos de procesamiento y los elementos de cálculo como vectores, matrices o volúmenes. La cantidad de hilos por bloque que es posible crear es limitado debido a que todos los hilos de un bloque residen en un mismo procesador y deben compartir la memoria de ese procesador. En las GPUs actuales la cantidad máxima de hilos por bloque es de 1024 hilos. Sin embargo, un kernel puede ejecutarse en varios bloques con la misma cantidad de hilos, así, el número de hilos creados es igual a la cantidad de hilos por bloques multiplicado por el número de bloques. En la figura 3.7 se muestra la declaración e invocación de un kernel con bloques de dos dimensiones.

```
// Declaración del kernel
__global__ void SumaMatrices(float A[N][N], float B[N][N], float
C[N][N])
{
 int i = threadIdx.x;
 int j = threadIdx.y;
 C[i][j] = A[i][j] + B[i][j];
}
int main()
{
 ...
 // Invocación del kernel con un bloque de N*N*1 hilos
 int numBlocks = 1;
 dim3 threadsPerBlock(N, N);
 SumaMatrices<<<numBlocks, threadsPerBlock>>>(A, B, C);
 ...
}
```

Figura 3.7: Ejemplo de llamado a un kernel con bloques de dos dimensiones. [27].

Los bloques de hilos están organizados en mallas de una, dos o tres dimensiones.

El número de bloques en una malla es definido por la información a procesar o por el número de procesadores en el sistema. Cada bloque dentro de una malla, puede ser identificado mediante la variable *blockId*, el cual asigna un índice de una, dos o tres dimensiones. En este caso, la identificación de cada hilo debe hacerse de la manera ilustrada en la figura 3.8.

```
// Declaración del kernel
__global__ void Suma Matrices(float A[N][N], float B[N][N],
float C[N][N])
{
 int i = blockIdx.x * blockDim.x + threadIdx.x;
 int j = blockIdx.y * blockDim.y + threadIdx.y;
 if (i < N && j < N)
 C[i][j] = A[i][j] + B[i][j];
}
int main()
{
 ...
 // Invocación del kernel
 dim3 threadsPerBlock(16, 16);
 dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
 Suma Matrices<<<numBlocks, threadsPerBlock>>>(A, B, C);
 ...
}
```

Figura 3.8: Ejemplo de un llamado a un kernel identificando cada hilo dentro de la malla.

De manera gráfica, en la figura 3.9 se muestra esta jerarquía CUDA.

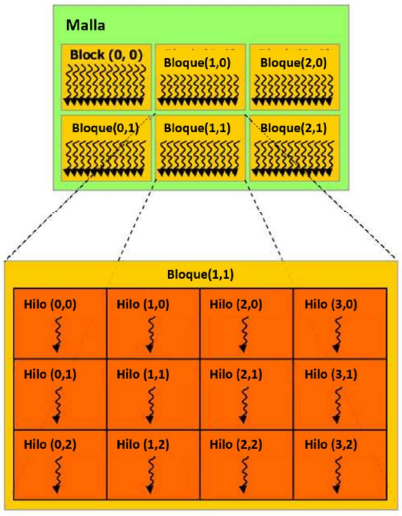


Figura 3.9: Jerarquía de elementos de procesamiento CUDA. Fuente: `https://riunet.upv.es/bitstream/handle/10251/11735/INCO2-2011-01.pdf`

Los hilos dentro de cada bloque pueden cooperar entre sí compartiendo datos a través de un espacio de memoria compartida y sincronizando su ejecución para coordinar los accesos a esta memoria. Esta sincronización puede ser especificada dentro del código mediante la función $\_\_syncthreads()$, la cual actúa como una barrera donde todos los hilos tienen que esperar hasta que todos hayan completado las tareas asignadas.

3.6.3. Jerarquía de memoria

CUDA otorga a cada hilo la posibilidad de acceder a diferentes espacios de memoria en cada una de sus ejecuciones, estos espacios de memoria están organizados de la siguiente manera: Cada hilo posee una memoria local privada. Cada bloque tiene un espacio de memoria compartida, accesible para todos los hilos del bloque durante la misma ejecución. Adicionalmente existe un espacio de memoria global al que pueden acceder todos los hilos de todos los bloques creados en una misma ejecución. Existen además, dos memorias derivadas del procesamento de pixeles y vértices: la memoria

constante y la memoria de texturas. Gráficamente se puede observar esta jerarquía en la imagen 3.10.

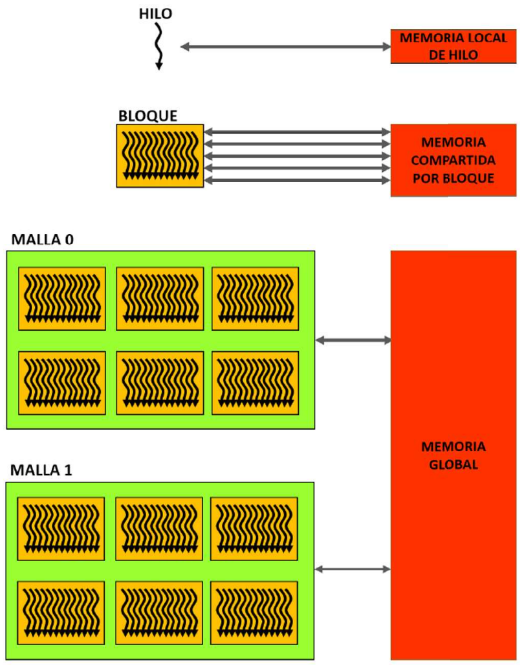


Figura 3.10: Jerarquía de espacios de memoria CUDA. [15]

El modelo de programación de CUDA asume que los hilos son ejecutados en una tarjeta gráfica que está físicamente separada y opera como un coprocesador diferente al procesador CPU que va a ejecutar el programa. También asume que la memoria que utiliza cada uno está separada en espacios de memoria propias. Así, cuando el kernel es ejecutado en la GPU, el resto del programa es ejecutado en el CPU. En la imagen 3.11 se muestra un esquema de este proceso.

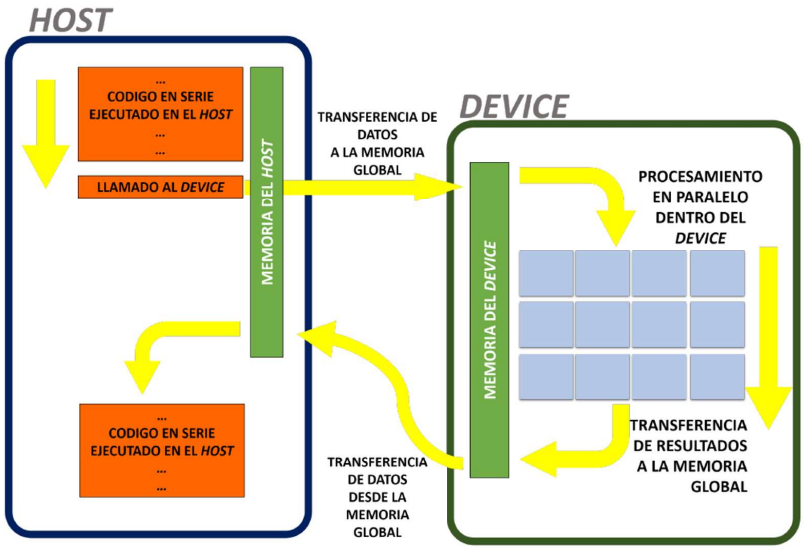


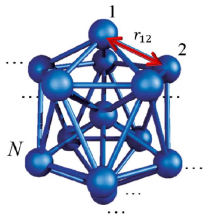
Figura 3.11: Ejecución de un programa en CUDA.

3.7. Modelo para la Interacción Atómica

El modelo seleccionado para describir las interacciones entre los átomos del cúmulo es el potencial Lennard-Jones. Este potencial modela la interacción tipo Van Der Waals entre dos átomos de gases nobles para una configuración electrónica dada. De acuerdo a esta expresión, la energía potencial de un par de átomos debido a las fuerzas interatómicas es una función de la distancia entre los centros de dichos átomos. Cada par de átomos está sujeto a dos fuerzas distintas: una fuerza atractiva o *fuerza de Van Der Waals* y una fuerza de repulsión o *repulsión de Paulli*, las cuales varían de acuerdo a la separación entre dichos átomos. La expresión del potencial de Lennard-Jones se describe en la ecuación 3.1.

$$U(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{3.1}$$

Donde ϵ representa la intensidad de la interacción y σ la escala de longitudes, elementos que varían de acuerdo a cada elemento. Para un $r > \sigma$, la fuerza es atractiva y para $r < \sigma$ la fuerza es altamente repulsiva; si $r = \sigma$ la fuerza es nula. Considerando la expresión anterior, la energía de interacción descrita por el modelo de Lennard-Jones para un cúmulo de N átomos es la siguiente:



$$U(\vec{r_i}, \ldots \vec{r_N}) = \frac{1}{2}\sum_{i=1}^{N}\sum_{\substack{j=1 \\ j\neq i}}^{N} 4\epsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right]$$

Figura 3.12: Modelo Lennard-Jones para un cúmulo de N átomos.

La implementación del modelo de Lennard-Jones para un cúmulo de N átomos presenta la particularidad de que la interacción entre cada par de átomos es independiente de los demás, es decir, cada par de átomos está aislado del resto de pares. Así mismo, la energía de interacción de los N átomos es la suma de las energías de interacción de los pares independientes de átomos.

A partir de esta ecuación se puede realizar la determinación de la fuerza a la que están sujetos dichos átomos. La fuerza sobre un átomo dado es menos el gradiente de la energía de interacción respecto de la posición del átomo dado.

$$\vec{F_i} = -\vec{\nabla}_i U\left(\vec{r_1}, ..., \vec{r_N}\right) \tag{3.2}$$

Donde la función $U\left(\vec{r_1}, ..., \vec{r_N}\right)$ describe las interacciones atómicas.

3.8. Cálculo del Movimiento Atómico

La simulación de la evolución de un cúmulo durante un periodo de tiempo determinado permite determinar las características de este cúmulo a lo largo de instantes representativos. Para ello, es necesario resolver numéricamente las ecuaciones de movimiento de Newton. Para fines de esta investigación se utiliza el Algoritmo de Verlet simple. Dada la configuración de un átomo r en un elemento t de una partición temporal T, el algoritmo de Verlet determina la configuración de dicho átomo en el elemento siguiente de la partición temporal $t + \Delta t$ usando la configuración en dos instantes consecutivos anteriores $\vec{r}\,(t - \Delta t)$ y $\vec{r}\,(t)$. Donde Δt representa el paso de integración entre dos instantes de la partición temporal T. El cálculo de $\vec{r}\,(t + \Delta t)$ y $\vec{r}\,(t - \Delta t)$ se expresa como una serie numérica (ecuación 3.3 y 3.4).

$$\vec{r_i}\,(t + \Delta t) = \vec{r_i}\,(t) + \dot{\vec{r_i}}\,(t)\,\Delta t + \frac{1}{2!}\ddot{\vec{r_i}}\,(t)\,\Delta t^2 + \frac{1}{3!}\dddot{\vec{r_i}}\,(t)\,\Delta t^3 + ... \tag{3.3}$$

$$\vec{r_i}\,(t - \Delta t) = \vec{r_i}\,(t) - \dot{\vec{r_i}}\,(t)\,\Delta t + \frac{1}{2!}\ddot{\vec{r_i}}\,(t)\,\Delta t^2 - \frac{1}{3!}\dddot{\vec{r_i}}\,(t)\,\Delta t^3 + ... \tag{3.4}$$

De restar y sumar cada uno de los términos de las dos ecuaciones se obtiene:

$$\vec{r_i}\,(t + \Delta t) = 2\vec{r_i}\,(t) - \vec{r_i}\,(t - \Delta t) + \ddot{\vec{r_i}}\,(t)\,\Delta t^2 + O\left(\Delta t^4\right) \tag{3.5}$$

$$\vec{v_i} = \frac{1}{2\Delta t}\left\{\vec{r_i}\,(t + \Delta t) - \vec{r_i}(t - \Delta t) + O(\Delta t^3)\right\} \tag{3.6}$$

Además de determinar la configuración en el instante $t + \Delta t$, el Algoritmo de Verlet también determina las velocidades en el instante denominado t, teniendo así, el estado dinámico del cúmulo en cada elemento de la partición temporal. El número total de instantes de la partición es denominado $kmax$. Este par de ecuaciones finales (3.5 y 3.6) utilizan la aceleracion de los átomos del cúmulo para el cálculo de la configuración

en el instante siguiente. La aceleración se obtiene a partir de la segunda ley de newton:

$$\ddot{\vec{r}}_i = \frac{1}{m}\vec{F}_i\,(t) \tag{3.7}$$

La descripción del movimiento interatómico de la nanopartícula está totalmente definida por estas expresiones matemáticas a lo largo de todo el espacio temporal determinado (ver capítulo 5). Así, el Algortimo de Verlet realiza una aproximación del estado dinámico del cúmulo en el instante siguiente del cúmulo. Para esto, considera que el movimiento de cada átomo en cada subintervalo de la partición es uniformemente acelerado. Aunque la aceleración puede ser distinta para cada átomo y para cada subintervalo. En la figura 3.13 se representa gráficamente la implementación del Algoritmo de Verlet.

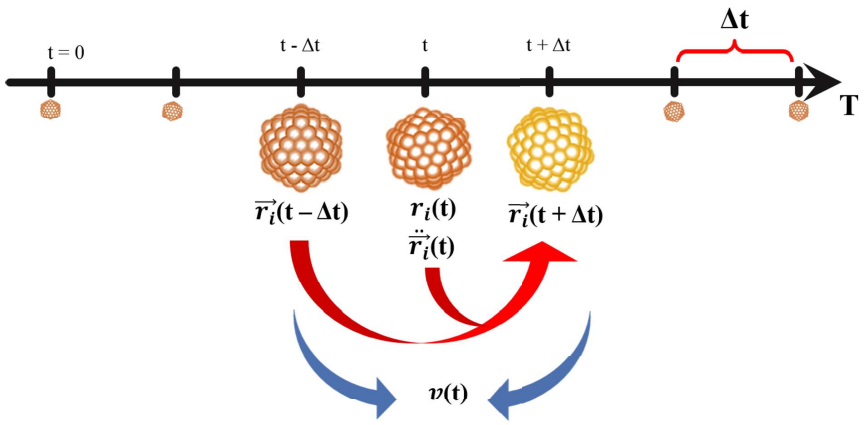


Figura 3.13: Representación gráfica del Algoritmo de Verlet.

Capítulo 4

Metodología

A continuación se muestra el procedimiento a seguir para la codificación del algoritmo del programa de dinámica molecular, donde el cálculo las fuerzas interatómicas se realiza mediante el paradigma de programación de CUDA:

1. **Identificación de las partes independientes del cálculo de las fuerzas**. Las fuerzas entre los átomos del cúmulo son los elementos principales a determinar en cualquier programa de dinámica molecular. La descripción de las interacciones atómicas a partir de las cuales se calculan las fuerzas son tareas independientes unas de otras. Para identificar estas partes independientes se precisan los siguientes elementos:

   a) **Determinación de la expresión de la fuerza**. A partir de un potencial de interacción se determinaría la expresión de las fuerzas interatómicas que sería implementada en la GPU.

   b) **Descripción del cálculo de las fuerzas atómicas**. La estructura de la expresión analítica de la fuerza atómica expresa la naturaleza de los cálculos a realizar para la determinación de las mismas, las variables de entrada requeridas, los cálculos intermedios necesarios para obtener las fuerzas inter-

atómicas, la independencia de estos cálculos, entre otros elementos.

c) **Identificación de los subcálculos independientes**. La descripción previa de la expresión de la fuerza permite establecer los sub-cálculos que pueden ser tratados de manera independiente. Si un grupo de cálculos depende de las posiciones atómicas iniciales, pueden considerarse de resolución simultánea.

2. **Construcción del algoritmo**. El análisis completo del cálculo de las fuerzas permite generar el algoritmo adecuado, haciendo una clara distinción de los procesos seriales y paralelos.

3. **Identificación de los elementos de CUDA-C adecuados para realizar los subcálculos**. El paradigma de programación de CUDA posee una jerarquía de procesamiento y de memoria que eficienta los cálculos dentro de las tarjetas gráficas. Los subcálculos independientes serán asignados a las unidades mínimas de procesamiento de CUDA (hilos). Aquellos hilos que realicen tareas con características comunes serán organizados en bloques de procesamiento.

4. **Codificación del algoritmo**. A partir del algoritmo se construye el programa en lenguaje CUDA-C. Los elementos antes alcanzados permiten la codificación de un programa que establezca una correlación adecuada del proceso de determinación de las fuerzas interatómicas y los elementos de procesamiento y jerarquía de espacios de memoria CUDA.

5. **Evaluación del código**. Se llevan a cabo pruebas con cúmulos de diferente número de átomos para comprobar la velocidad y precisión de los resultados.

En la figura 4.1 se esquematiza el procedimiento antes descrito.

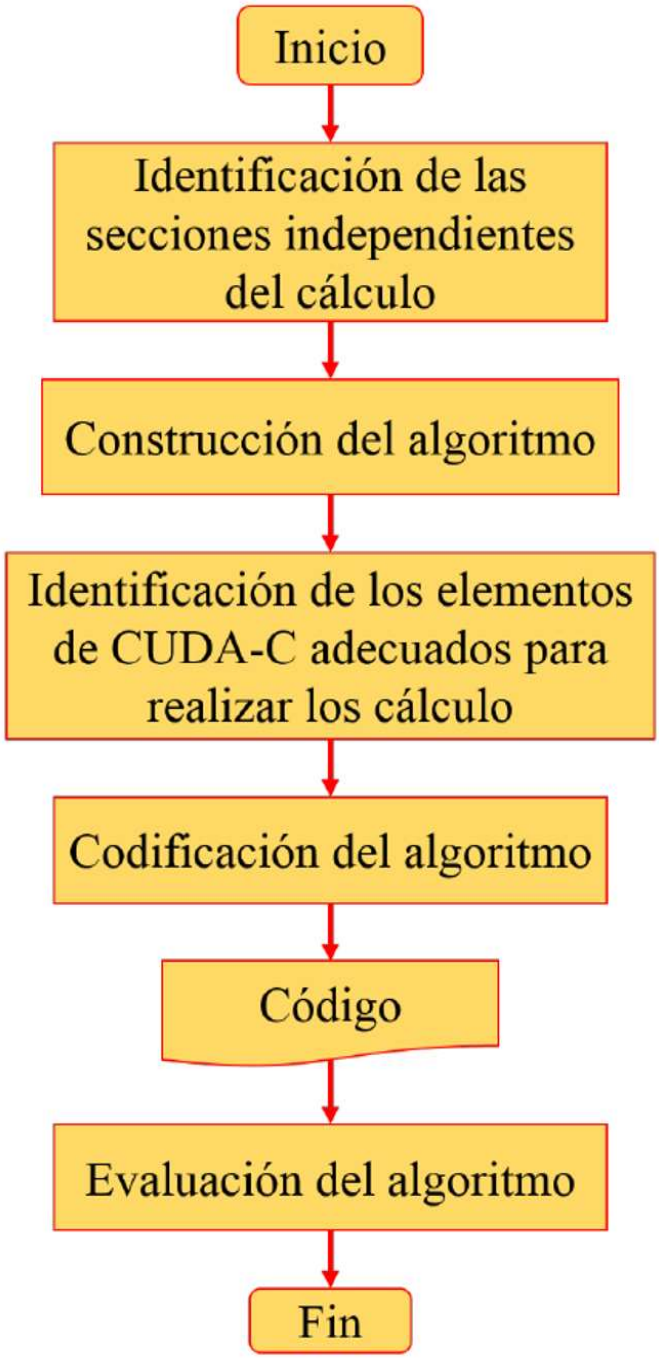


Figura 4.1: Diagrama de flujo de la metodología utilizada.

4.1. Expresión de la fuerza

Las interacciones atómicas están descritas por su función energía potencial $U(\vec{r_1}, ..., \vec{r_N})$, de forma que la fuerza ejercida sobre el i-ésimo átomo por el resto de los átomos del cúmulo está dada por:

$$\vec{F_i} = -\vec{\nabla}_i U\left(\vec{r_1}, ..., \vec{r_N}\right) \tag{4.1}$$

Si la interacción atómica está determinada únicamente por el conjunto de todas las distancias interatómicas

$$r_{ij} = |\vec{r_i} - \vec{r_j}| \tag{4.2}$$

$$i, j = 1, ..., N;$$

la estructura funcional de U consiste en la composición de funciones:

$$(x_i, y_i, z_i, ..., x_N, y_N, z_N) \mapsto r_{ij} \mapsto U \tag{4.3}$$

Es decir,

$$U\left(x_1, y_1, z_1, ..., x_N, y_N, z_N\right) = \left(r_{ij} \circ U\right)\left(x_1, y_1, z_1, ..., x_N, y_N, z_N\right) \tag{4.4}$$

El conjunto de variables independientes r_{ij} de la función $U(r_{ij})$ está representado por el esquema matricial de la figura 4.2

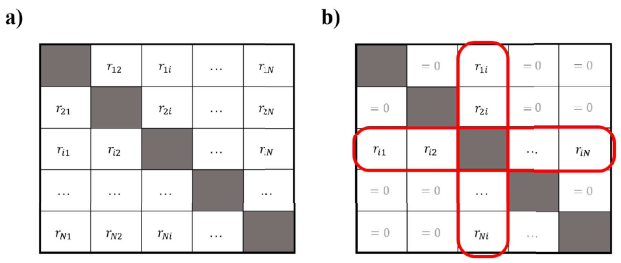


Figura 4.2: Pares ordenados de átomos. a)Matriz completa b)Selección del átomo P_i.

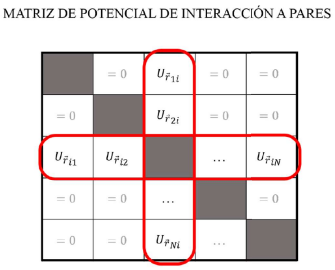


Figura 4.3: Potencial de interacción para el átomo i.

El gradiente de la función compuesta $U\left(x_1, y_1, z_1, ..., x_N, y_N, z_N\right)$ respecto de las coordenas del átomo i-ésimo es determinado mediante la regla de la cadena del cálculo vectorial:

$$\frac{\partial}{\partial x_i} U\left(x_1, y_1, z_1, ..., x_N, y_N, z_N\right) = \sum_m \left(\frac{\partial}{\partial r_{im}} U + \frac{\partial}{\partial r_{mi}} U\right)\left(x_i - x_m\right) \tag{4.5}$$
$$m = 1, 2, ..., N;$$

Las interacciones existentes entre los átomos de un cúmulo compuesto por N átomos de un gas noble se modelan por la función energía potencial:

$$U\left(\vec{r_1}, ..., \vec{r_N}\right) = \frac{1}{2} \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j \neq i}}^{N} 4\epsilon \left[\left(\frac{\sigma}{r_{ij}}^{12}\right) - \left(\frac{\sigma}{r_{ij}}^{6}\right)\right] \tag{4.6}$$

Por lo tanto:

$$\vec{\nabla} U\left(\vec{r_1}, ..., \vec{r_N}\right) = -\sum_{i \neq j}^{N} \frac{24\epsilon}{\sigma^2} \left[2\left(\frac{\sigma}{r_{ij}}\right)^{14} - \left(\frac{\sigma}{r_{ij}}\right)^{8}\right] \vec{r_{ij}} \tag{4.7}$$

Considerando la ecuación 4.1 y la ecuación anterior, es posible determinar la expresión de la fuerza ejercida sobre un átomo i con respecto de los demás átomos del cúmulo. Así mismo, derivar el cálculo de la fuerza del modelo de interacción Lennard-Jones presenta la característica de que fuerza ejercida sobre un átomo dado es la suma de las fuerzas que de forma independientemente ejercen cada uno de los otros átomos ejercen sobre el átomo en cuestión.

$$\vec{F_i} = \sum_{i \neq j}^{N} \frac{24\epsilon}{\sigma^2} \left[2\left(\frac{\sigma}{r_{ij}}\right)^{14} - \left(\frac{\sigma}{r_{ij}}\right)^{8}\right] \vec{r_{ij}} \tag{4.8}$$

4.2. Cálculo de las fuerzas atómicas: Algoritmo

La principal característica del algoritmo por construir es el paralelismo. Este elemento se deriva directamente de la fragmentación del cálculo completo en partes independientes. Cambiaremos ligeramente la expresión de las fuerzas con el fin identificar los fragmentos independientes del cálculo. Así:

$$\vec{F_i} = \sum_{\substack{i=1 \\ i \neq j}}^{N} \vec{F_{ij}} \tag{4.9}$$

$$i = 1, 2, ..., N;$$

donde

$$\vec{F_{ij}} = \frac{24\epsilon}{\sigma^2} \left[2 \left(\frac{\sigma}{r_{ij}} \right)^{14} - \left(\frac{\sigma}{r_{ij}} \right)^{8} \right] \vec{r_{ij}} \tag{4.10}$$

Esta expresión evidencia 2 propiedades fundamentales de las fuerzas atómicas:

- La independencia de las fuerzas ejercidas sobre cada átomo. La fuerza sobre un átomo dado está determinada únicamente por la configuración del cúmulo, no está determinada por la fuerza ejercida sobre algún otro átomo.

- La dependencia de la fuerza ejercida sobre un átomo dado, respecto de las fuerzas que ejercen de forma independiente cada uno del resto de los átomos del cúmulo sobre él.

La independencia de las fuerzas atómicas revela que los N cálculos de las N fuerzas atómicas son fragmentos independientes del cálculo completo. Por otra parte, la independencia de las fuerzas que sobre un átomo dado ejercen cada uno de los átomos del resto del cúmulo revela que los cálculos de estas $N - 1$ fuerzas a pares también son fragmentos independientes del cálculo completo. Por lo tanto, el cálculo como un todo

es separable a lo más en $N*(N-1)$ segmentos independientes. La ejecución simultánea de estos subcálculos requiere de $N*(N-1)$ núcleos de procesamiento. Sin embargo, este número supera por mucho al número de núcleos contenidos en una GPU. Para lograr la simultaneidad del cálculo completo, cada núcleo deberá ejecutar un grupo de subcálculos en vez de limitarse a ejecutar solo uno de ellos. Así, el cálculo de cada fuerza atómica será realizado por g núcleos. Los subcálculos serían distribuidos uniformemente entre ellos con el fin de optimizar el paralelismo de su ejecución: L núcleos ejecutarían $M+1$, y los $g-L$ restantes ejecutarían M, ver figura 5.1. Donde M y L, son el cociente y el residuo de la división de $N-1$ entre g. En resumen el cálculo completo de las N fuerzas atómicas es fragmentado en $N*g$ partes independientes, donde cada fuerza es calculada mediante g procesos independientes. Para un cúmulo compuesto de mil átomos, el número de núcleos requeridos sería del orden de un millón. La GPU Kepler K20 es una de las más avanzadas en este momento, y posee sólo 2496 núcleos. Agrupando en sumatorias diferentes los términos que serían calculados por núcleos distintos, la fuerza se expresa así:

$$\vec{F_i} = \sum_{\substack{j=0 \\ j\neq i}}^{M+1} F_{i,j*g} + \sum_{\substack{j=0 \\ j\neq i}}^{M+1} F_{i,j*g+1} + \sum_{\substack{j=0 \\ j\neq i}}^{M} F_{i,j*g+(g-2)} + ... + \sum_{\substack{j=0 \\ j\neq i}}^{M} F_{i,j*g+(g-1)} \tag{4.11}$$

Con base en esta ecuación, es posible describir la carga de trabajo que tendrá cada núcleo. A partir de esta descripción se genera el algoritmo del procesamiento dentro de la tarjeta gráfica, el cual posteriormente será codificado en CUDA. Ver esquema 5.1 y esquema 5.2. El proceso es el siguiente:

- Se designa una cantidad g de núcleos que determinarán la fuerza ejercida sobre cada uno de los átomos del cúmulo.

- La determinación de la fuerza se deriva de la energía de interacción de pares ordenados de cada uno de los átomos con el resto. El conjunto de pares son

divididos en g fragmentos.

- Estos fragmentos son distribuídos uniformente entre todos los núcleos. Cada uno de ellos, determina un parte de la fuerza ejercida sobre el átomo en cuestión de forma paralela.

- Gracias a la jerarquía interna de CUDA, dentro de cada núcleo se realiza una paralelización para que la mayor cantidad de pares ordenados sean procesados de manera simultánea.

- Al terminar estos procesamientos, cada núcleo posee una sección de la fuerza total ejercida sobre cada átomo. Y éstas son acumuladas para obtener la fuerza total sobre cada átomo.

4.3. Subprograma

Se presenta un algoritmo y su codificación en CUDA-C para la ejecución en paralelo de las fuerzas ejercidas sobre los N átomos de un cúmulo de gases nobles. El código se construye identificando con cada una de las unidades fundamentales de procesamiento (hilos CUDA), un subconjunto de los subcálculos asociados a la fuerza ejercida sobre un átomo dado. El algoritmo consiste en 1) separar el cúmulo en 128 fragmentos y 2) calcular las N fuerzas por medio de la ejecución simultánea de $N * 128$ hilos de procesamiento, distribuídos en N bloques. Cada bloque está dedicado exclusivamente a la determinación de la fuerza total de un átomo, mediante el cálculo y la acumulación de las fuerzas aplicadas por cada átomo del fragmento correspondiente.La fuerza resultante sobre este átomo es determinada por la reducción de las fuerzas aplicadas por los 128 fragmentos. En CUDA, las instrucciones que se ejecutan en las tarjetas gráficas se

escriben como funciones en C, constituyen el *kernel* y se indica iniciando con la expresión *__global__*. Dentro de estos *kernels* se realiza la distribución de los pares ordenados creados a todos los bloques generados en el llamado al *device*; dentro de los bloques se etiquetan los hilos y se les asignan una cantidad específica de pares a procesar, después guardan los resultados en arreglos que tiene un tama no igual a la variable *grupo* (fig 5.3), que es 128 en este caso.

Cada vez que se realiza el cálculo del estado dinámico del cúmulo, debe realizarse un llamado a la GPU. Antes de esto es necesario indicar qué elementos son los que se transferirán al *device*. Específicamente, la configuración del cúmulo en sus coordenadas x, y, z. Al realizar el llamado se crean la cantidad de hilos y bloques que se indica y se transfieren los datos a la memoria global del *device*; los bloques generados copian los arreglos de configuración a la memoria compartida de cada bloque y cada hilo se encarga de obtener las fuerzas totales para cada elemento del cúmulo; el cálculo del potencial de interacción se realiza en conjunto con la determinación de las fuerzas. Una vez realizado ambos procesos, la fuerza y la energía de interacción de cada átomo del cúmulo son enviados a la memoria global del *device* y posteriormente al host para proseguir con el cálculo del estado dinámico y propiedades del cúmulo. La codificación de este proceso es la parte medular del código, ya que la velocidad de procesamiento y la precisión de los resultados depende de una correcta organización de las tareas a realizar dentro de la tarjeta gráfica.

4.4. Evaluación del código

En este último apartado evaluamos la eficacia del código mediante la determinación de la energía total del cúmulo a lo largo y al finnal de toda la evolución. Esta energía

debe conservarse debido a que este cúmulo está aislado, es decir, no interactúa con el medio. Esto se realizó calculando las fuerzas atómicas sobre un cúmulo en equilibrio mecánico. Para dar mayor certidumbre, se eligió una configuración de experimentos previos reportada en http://doye.chem.ox.ac.uk/jon/structures/LJ.html por Jonathan Doye. El programa deberá reproducir la nulidad de cada una de estas fuerzas, así como la conservación de la energía. La ejecución de este programa se realiza mediante el uso de la terminal de cualquier distribución Unix. Primero se compila bajo la instrucción nvcc, que reconoce la extensión .cu. Posteriormente, se ejecuta el programa. Ambos procesos se realizan como se muestra en la figura siguiente:

```
usuario@tzolkin: ~$ nvcc INTERACTUA.cu
usuario@tzolkin: ~$ ./INTERACTUA
```

Figura 4.4: Ejecución del programa.

Cabe se nalar que los archivos de entrada deben estar en la misma ubicación que el código a ejecutar. Al terminar el proceso se realiza una valoración sobre la velocidad de procesamiento y la precisión de los resultados.

Capítulo 5

Resultados

Este capítulo consiste en la descripción de cada uno de los resultados de cada elemento de esta tesis. Estos resultados constituyen por sí mismos los elementos que conforman el código de dinámica molecular, el cual es uno de los resultados de este trabajo de investigación. A lo largo del capítulo, se describirá cada uno de estos elementos del trabajo de investigación. La determinación de las fuerzas atómicas es el componente fundamental de todo programa de dinámica molecular, constituye casi la totalidad de los cálculos realizados. Estos programas, tanto su algoritmo como su codificación, actualmente están documentados en detalle, por ejemplo [16]. Por el contrario, el cálculo de las fuerzas no está documentado, menos aún su cálculo en paralelo. Así, este capítulo se enfoca únicamente en la construcción del algoritmo para calcular las fuerzas atómicas y en la codificación del mismo para su ejecución en paralelo utilizando una GPU. Este enfoque facilita en extremo la extrapolación de los elementos aquí presentados a la codificación de cualquier otro programa en CUDA-C. Sin importar la naturaleza tanto de los sistemas como de las propiedades que calcule, por ejemplo el pronóstico del tiempo [24]. Sin embargo, en el apéndice A se describe el programa de dinámica molecular codificado en esta investigación. Éste, por sí mismo, constituye un resultado adicional de la tesis. Se inicia con la determinación de la expresión de las fuerzas atómicas. Mediante

el potencial de Lennard-Jones se modela la interacción existente entre los átomos del cúmulo, del gradiente de este modelo se construye la expresión de la fuerza. Enseguida se aborda la construcción del algoritmo para calcular estas fuerzas. Ésto se realiza con base en dos elementos fundamentales, la identificación de las partes independientes del cálculo y la distribución uniforme de estas partes entre los núcleos de procesamiento de la tarjeta gráficca. La independencia de los cálculos determina la carga de trabajo que cada núcleo debe realizar simultáneamente, es decir, el nivel de paralelización, y la distribución uniforme equilibra el trabajo de cada núcleo. De este modo se llega al resultado central del capítulo, la descripción de la codificación en lenguaje CUDA-C del algoritmo construído. La paralelización dentro de las tarjetas gráficas es definida por la organización de las unidades fundamentales de procesamiento, hilos CUDA, y su jerarquía. El código se construye asociando un subconjunto de los cálculos independientes a una de estas unidades. Se debe tomar en cuenta la organización de los hilos dentro de la tarjeta gráfica para crear un código eficiente, así, es importante que las distribución de los subconjuntos de subcálculos consideren esta jerarquía. El código obtenido se describe con extremo detalle, esquematizando el cálculo tanto en términos de los elementos fundamentales de procesamiento dentro de la GPU (hilos) y su organización jerárquica (bloques y mallas), como en términos de la estructura jerárquica de la memoria utilizada (local, compartida y global). Se inserta el código obtenido en CUDA-C en un programa en lenguaje C para calcular las fuerzas atómicas en un cúmulo, con el fin de describir los comandos de comunicación entre el anfitrión (CPU) y el huésped (GPU). Por último, se describe un ejemplo de ejecución del código del sistema: anfitrión + húesped.

5.1. Expresión de la fuerza

De acuerdo con las ecuaciones 4.1 y 4.7 la expresión que determina la fuerza aplicada sobre el átomo i es la siguiente:

$$\vec{F_i} = \sum_{i \neq j}^{N} \frac{24\epsilon}{\sigma^2} \left[2 \left(\frac{\sigma}{r_{ij}} \right)^{14} - \left(\frac{\sigma}{r_{ij}} \right)^{8} \right] \vec{r_{ij}} \tag{5.1}$$

Donde N es el número de átomos del cúmulo y r representa la separación entre los centros del átomo i y j. Notar que la fuerza que sobre un átomo es ejercida por el resto de los átomos del cúmulo es la suma de las fuerzas que cada uno esos átomos ejercen de forma independiente sobre el átomo en cuestión. De este modo, la resolución de cada una de estas fuerzas puede realizarse simultáneamente.

5.2. Cálculo de las fuerzas atómicas: Algoritmo

La ecuación 4.11 (véase capítulo 4) expresa la naturaleza de las operaciones que se llevan a cabo para la obtención de las fuerzas aplicadas sobre el átomo i, la cantidad de términos que contiene esta expresión es igual a la cantidad de hilos generados, en este caso es 128 y está denotado por la variable g, (dentro del código puede leerse como *grupo*); la determinación de la fuerza está basada en sumatorias independientes de las fuerzas entre el átomo i y los demás elementos del cúmulo; así, cada hilo realiza simuláneamente el cálculo para cada uno de los términos de la ecuación 4.11 obteniendo fuerzas parciales, es decir, simultáneamente se realizan 128 cálculos. Durante estos procesos se realiza una acumulación por hilo de los resultados parciales provenientes de los pares ordenados que a cada hilo le correspondió calcular, obteniendo fuerzas parciales por hilo que, sucesivamente, se acumulan con los resultados de los demás hilos

del bloque, hasta tener la fuerza total que actúa sobre el átomo i y lo mismo ocurre de manera paralela en los demás bloques. El diagrama 5.1, presentado a continuación muestra el procesamiento que se lleva a cabo dentro de cada bloque, usando las ecuaciones explicadas anteriormente.

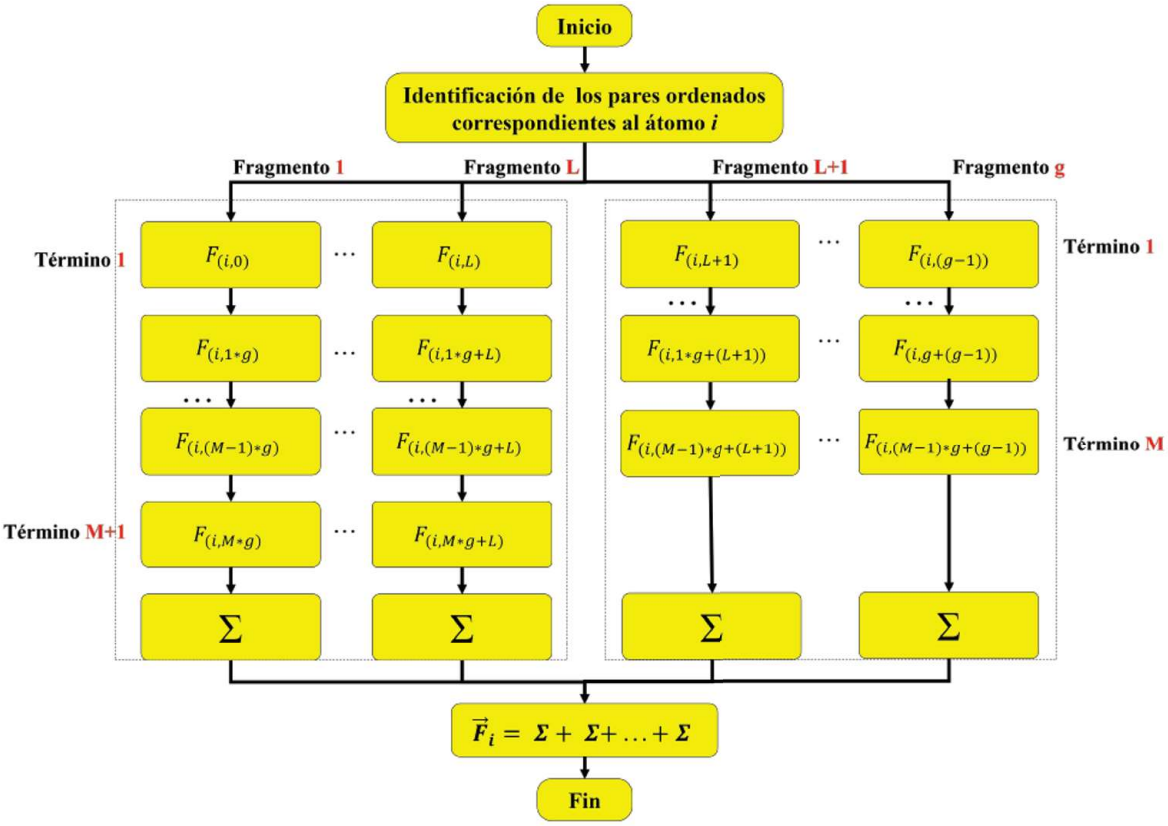


Figura 5.1: Procesamiento dentro de cada bloque.

Así, es posible elaborar un esquema general acerca del procesamiento que se lleva a cabo dentro del device en cada iteración para el cálculo de las fuerzas. Cada sección de este esquema representa el trabajo realizado por cada bloque.

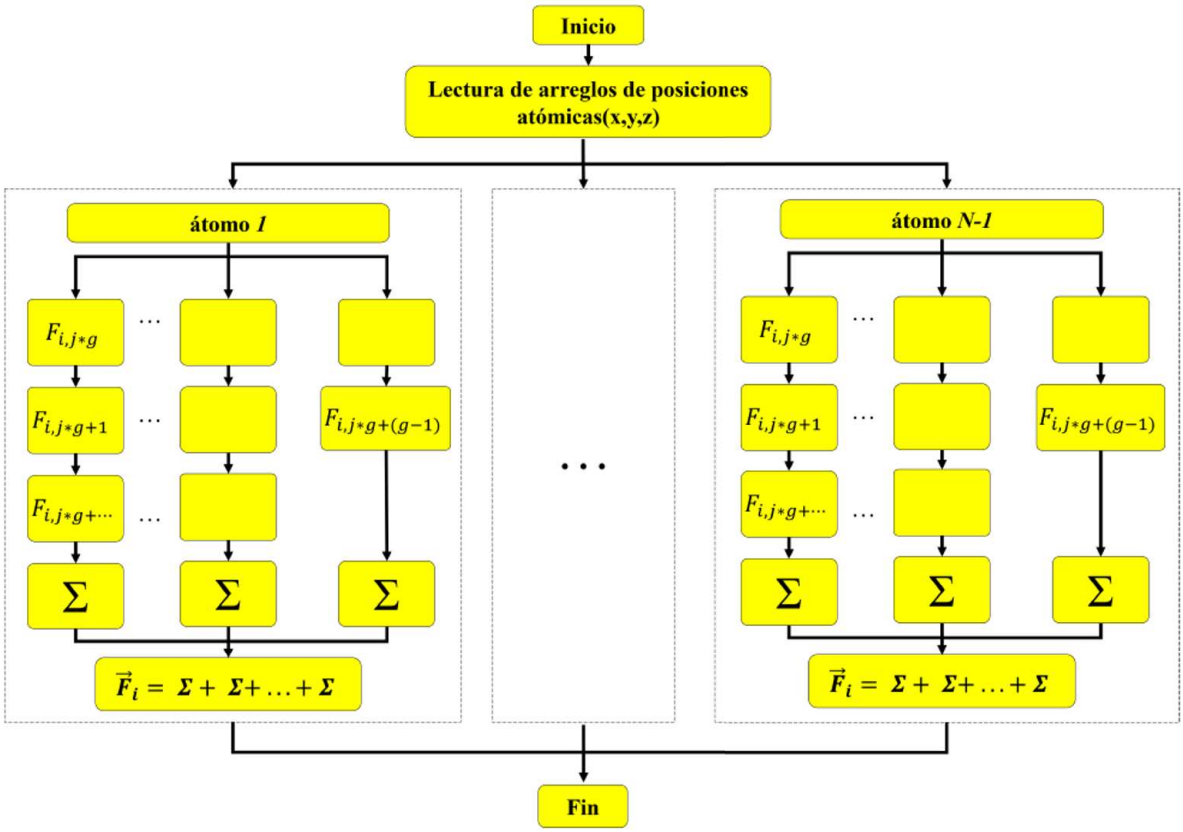


Figura 5.2: Algoritmo del programa INTERACTUA.CU.

La cantidad de bloques CUDA que pueden ser creados en esta tarjeta es su

eficiente para poder procesar un átomo por bloque, esto es de gran utilidad ya que permite dividir aún más el procesamiento de acuerdo al paradigma de paralelización de CUDA, es decir, dentro de cada bloque 128 hilos se encargan de determinar las fuerzas ejercidas por los demás átomos sobre el átomo en cuestión y se almacena en la memora compartida de cada bloque. Al finalizar, un hilo (hilo 0) de cada bloque se encarga de recolectar los datos y enviarlos a la memoria global, con la cual solamente se tiene comunicación al inicio y final de la determinación de las fuerzas totales. Así, el trabajo simultáneo dentro de cada bloque se ve optimizado de acuerdo al paradigma de paralelización de CUDA.

5.3. Subprograma

El algoritmo descrito en la sección anterior se codificó empleando la estructuración de los subcálculos en bloques e hilos, la estructura jerárquica de la memoria y los comandos básicos de CUDA. El código resultante está descrito en la figura 5.3 Sobre el código se agregó esquemas del proceso que se realiza en cada sección.

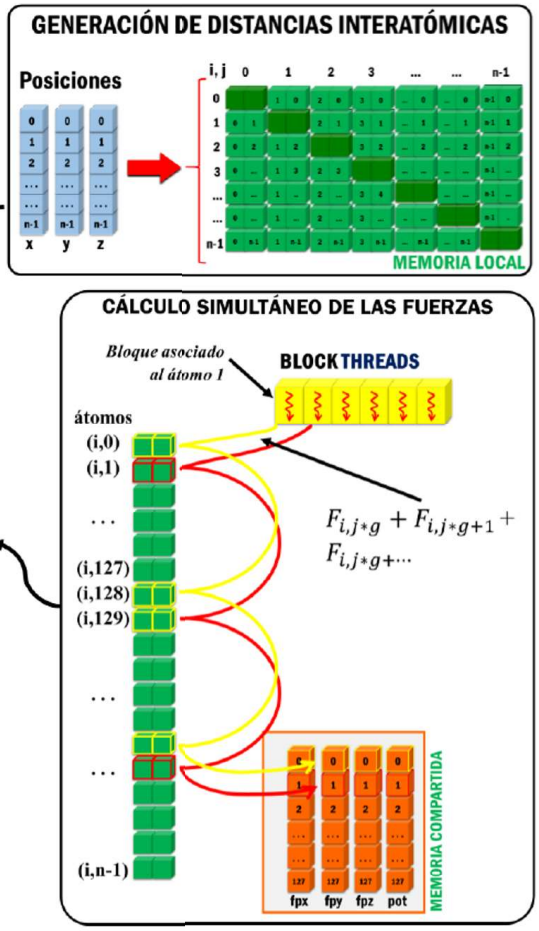
```
# define grupo 128

__global__ void interactua(
                /*posición*/ double *x, double *y, double *z,
                /*número de átomos*/ int n,
                /*potencial de interacción*/ double *upot,
                /*fuerzas*/ double *fx, double *fy, double *fz)
{
//  Declaración de variables compartidas por los hilos de cada bloque
    __shared__ double fpx[grupo];
    __shared__ double fpy[grupo];
    __shared__ double fpz[grupo];
    __shared__ double pot[grupo];

//  Declaración de variables
    /*iteraciones*/ int i,j,l;
    /*distancias*/  double dx,dy,dz,r2,ir2,p6ir,p8ir,p14ir,
    /*fuerza*/      f,sx,sy,sz,sp;
    /*hilos*/       int hilx = blockIdx.x;
                    int hily = threadIdx.x;
```

Espacio de memoria global

Espacio de memoria compartida

Espacio de memoria local

```
//  Inicialización de fuerzas parciales
    sx=0.0; sy=0.0; sz=0.0; sp=0.0;

    i = hilx;
    for(j=hily;j<n;j+=grupo){
        if(j!=i){
          dx=x[i]-x[j];
          dy=y[i]-y[j];
          dz=z[i]-z[j];
          r2=dx*dx+dy*dy+dz*dz;
          ir2=1.0/r2;
          p6ir=ir2*ir2*ir2;
          p8ir=p6ir*ir2;
          p14ir=p6ir*p6ir*ir2;
          sp+=4.0*p6ir*(p6ir-1.0);
          f=24.0*(2.0*p14ir-p8ir);
          sx+=f*dx;
          sy+=f*dy;
          sz+=f*dz;}}
    __syncthreads();
    fpx[hily]=sx;
    fpy[hily]=sy;
    fpz[hily]=sz;
    pot[hily]=sp;
    __syncthreads();

l = grupo/2;
```



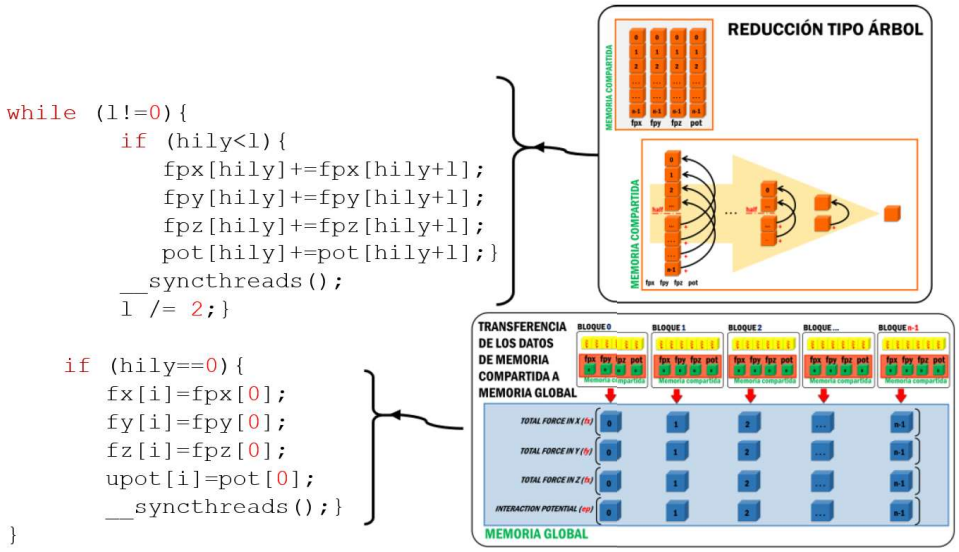


Figura 5.3: Código del programa INTERACTUA.CU.

La instrucción que se utiliza para el llamado al *kernel* se muestra en la figura 5.4.

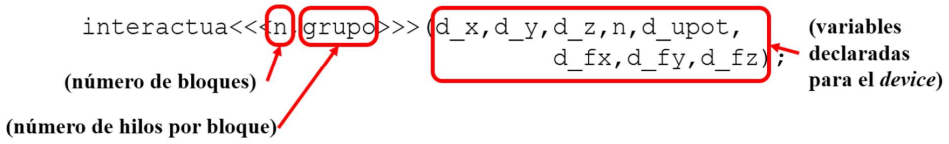


Figura 5.4: Llamado al device.

A excepción de esta sección del programa, el código es escrito en lenguaje C. Ya que CUDA es una derivación de C, las instrucciones son bastante semejantes y permite tener un código claro y consistente entre lenguajes. En la imagen siguiente se muestra el código únicamente para el cálculo de las fuerzas en el *device*, cabe destacar el proceso de llamado y transferencia de archivos entre el *host* y el *device*.

```
/*****************************************************************/
*       Programa para calcular las fuerzas ejercidas sobre cada átomo en
*       un cúmulo de 1000 átomos. En el anfitrión se realiza la lectura
*       de los datos de entrada, se hace el llamado la kernel y se alma_
*       cenan las fuerzas totales. En el huésped se calculan en paralelo
*       las fuerzas sobre cada átomo.
*****************************************************************/

//Librerías utilizadas
#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>

//Constantes globales
#define grupo 128
#define maxatom 3871

/*************************************************************/
*       KERNEL - Función interactúa
*       Cálculo de las fuerzas sobre cada átomo utilizando la
*       Interacción a pares de cada átomo.
*       Determinación del potencial de interacción mediante
*       Lennad-Jones para la interacción a pares de cada átomo.
*       Const: n
*       Var-I: /pos/ x, y, z,
*       Var-O: /fuerza/ fx, fy, fz, /potencial/ upot,
*************************************************************/
__global__ void interactua(
               /*posición*/ double *x, double *y, double *z,
               /*número de átomos*/ int n,
               /*potencial de interacción*/ double *upot,
               /*fuerzas*/ double *fx, double *fy, double *fz){

//   Declaración de variables compartidas por los hilos de cada bloque
     __shared__ double fpx[grupo];
     __shared__ double fpy[grupo];
     __shared__ double fpz[grupo];
     __shared__ double pot[grupo];

//   Declaración de variables
     /*interacciones*/ int i,j,l;
     /*distancias*/    double dx,dy,dz,r2,ir2,p6ir,p8ir,p14ir,
     /*fuerza*/        f,sx,sy,sz,sp;
     /*hilos*/         int hilx = blockIdx.x;
                       int hily = threadIdx.x;

//   Inicialización de fuerzas parciales
     sx=0.0; sy=0.0; sz=0.0; sp=0.0;
```

```
/*-------------------------Inicia ciclo principal-----------------------*/
int main()
{

//   Declaración de variables
     int     /*iteracciones*/ i,
             /*número de átomos*/ n;
     double  /*energía U total*/ utot;

//   Declaración de arreglos del host
     double h_x[maxatom];
     double h_y[maxatom];
     double h_z[maxatom];
     double h_upot[maxatom];
     double h_fx[maxatom];
     double h_fy[maxatom];
     double h_fz[maxatom];

//   Lectura del número de átomos en el cúmulo
     FILE *f0;
     f0=fopen("particula.dat","r");
     if(f1=NULL){printf("\nError de apertura de archivo.\n");
       else{fscanf(f0,"%d",&n);}

//   Lectura de la configuración de un estado dinámico
     FILE *f0;
         f0=fopen("configuración.dat","r");
         if(f1=NULL){printf("\nError de apertura de archivo.\n");
           else{
               for(i=0;i<n;i++){
                  fscanf(f1,"%lf %lf %lf",&h_x[i],&h_y[i],&h_z[i]);}}

//   Impresión de posiciones atómicas
     printf("Posiciones atómicas\n");
     for(i=0;i<n;i++){
        printf("%d % .10lf % .10lf % .10lf\n",i,h_x[i],h_y[i],h_z[i]);}

//   Declaración de arreglos usados en el device
     double *d_x = NULL;
     double *d_y = NULL;
     double *d_z = NULL;
     double *d_upot = NULL;
     double *d_fx = NULL;
     double *d_fy = NULL;
     double *d_fz = NULL;
     cudaMalloc((void**)&d_x,n*sizeof(double));
     cudaMalloc((void**)&d_y,n*sizeof(double));
     cudaMalloc((void**)&d_z,n*sizeof(double));
```

```
      cudaMalloc((void**)&d_upot,n*sizeof(double));
      cudaMalloc((void**)&d_fx,n*sizeof(double));
      cudaMalloc((void**)&d_fy,n*sizeof(double));
      cudaMalloc((void**)&d_fz,n*sizeof(double));

//    Copia de la configuración del host al device
      cudaMemcpy(d_x,h_x,n*sizeof(double),cudaMemcpyHostToDevice);
      cudaMemcpy(d_y,h_y,n*sizeof(double),cudaMemcpyHostToDevice);
      cudaMemcpy(d_z,h_z,n*sizeof(double),cudaMemcpyHostToDevice);

//    Llamado al kernel para ejecutarlo en el device
      interactua<<<n,grupo>>>(d_x,d_y,d_z,
                              n,
                              d_upot,
                              d_fx,d_fy,d_fz)

//    Copia de la configuración del host al device
      cudaMemcpy(h_upot,d_upot,n*sizeof(double),cudaMemcpyDeviceToHost);
      cudaMemcpy(h_fx,d_fx,n*sizeof(double),cudaMemcpyDeviceToHost);
      cudaMemcpy(h_fy,d_fy,n*sizeof(double),cudaMemcpyDeviceToHost);
      cudaMemcpy(h_fz,d_fz,n*sizeof(double),cudaMemcpyDeviceToHost);

//    Llamado de fuerzas totales y potencial de interacción
      printf("\n\n");
      printf("Fuerza total ejercida sobre cada átomo\n");
      for(i=0;i<n;i++){
         printf(" %d % .10lf % .10lf % .10lf",i,h_fx[i],h_fy[i],h_fz[i]);}

//    Interacción de la energía de interacción por átomo
      printf("\n\n");
      printf("Energía de interacción\n");

      for(i=0;i<n;i++){
         utot+=h_upot[i];       // Acumulación
         printf("% .10lf\n",h_upot[i]);}        // Impresión de energía por átomo

      utot=0.5*utot;     // Energía total de interacción

//    Impresión de la energía de interacción total
      printf("\n\n");
      printf("Energía total de interacción\n");
      printf("% .10lf\n",utot);

//    Liberación de memoria en el host
      Free(h_x);
      Free(h_y);
      Free(h_z);
      Free(h_upot);
```

Llamado al kernel

```
        Free(h_fx);
        Free(h_fy);
        Free(h_fz);

//      Liberación de memoria en el device
        cudaFree(d_x);
        cudaFree(d_y);
        cudaFree(d_z);
        cudaFree(d_upot);
        cudaFree(d_fy);
        cudaFree(d_fz);
        cudaFree(d_fz);

return 0;
}
```

Figura 5.5: Código FUERZA.CU para transferencia de datos entre host y device.

5.4. Ejemplo de Ejecución

Las figuras 5.6 y 5.7 muestran los archivos de entrada y salida, respectivamente, utilizados en la ejecución. En este caso es un cúmulo compuesto por $N = 1000$ átomos. El formato del archivo de entrada es mostrado en la figura 5.6

átomo:	x	y	z
0	0.416613	0.019724	1.204809
1	0.690190	-0.821800	0.658039
2	-0.295020	-0.500291	0.660023
...	. . .	. . .	. . .
...	. . .	. . .	. . .
...	. . .	. . .	. . .
N-3	-4.022963	-1.797735	-3.736121
N-2	-2.724171	3.646723	-3.741703
N-1	-4.660764	-0.889811	-3.721657

Figura 5.6: Estructura del archivo *configuracion.dat*.

Este archivo de entrada proporciona las posiciones de los átomos del cúmulo a la CPU. Las posiciones mostradas corresponden a un cúmulo compuesto por $N = 1000$ átomos en equilibrio mecánico. La ejecución del programa genera como resultado tres

arreglos, uno para cada componente de la fuerza (f_x, f_y, f_z) y un arreglo para el potencial de interacción de cada átomo con el resto del cúmulo, un ejemplo del archivo generado se muestra en la 5.7.

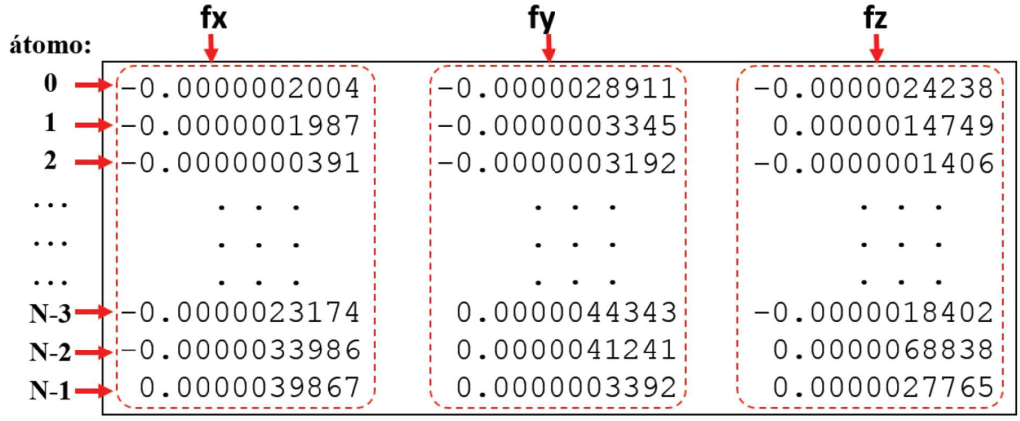


Figura 5.7: Archivo de componentes cartesianas f_x, f_y, y f_z de las fuerzas atómicas de un cúmulo de $N = 1000$ átomos.

Se seleccionó una configuración en equilibrio mecánico de un cúmulo Lennard-Jones de 1000 átomos tomada de la página web http://doye.chem.ox.ac.uk/jon/structures/LJ.html, de Jonathan Doye; al ejecutar el código en el *device*, debido a que el cúmulo se encuentra en equilibrio mecánico las fuerzas deber ser iguales a cero. Sin embargo, debido a los errores numéricos presentes en todo cálculo, los valores resultantes son cercanos al cero. Todas las tareas se realizan en un único ingreso al *huésped*. El resultado de esta labor es satisfactoria al comparar los tiempos de ejecución usando CPUs y GPUs. La fracción del tiempo correspondiente a la comunicación *anfitrión-huésped* es menor a mayor tama no del cúmulo, es decir, al tener cúmulos con mayor número de átomos más cálculos son realizados en la misma llamada al *huésped*.

De este modo se puede aseverar que una GPU K20 realiza un cálculo con una rapidez del orden de 70 veces la velocidad con que éste es ejecutado por un núcleo de una CPU. Este hecho fue evidenciado por el tiempo requerido para el cálculo de una evolución a

lo largo de una partición temporal de un millón de instantes. Un núcleo de CPU AMD Opteron 6272 tiene un desempe no de 16,8 GFLOPS, mientra que un sólo núcleo de la tarjeta gráfica Kepler K20 tiene un rendimiento de 0,469 GFLOPS. La tarjeta gráfica Tesla Kepler K20 requirió 38 minutos mientras que un núcleo de CPU tardó alrededor de dos días. El núcleo de la CPU es 33 veces más rápido que el de la GPU. Por lo tanto, el cómputo en paralelo en las tarjetas gráficas permite mejorar el rendimiento en la ejecución de programas, debido a la enorme cantidad de núcleos. En particular, la GPU empleada posee 2496 núcleos. La confiabilidad de los resultados generados por el código es monitoreada mediante el registro de los errores numéricos en la evolución calculada. Los cálculos numéricos inducen en la conservación de la energía un error del orden de 10^{-6} en una evolución consistente de un millón de pasos de integración en cúmulos de alrededor de 4000 átomos. Así, se confirma tanto la veracidad de los resultados del programa codificado, como la capacidad de la GPU para realizar cálculos numéricos con alta precisión.

Capítulo 6

Conclusión

Esta investigación resuelve el problema de la inexistencia de códigos de cómputo de alto desempe no que calculen propiedades no determinadas por los programas existentes. Esto se logró mediante la creación del código semilla presentado. Con base en éste, es posible codificar programas de dinámica molecular para GPUs que determinen la evolución de cúmulos de cualquier sustancia (agua, platino, iridio, dióxido de titanio, oro, etc.). Evidentemente, siempre y cuando, se disponga de un modelo para describir la interacción atómica existente entre sus átomos. Los objetivos de este trabajo de investigación fueron alcanzados: Se creó y evaluó el desempe no del programa (ver capítulo 5) codificado en CUDA para determinar el movimiento de los átomos en un cúmulo libre de gas noble. Mediante su estructuración, se describieron los elementos básicos y el proceso a seguir para codificar cualquier otro programa de dinámica molecular para GPU utilizando lenguaje CUDA. En resumen, para la codificación de un programa, es necesario que el conjunto de cálculos a realizar posea independencia entre estos cálculos, de forma que ellos puedan realizarse simultáneamente. Estos subcálculos deben organizarse considerando la jerarquía de procesamiento de CUDA. En general, dentro de esta jerarquía, la unidad mínima de procesamiento es el hilo. Los hilos se organizan en bloques, los cuales a su vez en mallas, y cada componente de esta jerarquiá posee

un espacio de memoria con características específicas. Además, es necesario tener en cuenta la estructura básica de la GPU: núcleos y Multiprocesadores (Streaming Multiprocessors). Tomar en cuenta estos elementos básicos permite una correcta codificación del programa y un correcto uso de la capacidad de las tarjetas gráficas. Así, la hipótesis fue confirmada, la codificación del programa de Dinámica Molecular paralelizado para GPU utilziando el paradigma de CUDA contiene los elementos básicos descritos con anterioridad.

Esta tesis es, por sí misma, el primer esfuerzo que realiza la UNICACH para asimilar la tecnología de cómputo de alto desempe no más avanzada en la actualidad. Además, el código semilla permite construir cualquier otro código, tanto de dinámica molecular como de cualquier otra índole. Por lo que es de utilidad en otras áreas de conocimiento que se cultivan en la Universidad. La difusión adecuada de esta tesis al interior de la institución permitirá el uso de las GPUs en las distintas investigaciones que se realizan, disminuyendo sus costos y tiempos de obtención de resultados y extendiendo sus alcances. Así mismo, es importante destacar tambiíen que los resultados de la presente investigación se presentaron en un cartel el 6th International Supercomputing Conference in Mexico, realizado en la ciudad de México en marzo del 2015, ver Apéndice B.

Apéndice A

Programa de Dinámica Molecular

Se presenta la construcción del algoritmo para determinar la evolución del cúmulo sobre una partición temporal y su codificación en CUDA-C. El proceso está constituído en dos secciones: 1) los subprocesos secuenciales que son ejecutados en el CPU, estos son la determinación de las configuraciones por las que atraviesa el cúmulo, las propiedades del mismo y la administración de los subprocesos y 2) los subprocesos de ejecución simultánea, que se llevan a cabo en la GPU, estos son los referentes al cálculo de las fuerzas sobre el cúmulo en cada elemento de la partición.

A.1. Algoritmo

La codificiación del programa sigue el procedimiento descrito en el diagrama A.1.

Figura A.1: Algoritmo del programa general.

A continuación se detallan los elementos del algoritmo mostrado:

- **Lectura de archivos**. El primer módulo del programa realiza una lectura de

datos contenidos en dos archivos, uno de ellos contiene algunas características del cúmulo, así como de la partición del tiempo sobre la cual evoluciona el cúmulo, las características son las siguientes:

- Número de átomos del cúmulo.
- Energía inicial del cúmulo.
- Número de elementos que componen la partición temporal en subintervalos.
- Incremento entre elementos de la partición.

El otro archivo contiene un conjunto de posiciones y velocidades de los átomos del cúmulo a partir del cual se obtiene el estado dinámico inicial, también se determina el elemento inicial de la partición del tiempo($t = 0$). Una vez que se ha configurado el inicio de la evolución, la simulación utiliza las expresiones mencionadas en el capítulo 5 para generar los distintos estado dinámicos por los que atraviesa el cúmulo a lo largo de la partición del tiempo, de acuerdo al incremento leído de archivo.

- **Reservación de memoria**. Las características leídas de archivo permiten determinar el tama no de los arreglos a utilizar. Así, se reservan únicamente los espacios de memorias necesarios para la ejecución del programa.
- **Obtención de la configuración del cúmulo en el instante inicial**. A partir del estado dinámico leído se obtienen las configuraciones y velocidades a partir de las cuales evolucionará el cúmulo.
- **Cálculo de las fuerzas interatómicas**. Ver capítulo 4.
- **Determinación del cúmulo en el elemento siguiente de la partición**. Cada uno de los estados dinámicos por los que atraviesa el cúmulo es determinado de

manera serial utilizando la configuración del cúmulo en dos elementos consecutivos anteriores al elemento de la partición actual. Esta tarea es ejecutada a lo largo de toda la partición.

- **Determinación de las propiedades del cúmulo**. En determinados elementos de la partición se obtienen el calor específico, temperatura, promedios cortos de energía cinética y fluctuaciones numéricas.

- **Liberación de la memoria**. Como último subproceso se libera la memoria que se utilizó en la ejecución del código.

A.2. Codificación

En la figura A.2 se muestra el programa principal de dinámica molecular, a partir del cual se realizan el cálculo de los estados dinámicos por los que atraviesa el cúmulo, el cálculo de las propiedades termodinámicas, la lectura y escritura de archivos, el llamado a la GPU y la transferencia de datos con la tarjeta gráfica. El llamado al *kernel* se realiza en cada instante de la partición, es decir, la comunicación entre la CPU y la GPU es constante a lo largo de toda la evolución. Cada uno de estos elemento está marcado dentro del código para su identificación.

```
/*---------------------Inicia ciclo principal----------------------*/
int main()
{
//  Declaración de variables
    int     /*iteraciones*/ i,k,m,
            /*átomos*/ n,
            /*partición*/ nps,nse,neresp,ntresp,npe,npresp,npt,kae,kas,karesp,
            /*histograma*/ lochis,lhis,khis,nhis[nhismax],
            /*energía total*/ meps=1000;
    double /*iteraciones*/ t,
            /*paso de integración*/ dt,dt2,dtsq,dtsqh,degrid,
            /*energía cinética*/ ek,ekk,ekit,eks,ekt,pekt,pekit,ekmax,
            /*histograma*/ dhis,pmdhis,
            /*Termodinámica*/ temp,capcal,
            /*factor de velocidades*/ facv=0.0,
            /*potencial*/ ucoh=0.0,ucoht,pucoht,
            /*energía total*/ et0,e0g,et2t,etk,pett,pet2t,ettf;

    FILE *f0;
    f0=fopen("particula.dat","r");
    fscanf(f0,"%d%lf%d%d%d%d%lf%d%lf",
              &n,&dt,&nps,&nse,&neresp,&ntresp,&e0g,&lhis,&degrid);

//Energía de referencia y cambio de energía del sistema completo
    e0g = n*e0g;
    degrid = n*degrid;

//  Determinación de contadores de la partición
    npe = nps*nse;
    npresp = npe*neresp;
    npt = ntresp*npresp;

//  Propagador
    dt2=dt*2.0;
    dtsq=dt*dt;
    dtsqh=dtsq*0.5;

    double *x=(double*)malloc(sizeof(double)*n);
    double *y=(double*)malloc(sizeof(double)*n);
    double *z=(double*)malloc(sizeof(double)*n);
    double *fx=(double*)malloc(sizeof(double)*n);
    double *fy=(double*)malloc(sizeof(double)*n);
    double *fz=(double*)malloc(sizeof(double)*n);
    double *upot=(double*)malloc(sizeof(double)*n);
```

Lectura de características de la partición y número de átomos

Reservación de espacios de memoria en la CPU

```
double *vx=(double*)malloc(sizeof(double)*n);
double *vy=(double*)malloc(sizeof(double)*n);
double *vz=(double*)malloc(sizeof(double)*n);
double *xold=(double*)malloc(sizeof(double)*n);
double *yold=(double*)malloc(sizeof(double)*n);
double *zold=(double*)malloc(sizeof(double)*n);
```

Reservación de espacios de memoria en la CPU

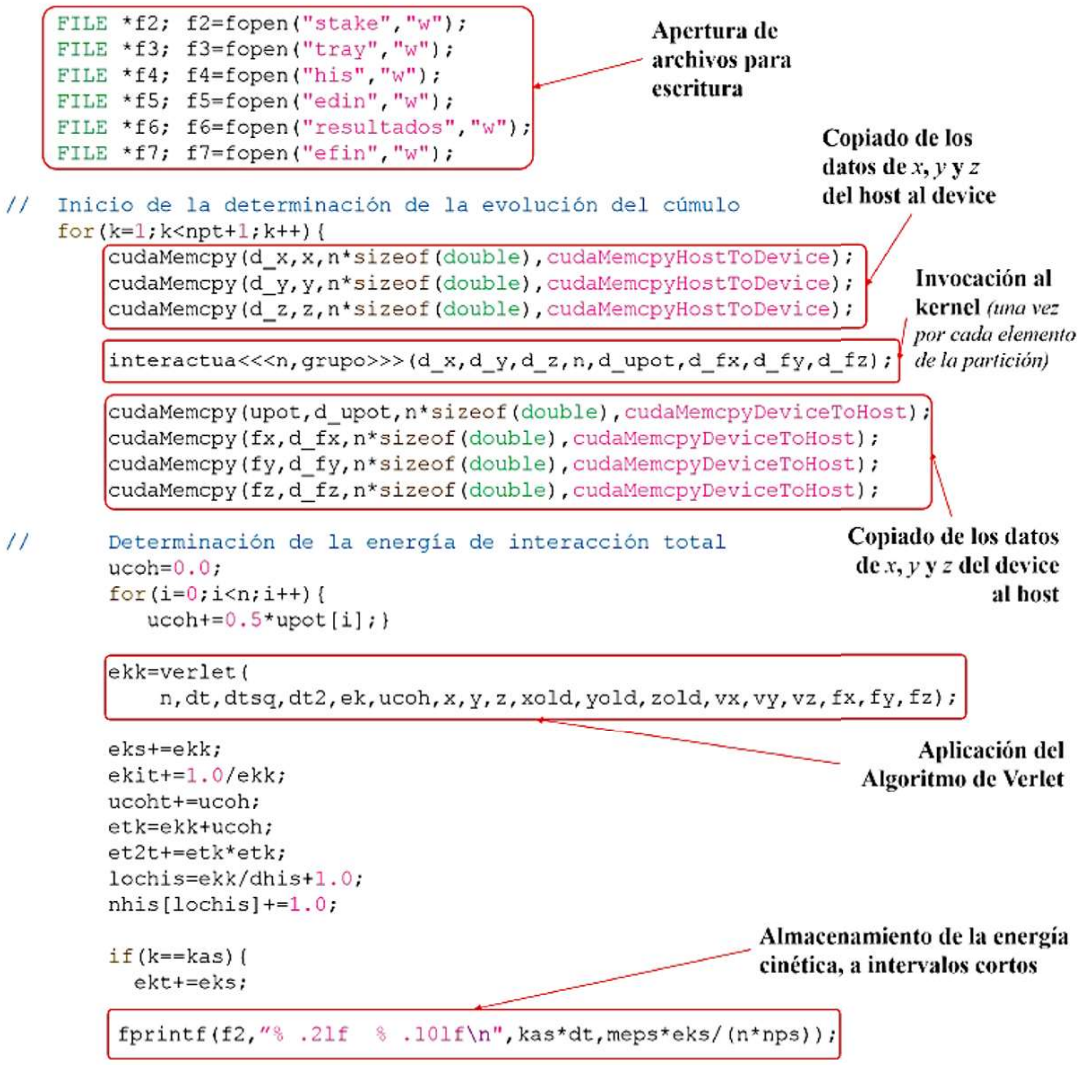
```
cudaMalloc((void**)&d_x,n*sizeof(double));
cudaMalloc((void**)&d_y,n*sizeof(double));
cudaMalloc((void**)&d_z,n*sizeof(double));
cudaMalloc((void**)&d_upot,n*sizeof(double));
cudaMalloc((void**)&d_fx,n*sizeof(double));
cudaMalloc((void**)&d_fy,n*sizeof(double));
cudaMalloc((void**)&d_fz,n*sizeof(double));
```

Reservación de espacios de memoria en la GPU

```
FILE *f1;
f1=fopen("configuracion.dat","r");
for(i=0;i<n;i++){
   fscanf(f1,"%lf %lf %lf",&x[i],&y[i],&z[i]);}
for(i=0;i<n;i++){
   fscanf(f1,"%lf %lf %lf",&vx[i],&vy[i],&vz[i]);}}
```

Lectura de un estado dinámico

```
// Obtención de la configuración del cúmulo en el instante inicial (t=0)
   calct0(n,facv,degrid,dt,dtsqh,&et0,e0g,&ekmax,upot,ucoh,d_upot,&dhis,lhis,
          x,y,z,d_x,d_y,d_z,xold,yold,zold,vx,vy,vz,fx,fy,fz,d_fx,d_fy,d_fz);

//  Contadores de puntos por intervalo
    kas=nps;
    kae=npe;
    karesp=npresp;

//  Inicialización de las propiedades físicas
    eks=0.0;
    ekt=0.0;
    ucoht=0.0;
    et2t=0.0;
    ekit=0.0;

    for(i=0;i<lhis;i++){
        nhis[i]=0.0;}
```


FILE *f2; f2=fopen("stake","w");
FILE *f3; f3=fopen("tray","w");
FILE *f4; f4=fopen("his","w");
FILE *f5; f5=fopen("edin","w");
FILE *f6; f6=fopen("resultados","w");
FILE *f7; f7=fopen("efin","w");
Apertura de archivos para escritura
Copiado de los datos de x, y y z del host al device
// Inicio de la determinación de la evolución del cúmulo
for(k=1;k<npt+1;k++){
cudaMemcpy(d_x,x,n*sizeof(double),cudaMemcpyHostToDevice);
cudaMemcpy(d_y,y,n*sizeof(double),cudaMemcpyHostToDevice);
cudaMemcpy(d_z,z,n*sizeof(double),cudaMemcpyHostToDevice);
Invocación al kernel (una vez por cada elemento de la partición)
interactua<<<n,grupo>>>(d_x,d_y,d_z,n,d_upot,d_fx,d_fy,d_fz);
cudaMemcpy(upot,d_upot,n*sizeof(double),cudaMemcpyDeviceToHost);
cudaMemcpy(fx,d_fx,n*sizeof(double),cudaMemcpyDeviceToHost);
cudaMemcpy(fy,d_fy,n*sizeof(double),cudaMemcpyDeviceToHost);
cudaMemcpy(fz,d_fz,n*sizeof(double),cudaMemcpyDeviceToHost);
Copiado de los datos de x, y y z del device al host
// Determinación de la energía de interacción total
ucoh=0.0;
for(i=0;i<n;i++){
ucoh+=0.5*upot[i];}
ekk=verlet(
n,dt,dtsq,dt2,ek,ucoh,x,y,z,xold,yold,zold,vx,vy,vz,fx,fy,fz);
Aplicación del Algoritmo de Verlet
eks+=ekk;
ekit+=1.0/ekk;
ucoht+=ucoh;
etk=ekk+ucoh;
et2t+=etk*etk;
lochis=ekk/dhis+1.0;
nhis[lochis]+=1.0;
Almacenamiento de la energía cinética, a intervalos cortos
if(k==kas){
ekt+=eks;
fprintf(f2,"% .2lf % .10lf\n",kas*dt,meps*eks/(n*nps));

```
if(k==kae){
 for(i=0;i<n;i++){
      fprintf(f3,"% .10lf  % .10lf  % .10lf\n",xold[i],yold[i],zold[i]);}
      for(i=0;i<n;i++){
      fprintf(f3,"% .10lf  % .10lf  % .10lf\n",vx[i],vy[i],vz[i]);}

   if(k==karesp){
//       Si se proporcionó la energía del mínimo global e0g:
//       Histograma de energías cinéticas instantáneas(milieps/atom,frecuencia)
       if(e0g<0.0){
          khis=0;
          for(m=1;m<lhis+1;m++){
              khis+=nhis[m];}
              fprintf(f4,"Total de ek's = %d Probable ek(meps/atom) máxima =
                    %e Valores de ek localizados =  % d\n",k,ekmax/n,khis);
          pmdhis=dhis/2.0;
           for(m=1;m<lhis+1;m++){
              fprintf(f4,"% .10lf %d\n",(meps*pmdhis)/n,nhis[m]);
              pmdhis+=dhis;}}

      pekt = ekt/karesp;
      pekit = ekit/karesp;
      pucoht = ucoht/karesp;
      pett = pekt+pucoht;
      pet2t = et2t/karesp;
      ettf = sqrt(abs(pet2t - pett*pett))/abs(pett);

      temp = 2.0/(3.0*n-8.0);
      temp = temp*(1.0/pekit);

      capcal = (1.0-2.0/(3.0*n-6.0))*pekt*pekit;
      capcal = 1.0 - capcal;
      capcal = 1.0/(capcal*n);

      for(i=0;i<n;i++){
          fprintf(f5,"% .10lf  % .10lf  % .10lf\n",xold[i],yold[i],zold[i]);}
          for(i=0;i<n;i++){
             fprintf(f5,"% .10lf  % .10lf  % .10lf\n",vx[i],vy[i],vz[i]);}
```

Almacenamiento de la trayectoria del estado dinámico

Almacenamiento de valores de energía cinética

Almacenamiento de la frecuencia de las energías cinéticas instantáneas encontradas

Cálculo de la energía cinética promedio y las fluctuaciones numéricas

Cálculo de la temperatura (K)

Capacidad calorífica (C / k_B*n)

Almacenamiento del estado dinámico

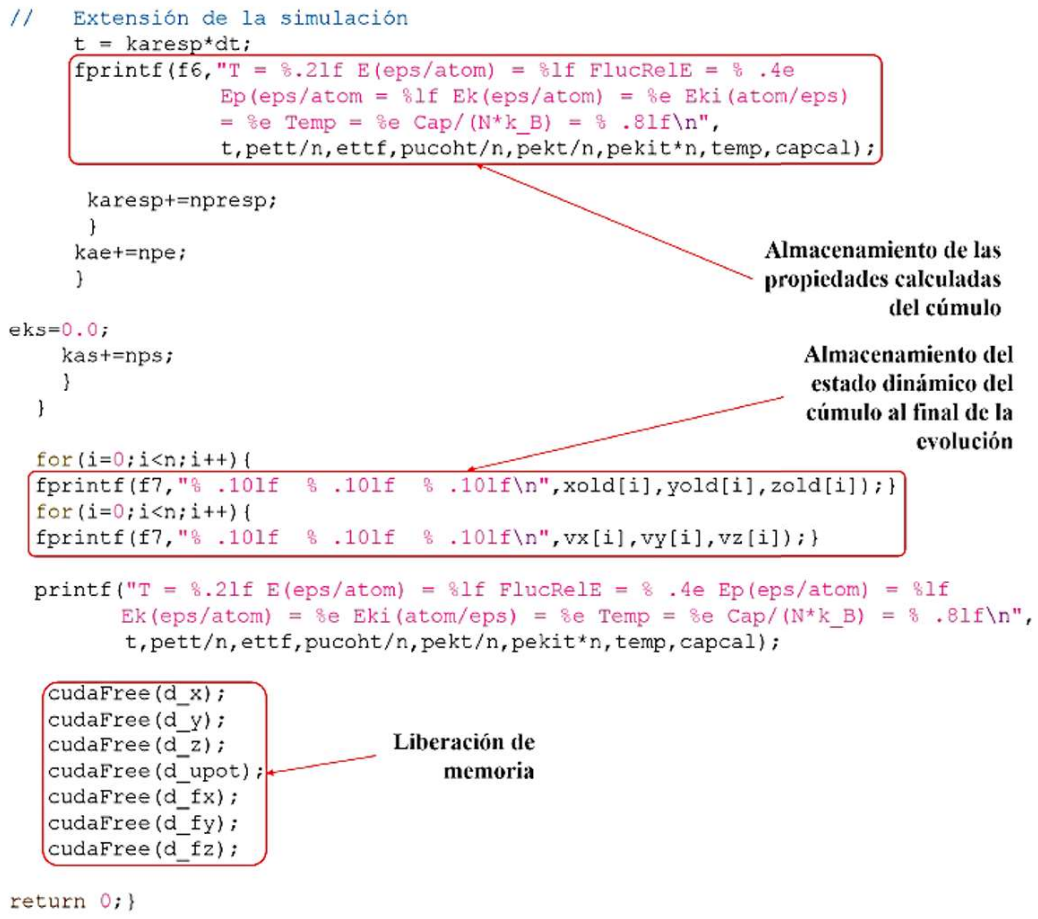


Figura A.2: Programa principal de DINAMICA.CU

A.3. Ejemplo de ejecución

Se simuló la evolución de un cúmulo de 3871 átomos de Argón, sobre una partición temporal de $1x10^6$ elementos a 30 diferentes niveles de energía. La tarjeta gráfica utilizada fue en una GPU Tesla Kepler K20, de la compa nía NVIDIA. El tiempo de ejecución total para este trabajo fue de 18 horas y 13 minutos. La ejecución inicia con la lectura de las condiciones iniciales del cúmulo y de la partición, dicho archivo se muestra en la figura A.3.

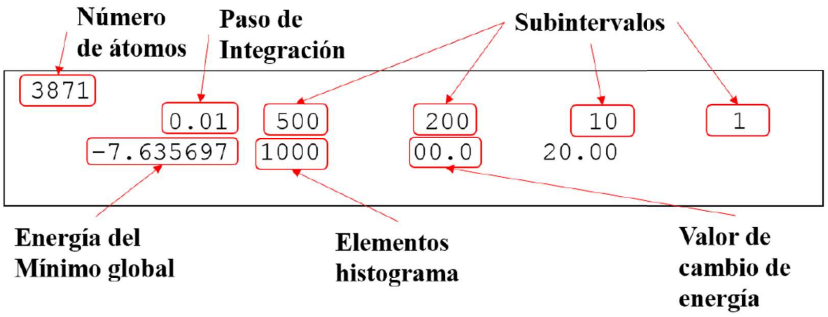


Figura A.3: Archivo de datos iniciales particula.dat.

El otro archivo contiene un estado dinámico del cúmulo con una energía específica. Este archivo es útil para poder realizar un escalamiento de velocidades y generar el estado dinámico inicial. La figura 5.6 del capítulo 4 muestra el formato de dicho archivo. Una vez que se deja evolucionar el cúmulo, se van generando archivos que poseen información del cúmulo a lo largo de su evolución en cada uno de los subintervalos indicados en el archivo configuracion.dat (fig A.3), uno de las propiedades fundamentales calculadas es la energía cinética. En el programa, la energía cinética del cúmulo a intervalos cortos es registrada en un archivo, el cual se presenta en la figura A.4.

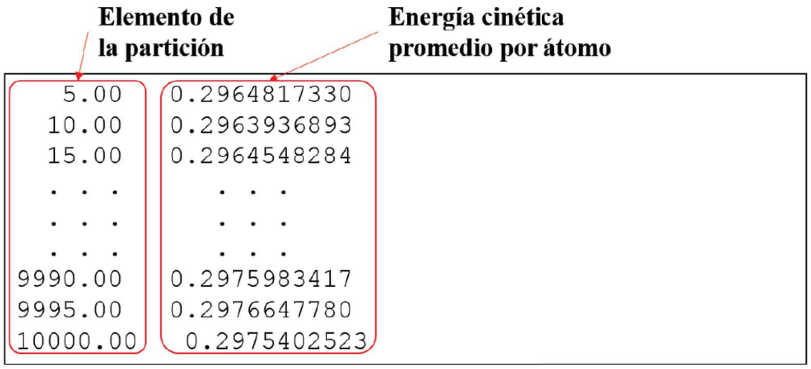


Figura A.4: Archivo stake.dat de energía cinética.

Así mismo, para la generación del histograma de energía cinética, se crea un archivo donde se almacena la frecuencia en la que los valores promedios de energía son encontrados. Este archivo se muestra en la imagen A.5.

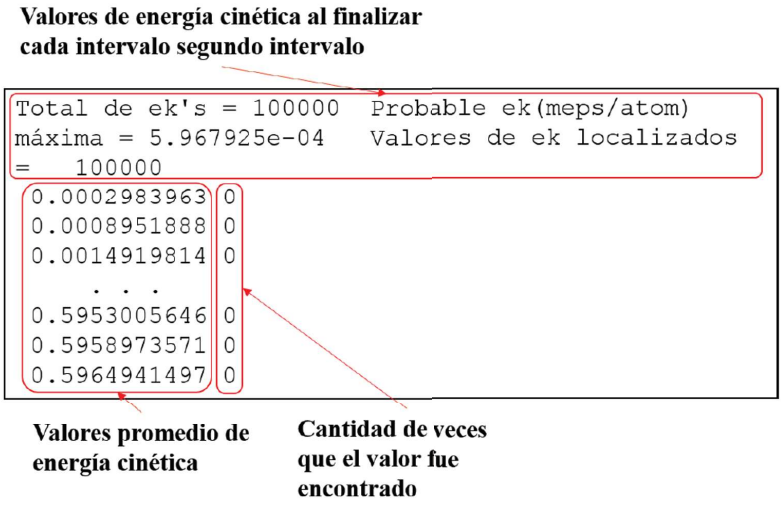


Figura A.5: Archivo his.dat de energía cinética.

De acuerdo a los subintervalos definidos al principio, se almacena un archivo que posee la configuración y la velocidad del cúmulo en determinados instantes de la partición. Con esto es posible iniciar otra evolución a partir de estos estados dinámicos, o bien, interrumpir y continuar la misma evolución. El archivo donde se almacenan estos estados dinámicos intermedios se muestra en la figura A.6

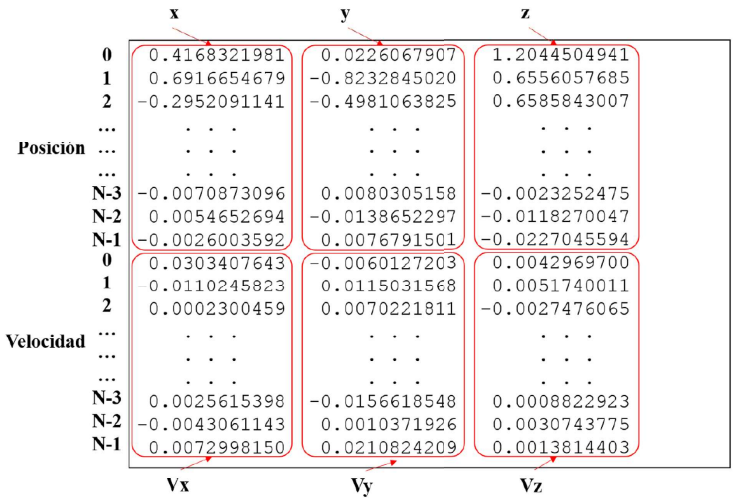


Figura A.6: Archivo tray.dat de estados dinámicos intermedios.

Así mismo se almacena el estado dinámico al

nal del subintervalo de mayor número de elementos con el mismo formato del archivo tray.dat. Éste recibe el nombre de edin.dat y se muestra en la figura A.7.

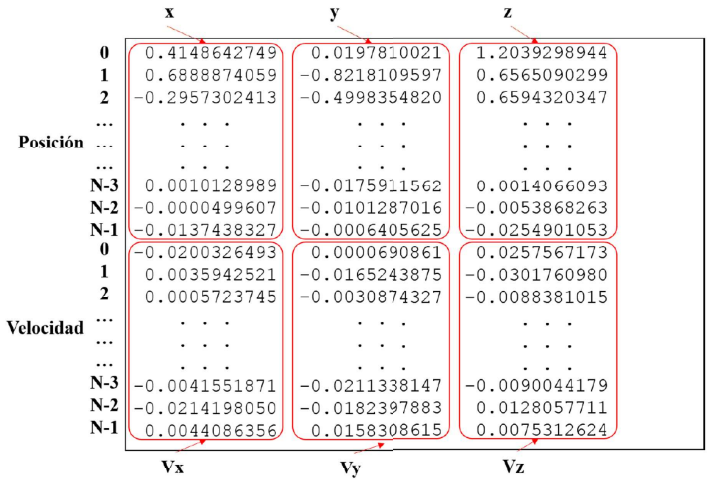


Figura A.7: Archivo edin.dat de estados dinámicos.

Al finalizar la evolución se crea un archivo que almacena las configuraciones y las velocidades de los átomos del cúmulo en el último instante de la partición, este archivo llamado efin.dat se muestra en la figura A.8.

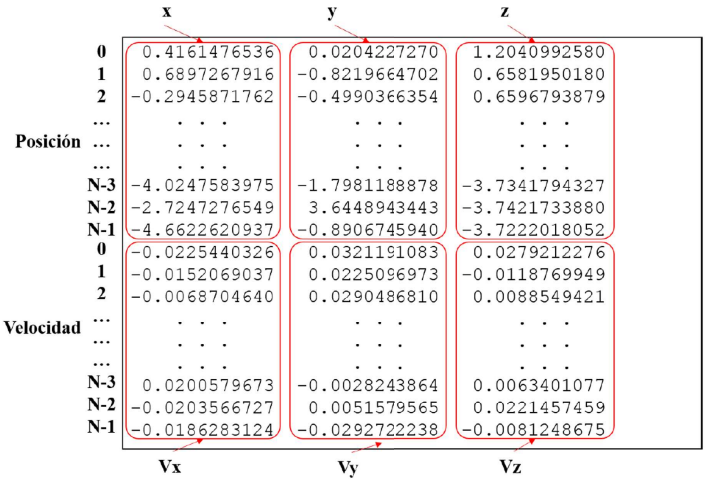


Figura A.8: Archivo efin.dat de estados dinámicos.

Para conocer las características del cúmulo a lo largo de la evolución, en el archivo

resultados.dat se almacena las propiedades del cúmulo al llegar al último elemento del segundo subintervalo. Esta propiedades son identi

cadas en la figura A.9.

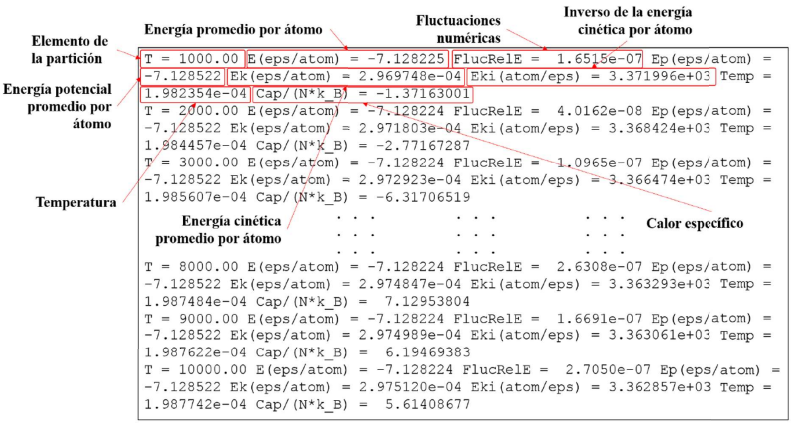


Figura A.9: Archivo resultados.dat de propiedades del cúmulo.

Los resultados de esta evolución se muestran en la gráfica A.10. Cada uno de los puntos rojos indican una evolución del cúmulo. Claramente se observa el aumento de temperatura conforme el cúmulo evoluciona a diferentes niveles de energía.

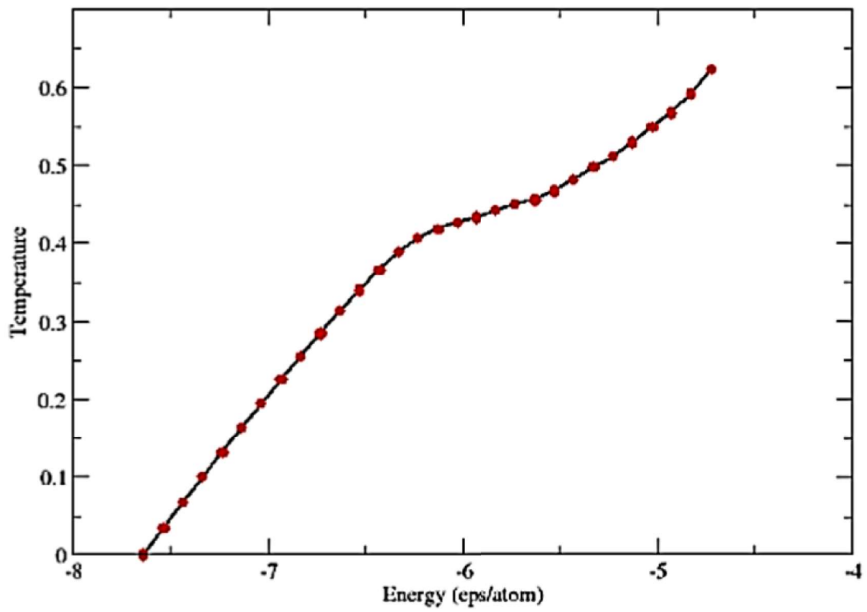


Figura A.10: Gráfica de energía contra temperatura del cúmulo de 3871 átomos.

Apéndice B

Cartel

En la siguiente página se presenta el cartel que fue expuesto en el 6th International Supercomputing Conference in Mexico, en marzo de 2015.

PARALLEL CALCULATION OF ATOMIC FORCES INSIDE A NANOPARTICLE BASED ON CUDA-C PARADIGM

A. Escobar López[1], J. A. Reyes Nava[1], F. Sagols Troncoso[2]
agustin.escobar.02@gmail.com, jareyesn@gmail.com,
(1) Centro de Investigación y Desarrollo Tecnológico en Energías Renovables, UNICACH,
(2) Departamento de Matemáticas, CINVESTAV

We present an algorithm and its codification in CUDA-C to the parallel computation of the forces induced on the N atoms of a tree noble gas cluster by their interaction at a given contiguration. The algorithm consists in 1) to separate the cluster into 128 fragments, and 2) to calculate the N forces by the simultaneous execution of N*128 processing threads, distributed in N blocks, each one exclusively dedicated to the determination of the total force of one atom. Any of them consists in the calculation and accumulation of the forces applied by each of the atoms of its corresponding fragment on the regarded atom. The resultant force on this atom is determined by the complete reduction of the forces applied on it by the 128 fragments.

INTERACTION MODEL

INTERACTION POTENCIAL MODEL [3]

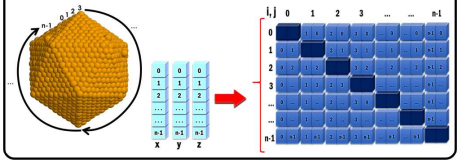
$$V(\vec{r}_1,\ldots,\vec{r}_n)=\sum_{\substack{j=1\\ j\neq i}}^{n}\sum_{i=1}^{n}4\epsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12}-\left(\frac{\sigma}{r_{ij}}\right)^{6}\right]$$

FORCES CALCULATION [4]

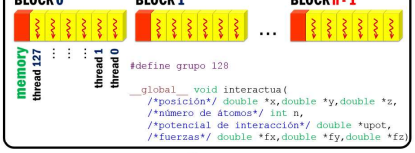
$$\vec{F}_i(\vec{r}_1,\ldots,\vec{r}_n)=-\vec{\nabla}_i V$$

$$\vec{F}_i(\vec{r}_1,\ldots,\vec{r}_n)=\sum_{\substack{j=1\\ j\neq i}}^{n}\frac{24\epsilon}{\sigma^2}\left[2\left(\frac{\sigma}{r_{ij}}\right)^{14}-\left(\frac{\sigma}{r_{ij}}\right)^{8}\right]\vec{r}_{ij}$$

MATRIX OF n X n COMBINATIONS

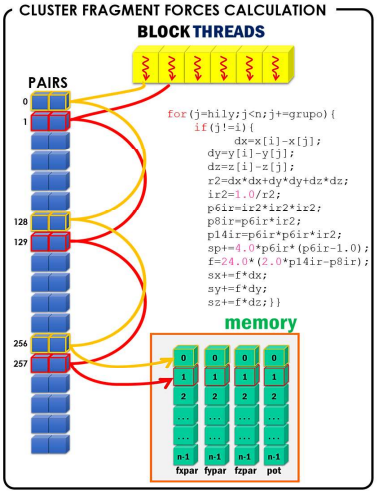


CUDA BLOCKS GENERATED



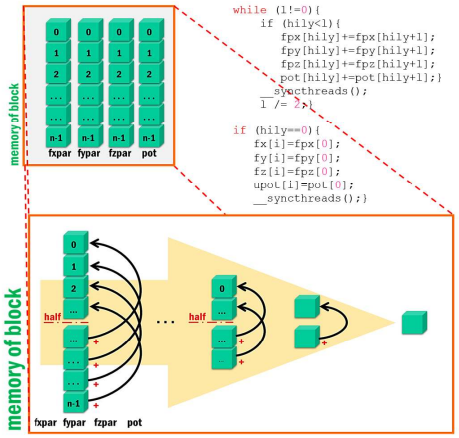

```
#define grupo 128

__global__ void interactua(
  /*posición*/ double *x,double *y,double *z,
  /*número de átomos*/ int n,
  /*potencial de interacción*/ double *upot,
  /*fuerzas*/ double *fx,double *fy,double *fz)
```

CLUSTER FRAGMENT FORCES CALCULATION



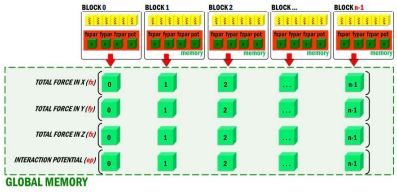

```
for(j=hily;j<n;j+=grupo){
  if(j!=i){
    dx=x[i]-x[j];
    dy=y[i]-y[j];
    dz=z[i]-z[j];
    r2=dx*dx+dy*dy+dz*dz;
    ir2=1.0/r2;
    p6ir=ir2*ir2*ir2;
    p8ir=p6ir*ir2;
    p14ir=p6ir*p6ir*ir2;
    sp+=4.0*p6ir*(p6ir-1.0);
    f=24.0*(2.0*p14ir-p8ir);
    sx+=f*dx;
    sy+=f*dy;
    sz+=f*dz;}}
```

REDUCTION BY SUM INSIDE BLOCKS [5]




```
while (l!=0){
    if (hily<l){
        fpx[hily]+=fpx[hily+l];
        fpy[hily]+=fpy[hily+l];
        fpz[hily]+=fpz[hily+l];
        pot[hily]+=pot[hily+l];}
    __syncthreads();
    l /= 2;}

if (hily==0){
    fx[i]=fpx[0];
    fy[i]=fpy[0];
    fz[i]=fpz[0];
    upot[i]=pot[0];
    __syncthreads();}
```

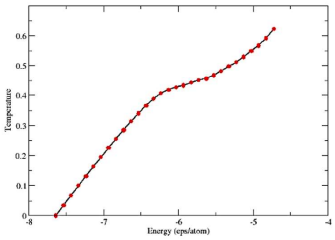
ACCUMULATION ON GLOBAL MEMORY



ROUTINE PERFORMANCE

Molecular Dynamics of a 3871 atoms LJ cluster.
Trajectories of 1x10⁶ integration steps solving motion equations by Verlet algorithm calculating atomic forces with a GPU Tesla Kepler K20.


```
interactua<<<n,grupo>>>(d_x,d_y,d_z,n,d_upot,d_fx,d_fy,d_fz);
```



6TH INTERNATIONAL SUPERCOMPUTING CONFERENCE IN MEXICO, MEXICO D.F., 2015

REFERENCES

1. NVIDIA Corporation: NVIDIA CUDA C programming guide, San Tomas Expressway, Santa Clara, CA (2012)
2. NVIDIA Corporation: CUDA C/C++ basics, San Tomas Expressway, Santa Clara, CA (2012)
3. J. Jellinek, T. L. Beck and R Stephen Berry: Solid-liquid phase changes in simulated isoenergetic Ar13, J Chemical Phys, 84, 2783 (1985)
4. N. Barrantes Melgar, H. Loro: Transiciones de fase y la función de distribución radial para el argón usando dinámica molecular, REVCIUNI, 12, 59 (2008)
5. NVIDIA Corporation: sample "Scalar product", from toolkit CUDA 5.0 installation.

Apéndice C

Glosario

Ab-initio. Se refiere a los métodos de química computacional que no incluyen ningún parámetro empírico o semi-empírico en sus ecuaciones, siendo derivadas directamente de principios teóricos, sin la inclusión de datos experimentales.

Ancho de Banda. Se refiere a la cantidad de información o de datos que se puede enviar a través de una conexión de red en un período de tiempo dado. El ancho de banda se mide en bites por segundo (BPS), kilobites por segundo (kbps), o megabites por segundo (mps).

Calor específico. Cantidad de calor que por kilogramo necesita un cuerpo para que su temperatura se eleve en un grado centígrado.

Control de Flujo. En ciencia computacional, control de flujo (o flujo de control) hace referencia a la especificación del orden en el cual las declaraciones individuales, instrucciones o llamados a funciones de un programa son ejecutadas o evaluadas. El énfasis en un control de flujo explícito distingue entre un lenguaje de programación imperativa (cómo realizar una tarea) y un lenguaje de programación declarativa (qué tarea ejecutar).

DFT. Siglas de Teoría Funcional de Densidad (Density Functional Theory) es un método de modelado computacional de mecánica cuántica usado en física, química y

estudio de materiales para investigar la estructura electrónica de sistemas de muchas partículas. En DFT, la energía total es expresada en términos de la densidad total en lugar de la función de onda. Los métodos DFT pueden ser muy precisos y con un bajo costo computacional.

DirectCompute. Es una API de reciente creación utilizada para desarrollo de aplicaciones de procesamiento en las GPUs, la cual se ejecuta sobre la actual arquitectura CUDA de NVIDIA en Windows VISTA y Windows 7.

Entorno de Ejecución. En ciencia computacional, es un estado de máquina virtual que suministra servicios para los procesos de un programa de computadora que se está ejecutando. Puede pertenecer al mismo sistema operativo, o ser creado por el software del programa en ejecución.

Estado dinámico. Conjunto de posiciones en el espacio y velocidades de los átomos de un cúmulo.

FLOPS. En informática, las operaciones de punto flotante por segundo son una medida del rendimiento de una computadora, especialmente en cálculos científicos que requieren un gran uso de operaciones de este tipo.

FORTRAN. Contracción de *Formula Translating System*, es un lenguaje de programación de alto nivel de propósito general adaptado al cálculo numérico y a la computación científica. Fue desarrollado por IBM en 1957 y es para aplicaciones científicas y de ingeniería.

Ley de Moore. Se refiere a la predicción realizada por Gordon E. Moore originada en la década de 1960, que establece que la velocidad del procesador o el poder de procesamiento total de las computadoras se duplica cada 24 meses.

Mapear. Hace un referencia a localizar y representar gráficamente la distribución relativa de las partes de un todo.

MPI. Contracción de *Message Passing Interface* es un estándar que de en la sintaxis

de las funciones de una biblioteca de paso de mensajes dise nada para ser usada en programas ejecutados en múltiples procesadores.

OpenACC. Es un estándar de programación en paralelo, dise nado para ejecutar instrucciones sobre sistemas heterogéneos de CPUs o GPUs. Fue desarrollado por las compa nías Cray, CAPS, NVIDIA y el PGI, conocido como Portland Group. La característica principal es que permite a los desarrolladores indicar al compilador las áreas que deben ser aceleradas, sin realizar modificaciones y adaptaciones al código existente. El objetivo de las directivas es indicar al compilador que paralelice específicamente ciertas secciones del código. El compilador se encarga del proceso de mover los datos desde y hacia la CPU y la GPU, y de mapear las tareas de cómputo en el procesador adecuado.

PCIe. Es un estándar de bus que permite la adaptación de tarjetas de expansión. PCIe tiene el mismo interfaz de software que el PCI, pero las tarjetas no son compatibles.

Streaming Multiprocessor. Nombre para designar a los núcleos CUDA, cada Streaming Multiprocessor, de las últimas series de tarjetas gráficas NVIDIA contiene 32 procesadores de simple precisión, banco de memoria interna y dos niveles de memoria caché. Estos Streaming Multiprocessors sólo pueden recibir una instrucción a la vez y la ejecutan simultáneamente ocupando 4 ciclos de reloj.

Transistor. En electrónica, dispositivo semiconductor que cierra o abre un circuito o amplifica una se nal; se emplea en circuitos integrados para generar bits.

Bibliografía

[1] U. S. Department of Energy Office of Science. Computational Materials Science and Chemistry: Accelerating Discovery and Innovation through Simulation-Based Engineering and Science. Technical report, U. S. Department of Energy, 2010.

[2] Jeffrey S. Vetter. *Contemporary High Performance Computing: From Petascale toward Exascale*, volume 2. Chapman & Hall/CRC, 2015. Capítulo 1.

[3] High Performance Computing Innovation Center. PHC Boosts Industry. `http://hpcinnovationcenter.llnl.gov/hpc-boosts-industry`, 2015.

[4] Theoretical and Computational Biophysisc Group. GPU Acceleration of Molecular Modeling Applications. `http://www.ks.uiuc.edu/Research/gpu/`, 2015.

[5] Theoretical and Computational Biophysisc Group. UC San Diego Team Aims to Broaden Researcher Access to Protein Simulation. `http://ucsdnews.ucsd.edu/pressrelease/uc_san_diego_team_aims_to_broaden_researcher_access_to_protein_simulation`, 2012.

[6] NVIDIA Corporation. CUDA C Programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2015.

[7] Mohammad Abouali. A high performance gPU implementation of Surface Energy Balance System (SEBS) based on CUDA-C. *Environmental Modelling & Software*, 41:134–138, 2013.

[8] NVIDIA Corporation. CUDA Spotlights: Compute the cure. `http://www.nvidia.com/content/cuda/spotlights/compute-cure.html`, 2015.

[9] NVIDIA Corporation. CUDA Spotlights: Diego Rivera. `http://www.nvidia.com/content/cuda/spotlights/diego-rivera-hologic.html`, 2015.

[10] NVIDIA Corporation. CUDA Spotlights: GPU-Accelerated Multi-Phase Flow Simulations. `http://www.nvidia.com/content/cuda/spotlights/gpu-accelerated-multi-phase-flow-simulations.html`, 2015.

[11] NVIDIA Corporation. CUDA Spotlights: Pete Willemsen. `http://www.nvidia.com/content/cuda/spotlights/pete-willemsen-uminn-duluth.html`, 2015.

[12] Sandia National Laboratories. LAMMPS Molecular Dynamics Simulator. `http://lammps.sandia.gov/`, 2015.

[13] Theoretical and Computational Biophysisc Group. NAMD Scalable Molecular Dynamics. `http://www.ks.uiuc.edu/Research/namd/`, 2015.

[14] NVIDIA Corporation. CUDA Samples. `hhttp://docs.nvidia.com/cuda/cudasamples/3h747XoDG`, 2015.

[15] Stephen Bartlett. Comparison of parallel programming paradigms. `http://www.cs.bath.ac.uk/mdv/courses/CM30082/projects.bho/2009-10/bartlettsjb-dissertation-2009-10.pdf`, 2010.

[16] Michael P. Allen. Introduction to molecular dynamics simulation. `http://udel.edu/_arthij/MD.pdf`, 2004.

[17] Tamar Schlick. Molecular Modeling and Simulation: An Interdisciplinary Guide, 2009.

[18] Tim Sirk. Numerical simulation of nanoscale fow: A molecular dynamics study of drag. master's thesis. Master's thesis, Virginia Polytechnic Institute and State University, 2006.

[19] Venancio Alejandro Gómez Jiménez. Determinación de temperaturas asociadas a transiciones de fase y la función de distribución radial para el argón. `http://www.unac.edu.pe/documentos/organizacion/vri/cdcitra/Informes_Finales_Investigacion=Febrero2012`, 2012.

[20] Aleida Josefina Bermúdez Di Lorenzo. *Agua líquida sobreenfriada: Un estudio de simulación con dinámica molecular*. PhD thesis, Universidad Nacional de Córdova, 2014.

[21] Julios Jellinek. Solid-liquid phase changes in simulated isoenergetic ar_{13}. *J. Chem. Phys.*, 84:2783–2794, 1985.

[22] Fabio Antonio Avellaneda Pachón. Computación de alto desempeño aplicada en técnicas de simulación de proteínas. Master's thesis, Pontificia Universidad Javeriana, 2009.

[23] John L. Klepeis. Long-timescale molecular dynamics simulations of protein structure and function. *Current Opinion in Structural Biology*, 19:120–127, 2009.

[24] John Michalakes. GPU aceleration of numerical weather prediction. *Parallel Processing Letters*, 18:531–548, 2008.

[25] Lilia Maliar. Assessing gains from parallel computation on supercomputers. Technical report, Instituto Valenciano de Investigaciones Económicas, 2013.

[26] Universidad de la República Uruguay Facultad de Ingeniería. Computación de alta performance. `http://www.fing.edu.uy/inco/cursos/hpc/material/clases/Clase2-2009.pdf`, 2009.

[27] NVIDIA Corporation. CUDA programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/3lZl7umwx`, 2015.

[28] NVIDIA Corporation. CUDA Programación paralela facilitada. `http://la.nvidia.com/object/cuda_home_new_la.html`, 2015.

[29] Abrahan Dopazo Garca Boris Caballero Lenza. ATI vs CUDA, ventajas y desventajas. `http://sabia.tic.udc.es/gc/trabajos`, 2015.

[30] Melgar Nilo Barrantes. Transiciones de fase y la función de distribución radial para el argón usando dinámica molecular. *Revista de la Facultad de Ciencias de la Univesidad Nacional de Ingeniería REVCIUNI*, 12:59–64, 2008.