Procesamiento Masivamente Paralelo de Aplicaciones Científicas en Arquitecturas Híbridas Soportado por Multi-GPU

Mónica Liliana Hernández Ariza

Universidad Industrial de Santander
Facultad de Ingenierías Fisico-mecánicas
Escuela de Ingeniería de Sistemas e Informática
Bucaramanga
2016

Procesamiento Masivamente Paralelo de Aplicaciones Científicas en Arquitecturas Híbridas Soportado por Multi-GPU

Mónica Liliana Hernández Ariza

Trabajo de Investigación para optar por el título de:

Magíster en Ingeniería de Sistemas e Informática

Director:
Ph.D Carlos Jaime Barrios Hernández
Codirector
Ph.D. Bruno Raffin

Universidad Industrial de Santander
Facultad de Ingenierías Fisico-mecánicas
Escuela de Ingeniería de Sistemas e Informática
Bucaramanga
2016

DEDICATORIA

A ti madre, quien has sido mi ejemplo de perseverancia y fortaleza.

AGRADECIMIENTOS

Los resultados de los experimentos presentados en esta publicación, fueron obtenidos usando la plataforma GridUIS-2, desarrollada por el Centro de Supercomputación y Cálculo Científico de la Universidad Industrial de Santander (SC3UIS). Esta acción es soportada por la Vicerrectoría de Investigación y Extensión de la UIS (VIE-UIS) y diferentes grupos de investigación de la universidad. (http://www.sc3.uis.edu.co)

CONTENIDO

Pág.

RESUMEN

TÍTULO: PROCESAMIENTO MASIVAMENTE PARALELO DE APLICACIONES CIENTÍFICAS EN ARQUITECTURAS HÍBRIDAS SOPORTADO POR MULTI-GPU[1]

AUTOR: Mónica Liliana Hernández Ariza[2]

PALABRAS CLAVE: FLOWVR; UNIDADES DE PROCESAMIENTO GRÁFICO GPU; GROMACS; ANÁLISIS IN SITU

DESCRIPCIÓN: Simulaciones numéricas usando supercomputadores están produciendo un creciente volumen de datos. La producción y análisis eficiente de los datos son clave para futuros descubrimientos. El paradigma in situ emerge como una solución prometedora para evitar el cuello de botella en los dispositivos E/S que se genera en el sistema de archivos por la simulación y el análisis. El principio es procesar los datos tan cerca como sea posible donde y cuando estos son producidos.

Varias estrategias e implementaciones han sido propuestas en los últimos años para soportar procesamiento in situ con un bajo impacto en el rendimiento de la simulación. Aún así, pocos esfuerzos se han hecho en el escenario de procesamiento de análisis in situ con aplicaciones híbridas que soporten aceleradores como GPUs.

En éste trabajo, se propone un estudio de las estrategias in situ usando Gromacs, un paquete de simulación de dinámica molecular con soporte multi-GPU, como aplicación de prueba. Dicho estudio se enfoca en el uso de recursos computacionales de la máquina por la simulación y el análisis in situ. Para finalizar, las estrategias de ubicación in situ son extendidas al caso de análisis in situ ejecutado en la GPU y se estudia su impacto en el rendimiento de Gromacs y la utilización de recursos. En particular se muestra que la ejecución de análisis in situ en GPU puede ser una solución más eficiente que en la CPU especialmente cuando la CPU es el cuello de botella de la simulación.

[1]Trabajo de Investigación de Maestría.

[2]Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingeniería de Sistemas e Informática. Director: Carlos Jaime Barrios Hernández, Doctor en Informática. Codirector: Bruno Raffin. Doctor en Informática.

ABSTRACT

TITLE: MASSIVE PARALLEL PROCESSING OF SCIENTIFIC APPLICATIONS IN HYBRID ARCHITECTURES SUPPORTED BY MULTIGPU[3]

AUTHOR: Mónica Liliana Hernández Ariza[4]

KEYWORDS: FLOWVR; GRAPHICS PROCESSING UNITS; GROMACS; IN SITU ANALYSIS

DESCRIPTION: Numerical simulations using supercomputers are producing an ever growing amount of data. Efficient production and analysis of these data are the key to future discoveries. The in situ paradigm is emerging as a promising solution to avoid the I/O bottleneck encountered in the file system for both the simulation and the analytics by treating the data as soon as they are produced in memory.

Various strategies and implementations have been proposed in the last years to support in situ treatments with a low impact on the simulation performance. Yet, little efforts have been made when it comes to perform in situ analytics with hybrid simulations supporting accelerators like GPUs.

In this project, a study of the in situ strategies with Gromacs is proposed, a molecular dynamic simulation code supporting multi-GPUs, as the application target. The study is specifically focused on the computational resources usage of the machine by the simulation and the in situ analytics. Finally the usual in situ placement strategies are extended to the case of in situ analytics running on a GPU and their impact on both Gromacs performance and the resource usage of the machine is studied. In particular it is shown that running in situ analytics on the GPU can be a more efficient solution than on the CPU especially when the CPU is the bottleneck of the simulation.

[3]Master's degree research work.

[4]Department of Physical-Mechanical Engineering. School of Systems Engineering and Computer Science. Advisor: Carlos Jaime Barrios Hernández, Ph.D. in Computer Science. Co-advisor: Bruno Raffin, Ph.D. in Computer Science.

INTRODUCCIÓN

Flujos de trabajo científicos en Exascale: Un cambio hacia aceleradores e in situ

Las simulaciones a gran escala son una importante herramienta para los científicos en diferentes dominios como biología, dinámica de fluidos o astrofísica. Analizar el creciente volumen de datos producido por estas simulaciones es cada vez más exigente. Este es el caso de GTC [12], una simulación de turbulencia que en 2010, produjo 260GB de datos cada dos minutos utilizando 16384 núcleos de procesamiento [33]. Recientemente, el modelo completo de la cápside del VIH fue determinado [31], para lo cual se requirió varias simulaciones que produjeron alrededor de 50TB de datos cada una, para un total de 1PB. En la era Exascale[5], se estima que menos del 1 % de los datos producidos por las simulaciones serán guardados en disco debido a restricciones de ancho de banda [18]. Por lo tanto, escribir datos en bruto en disco no será viable por la consecuente pérdida de información. La brecha entre computación y el ancho de banda de E/S en la era Exascale representa un cambio importante tanto para las simulaciones como para flujos de trabajo científicos.

Otro cambio importante es la integración de aceleradores como GPUs o procesadores Xeon PHI en los supercomputadores. Esta tendencia ya ha sido implementada por máquinas como *Tianhe-2* y *BlueWaters*. La futura máquina *Summit* del *Oak Ridge National Laboratory* integrará GPUs en su arquitectura y se espera que alcance un rendimiento de más de 150 PetaFlops[6]. Estos aceleradores ofrecen una alta tasa de Flops por vatio que se requiere para alcanzar Exascale. Códigos como NAMD [19] o Gromacs [10] han sido adaptados para aprovechar estos aceleradores y ganar una aceleración significativa. Sin embargo, usar de manera eficiente recursos computacionales compuestos por CPUs y aceleradores es aún más complejo para las simulaciones.

Una importante discusión que se da actualmente es la subutilización de recursos de cómputo en máquinas híbridas, lo cual se presenta principalmente por tres razones: 1) en la mayoría de las simulaciones, no todos los cálculos son realizados en los aceleradores. En consecuencia, hay periodos de ocio durante la ejecución de la simulación, 2) las simulaciones no pueden usar y escalar completamente con todos los recursos computacionales disponibles en un supercomputador [34] y 3) la optimización de los algoritmos es dificil de lograr puesto que la arquitectura de las GPUs cambia notablemente entre una generación y otra.

[5] Una máquina Exascale es capaz de computar 10^{18} flops (operaciones de coma flotante por segundo)
[6] 1 petaflops = 10^{15} flops

El paradigma in situ es una solución prometedora para tratar esta problemática. El principio es procesar los datos tan cerca como sea posible a su fuente mientras los datos aún residen en memória [30]. De esta manera, tanto la simulación como el análisis se benefician de este enfoque pues no se requiere escritura/lectura de datos hacia/desde el sistemas de archivos. Aunque el procesamiento in situ estuvo motivado inicialmente para E/S, se ha extendido a numerosas aplicaciones: visualización en vivo, generación de estadísticas, monitoreo de simulaciones, etc. Sin embargo, la configuración de este tipo de procesamiento implica un mayor grado de complejidad. Debido a que tanto la simulación con el análisis se ejecutan de manera concurrente, la contención en los recursos de cómputo y red puede generar una degradación significativa en el rendimiento de la simulación. En los últimos años, diferentes estrategias y sistemas han sido propuestos para mitigar esta penalidad. Por ejemplo, ejecutar el código GTC en 512 núcleos de procesamiento usando 25 % menos de núcleos CPU, reduce el rendimiento de la simulación tan sólo en 2.7 % comparado con usar todos los núcleos disponibles [35]. Para estos casos, puede ser más eficiente usar 25 % de los núcleos para ejecutar procesamiento in situ y de esta manera acelerar la fase de análisis.

Implementación de estrategias in situ comunes para simulaciones híbridas

Se han hecho esfuerzos significativos para proponer sistemas in situ con un bajo impacto en el rendimiento de la simulación. Sin embargo, la mayoría están enfocados en la ejecución de simulación y análisis solamente en la CPU [34, 3, 6, 28].

Simulaciones híbridas como Gromacs, están basadas en un balance de carga entre los cálculos realizados en la CPU y la GPU. Por tanto, la asignación de cómputo extra en la CPU puede afectar dicho balance y tener un impacto negativo en el rendimiento de la simulación.

Contribuciones

Este trabajo tiene dos contribuciones que se exponen brevemente a continiación:

1. El estúdio de las actuales estrategias in situ aplicadas a simulaciones híbridas, utilizando Gromacs (un paquete de simulación de dinámica molecular bien establecido con soporte para multi-GPU) como simulación de prueba. Primero se estudia la utilización de recursos computacionales de la versión nativa de Gromacs

durante ejecuciones clásicas. Luego, se estudia la utilización de recursos cuando se ejecuta análisis in situ en la CPU usando las estrategias *asynchronous time-partitioning* y *helper core*. Se muestra que a) la CPU es el cuello de botella de la simulación y b) asignar cómputo extra en la CPU, reduce tanto el rendimiento de la simulación como la utilización de la GPU.

2. La adaptación de las estrategias *asynchronous time-partitioning* y *helper core* para la ejecución de análisis in situ en la GPU. Se analiza la utilización de recursos y se muestra en particular que la ejecución de análisis in situ en la GPU puede ser una solución más eficiente que en la CPU, especialmente cuando la CPU es el cuello de botella de la simulación.

Presentación del documento

El presente documento está organizado de la siguiente manera: Primero se presenta en estado del arte en la sección 2. Se describe el algoritmo de Gromacs en la sección 3. La sección 4 se presenta FlowVR, la solución para gestionar procesamiento in situ en arquitecturas híbridas. En la sección 5 se describe el framework diseñado para conducir los experimentos. Estos experimentos se extienden a un caso de estudio empleando una aplicación in situ real en la sección 6. Finalmente, se concluye el trabajo de investigación.

1. OBJETIVOS

1.1 Objetivo General

Proponer mecanismos de implementación para una aplicación científica que soporten el procesamiento masivamente paralelo en arquitecturas computacionales híbridas basado en aceleración por Multi-GPU.

1.2 Objetivos Específicos

- Analizar y extraer las oportunidades de concurrencia y las posibilidades de explotación del paralelismo de la estructura del algoritmo en una aplicación científica seleccionada, para caracterizar el tipo de paralelismo y granularidad que permita la aceleración eficiente de procesamiento en arquitecturas híbridas basadas en multi-GPU.
- Seleccionar el modelo de programación y definir los mecanismos de implementación que usen eficientemente la infraestructura.
- Observar aspectos arquitecturales a nivel de plataforma en tiempo de ejecución tales como gestión de memoria, tiempos de comunicación (entre procesos), mapeo y ciclos de procesos, que permitan un trazado en términos de evaluación de rendimiento.
- Realizar una validación de la coherencia de los resultados y la representación de los mismos mediante herramientas de visualización, que permitan confrontar los resultados obtenidos.

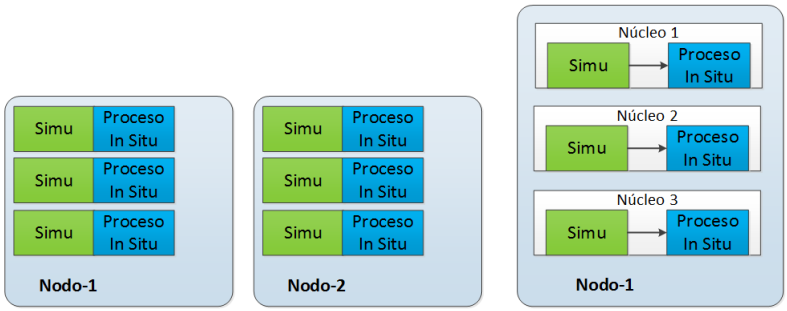
2. ESTADO DEL ARTE

En este capítulo se introducen fundamentos teóricos de procesamiento in situ y cómputo empleando GPUs. Se presentan sistemas y estrategias propuestos para procesamiento in situ. Algunos casos de procesamiento con GPUs en el contexto de in situ y *offline processing* son descritos.

2.1 Sistemas in situ

La ubicación del procesamiento de análisis es clave para el diseño de sistemas in situ. El enfoque más directo es alojar tanto la simulación como el análisis en los mismos recursos computacionales. Esta estrategia, denominada *time-partitioning*, es usualmente la más fácil de implementar y permite compartir estructuras de datos entre la simulación y el análisis, reduciendo el consumo de memoria. La Figura 1(a) representa el enfoque *time-partitioning* síncrono. Trabajos como Ma et al. [30] han implementado *time-partitioning* para integrar un motor de renderizado de volumen directamente en el código de una simulación de turbulencia. Cerca del 10 % del tiempo total de ejecución fue empleado en la renderización. Algunas herramientas de visualización como Visit [29] o Paraview [7] cuentan con librerías ligeras para instrumentar la simulación. El principal objetivo es convertir el formato de datos de la simulación a formato VTK antes de ejecutar una secuencia de análisis. Una integración completa entre una simulación de sismos y el framework Hercules para ejecutar visualización in situ fue propuesta por Tu et al. [27]. La única salida es un conjunto de imágenes en formato JPEG.

Para todos estos sistemas, el tiempo empleado para la ejecución del análisis es directamente adicionado al tiempo de simulación. Por tanto, este enfoque puede ser muy costoso tanto en tiempo como en consumo de memoria. Un estudio ha sido propuesto con Catalyst [17] en simulaciones industriales. Con escenarios de análisis comúnmente usados, se observó un incremento de hasta un 30 % en el tiempo de ejecución y hasta un 300 % de incremento en el consumo de memoria debido a requerimientos de conversión de datos. Goldrush [34] aborda el problema de tiempo global de ejecución haciendo procesamiento asíncrono representado en la Figura 1(b). Para limitar el impacto del procesamiento asíncrono sobre el tiempo de ejecución de la simulación, la ejecución de dicho procesamiento se calendariza para cuando la simulación no está haciendo uso de todos los núcleos de procesamiento disponibles (fuera de una sección OpenMP). El objetivo es mejorar el uso global de recursos en la maquina calendarizando la ejecución del análisis cuando los recursos son subutilizados.



(a) (b)

Figura 1: Estrategia *time-partitioning*: Procesamiento in situ (a) síncrono y (b) asíncrono.

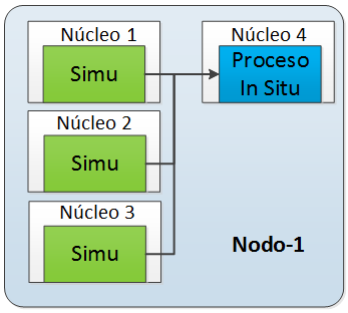


Figura 2: Estrategia *space-partitioning*: Enfoque *helper core*.

Otros trabajos proponen el uso de recursos dedicados para ejecutar el procesamiento in situ. Este enfoque, denominado *space-partitioning*, permite la ejecución asíncrona de análisis in situ evitando algunas contenciones en los recursos computacionales, sin embargo requiere al menos una copia de datos. Sistemas como Damaris [3] usan núcleos de procesamiento dedicados -denominados *helper cores*- en cada nodo de la simulación para ejecutar de manera asíncrona el análisis in situ. El enfoque *helper core* se muestra en la Figura 2. Los datos son copiados desde la simulación hacia un espacio de memoria compartida. De esta forma, el análisis puede leer y procesar los datos de este espacio de manera asíncrona. Análisis como E/S o visualización científica con Visit [4] son ahora posibles gracias a este enfoque. La estrategia *helper core* ha sido también implementada por *Functional Partitioning* [15] y GePSeA [21] enfocado principalmente en operaciones de E/S.

Otros trabajos proponen el uso de nodos dedicados, denominados *staging nodes* para ejecutar el análisis como se muestra en la Figura 3. Este enfoque se denomina análisis in transit. PreData [33] está construido dentro del framework ADIOS [16] y permite la ejecución in situ de operaciones ligeras antes de mover los datos hacia *staging nodes*.

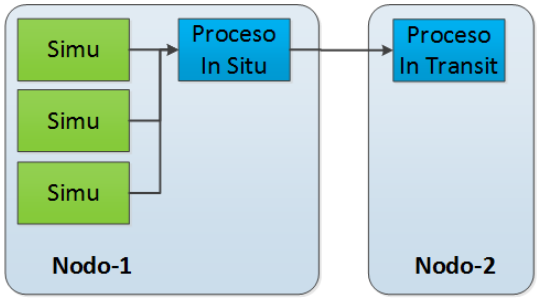


Figura 3: Estrategia *space-partitioning*: Enfoque *staging nodes*.

Los datos son procesados in transit usando un modelo Map/Reduce. DataTap es usado para calendarizar transferencia de datos cuando la simulación está en una fase de computación y no está haciendo uso extensivo de la tarjeta de red. HDF5/DMS [22] usa la interfaz HDF5 para captura de datos y posterior almacenamiento en un espacio de memoria distribuida compartida. Otras aplicaciones pueden entonces leer los datos con el API de lectura HDF5 usualmente en un conjunto de nodos diferente. DataSpaces [2] implementa un sistema de publicación/subscripción distribuido. La simulación inserta los datos en un espacio distribuido indexado y otros procesos recuperan los datos necesarios.

Recientemente, sistemas híbridos combinando procesamiento in situ (síncrono o asíncrono) e in transit han surgido. Fheng et al. [32] resaltan la necesidad de flexibilidad en la ubicación del análisis y proponen un modelo analítico para evaluar el costo de las estrategias de ubicación. Glean [28] permite procesamiento síncrono in situ y procesamiento asíncrono in transit. FlexIO [35] está construido sobre ADIOS y permite procesamiento asíncrono in situ en núcleos dedicados y procesamiento asíncrono in transit. El sistema monitorea el rendimiento de la simulación y puede migrar el análisis en tiempo de ejecución o desacelerarlo si éste impacta demasiado el rendimiento de la simulación. FlowVR [6, 1] permite describir el flujo de datos entre componentes (simulación, módulos de análisis). Los procesamientos son ejecutados de manera asíncrona a la simulación. El usuario puede especificar la ubicación del procesamiento: en un conjunto de núcleos dedicados, nodos dedicados, o en los mismos recursos de la simulación.

El presente proyecto sigue el trabajo hecho en el enfoque *helper core* y los enfoques de *time-partitioning* asíncrono. Estos trabajos son extendidos al dominio de simulaciones multi-GPU y análisis in situ usando GPUs. Para implementar el enfoque presentado en este proyecto, se utiliza el middleware FlowVR para alojar y coordinar la ejecución de análisis in situ.

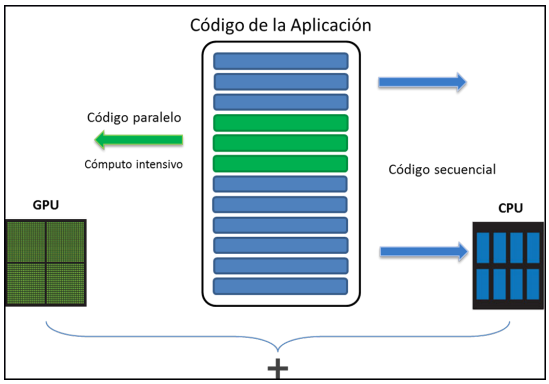


Figura 4: Flujo de ejecución de una aplicación en una arquitectura híbrida CPU-GPU.[2]

2.2 Arquitecturas Híbridas CPU-GPU

Algunos de los supercomputadores más potentes a nivel mundial según la lista top500[1] incorporan GPUs en su arquitectura, una tendencia que será implementada por futuras máquinas paralelas. Estas máquinas basadas en GPUs son un sistema híbrido en el cual la GPU se utiliza como un coprocesador, en el que se ejecutan aquellas secciones de una aplicación con mayor carga computacional. El resto de la aplicación se ejecuta en la CPU como se muestra en la Figura 4. La combinación de estos recursos se hace con el fin de acelerar las aplicaciones que van desde aplicaciones para empresas hasta cálculo científico.

La principal diferencia entre la CPU y la GPU es la cantidad de unidades de procesamiento. El diseño de la CPU se basa en unas cuantas unidades de procesamiento y varios niveles de caché. Estas características permiten una baja latencia necesaria para tareas con poca tolerancia a latencia. La GPU, por otra parte, tiene más ALUs (Unidad Aritmético Lógica) por área comparado con la CPU y menor caché. Por tanto, ejecuta muchos más hilos de procesamiento en paralelo, lo que permite un mayor rendimiento comparado con la CPU a costo de una alta latencia (ver Figura 5).

La CPU se basa en un diseño de paralelismo de tareas. Múltiples tareas se mapean a múltiples hilos y cada tarea ejecuta diferentes instrucciones. Típicamente, estas tareas son de grano grueso y los hilos de cómputo se gestionan de manera individual. La GPU se basa en paralelismo de datos, bajo el modelo SIMD (*Single Instruction Multiple Data*). Las tareas son de grano fino y los hilos de cómputo se gestionan por

[1] http://www.top500.org/lists/2015/06/
[2] Fuente: http://www.nvidia.com/object/what-is-gpu-computing.html
[3] Fuente: http://www.nvidia.com/object/what-is-gpu-computing.html

Figura 5: CPU vs GPU: Unidades de procesamiento y caché.[3]

hardware. La GPU puede soportar el mapeo de miles de hilos sobre cientos de núcleos de procesamiento. Estos hilos se ejecutan de manera concurrente. Para obtener el mejor rendimiento posible, es necesario seleccionar el dispositivo CPU o GPU más apropiado para ejecutar tareas específicas. Tareas con poco nivel de paralelismo, estructuras de datos complejas y/o frecuentes accesos a memoria generalmente resultan más apropiadas de tratar en la CPU. La GPU resulta más apropiada para ejecutar tareas con un alto grado de paralelismo y poco acceso a memoria. Para hacer uso de la GPU, los algoritmos deben ser modificados antes de ser portados.

Si bien arquitecturas híbridas CPU-GPU permiten acelerar las aplicaciones, hay desafíos a tratar. Al utilizar la GPU como coprocesador, se requiere realizar transferencia de datos entre la memoria RAM y la memoria de la GPU. Operaciones de transferencia pueden ser costosas tanto en tiempo como en consumo de memoria. Por otra parte, para garantizar un mejor uso de los recursos disponibles en la máquina se requiere de cómputo concurrente en la CPU y la GPU. Idealmente, las aplicaciones deben minimizar el tiempo en que la CPU está esperando por resultados de la GPU o viceversa. En cuyo caso, el cómputo entre la CPU y la GPU está bien balanceado (ver Figura 6). Sin embargo, este balance no es fácil de lograr y es específico a cada aplicación, lo que puede conllevar a subutilización de recursos o degradación en el rendimiento de la aplicación.

El imbalance en una aplicación híbrida se puede presentar cuando hay concurrencia parcial (ver Figura 7(a)) o ausencia de concurrencia (ver Figura 7(b)). La concurrencia parcial se presenta cuando el cómputo en la GPU finaliza primero que el de la CPU o viceversa. Cuando hay ausencia de concurrencia, los periodos de cómputo de la CPU y la GPU se ejecutan de manera serial. En estos casos, hay periodos de ocio en los que los recursos disponibles en la máquina son subutilizados por la aplicación.

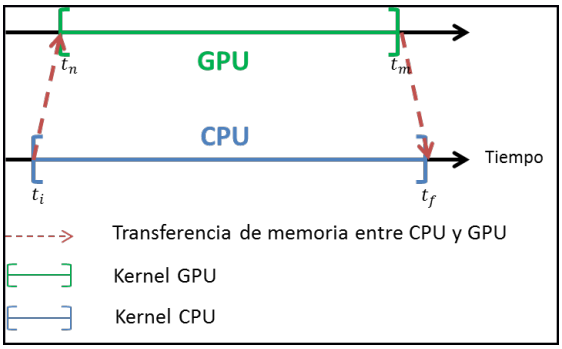
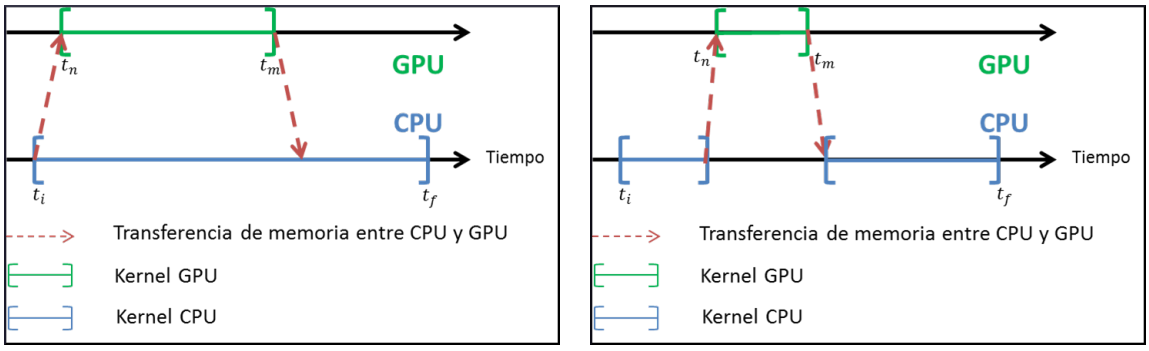


Figura 6: Aplicaciones híbridas: balance de cómputo CPU-GPU.



(a) (b)

Figura 7: Imbalance en aplicaciones híbridas: (a) concurrencia parcial y (b) ausencia de concurrencia.

2.3 Procesamiento con GPUs

Una implementación de un sistema in situ usando GPUs es el trabajo presentado por R. Hagan et al. [9] en el que se propone un método de balance de carga para visualización in situ en un sistema multi-GPU. Este método se basa en una estrategia asíncrona de espacio compartido donde N/2 GPUs son usadas como GPUs dedicadas para visualización, siendo N el número de GPUs disponibles en el sistema. Las restantes N/2 GPUs ejecutan la simulación de N-cuerpos y transfieren los datos procesados a la RAM. Una vez en memoria, los datos son transferidos a una GPU dedicada para ejecutar tareas de renderización a través de un algoritmo de visualización de trazado de rayos. Cada GPU es gestionada por buffers separados en la CPU a fin de escribir/leer los datos hacia/desde memoria de manera asíncrona.

Realizar procesamiento *offline* en GPUs es un creciente campo de interés. VMD [11] es una herramienta ampliamente usada para visualización y análisis de sistemas biológicos

tales como proteinas y ácidos nucléicos. Durante los últimos años, diversas visualizaciones y análisis han sido adaptados para soportar GPUs usando CUDA. El algoritmo Quicksurf [13], por ejemplo, ha sido propuesto para visualizar superficies moleculares de grandes conjuntos de átomos. Recientemente fue usado para visualizar el modelo completo de la cápside del VIH usando VMD [11] en el supercomputador *BlueWaters* [26]. Otros análisis tales como funciones de distribución radiales [14], *fitting* [25], entre otros [23], son acelerados con GPUs. Aunque VMD no cuenta con un soporte completo para análisis in situ, algunas aplicaciones interactivas que combinan simulación y visualización en vivo son posibles como lo es *Interactive Molecular Dynamic simulations* [24].

3. ANÁLISIS DEL ALGORITMO DE GROMACS

El código de simulación utilizado para el desarrollo de este trabajo es Gromacs, un paquete para simulación de dinámica molecular ampliamente usado. Esta selección se basa principalmente en tres razones: a) Gromacs es uno de los paquete estandar para simulación de dinámica molecular. b) Gromacs cuenta con soporte para arquitecturas híbridas multi-GPU, apropiado para el estudio de balance de cómputo CPU/GPU y utilización de recursos. c) Este trabajo hace parte de una colaboración con MOAIS, grupo de investigación del instituto INRIA (*Institut National de Recherche en Informatique et en Automatique*). MOAIS puede proporcionar escenarios reales basados en las necesidades de la comunidad de biología. En éste capítulo se describe Gromacs, su funcionamiento y la posible mejora que se puede proponer con éste trabajo y métodos de procesamiento in situ.

3.1 Descripción de Gromacs

Gromacs [10, 20] es un código de simulación de dinámica molecular con un paralelismo híbrido MPI/OpenMP/GPU. Los átomos simulados son distribuidos en una malla irregular donde cada celda de la malla es gestionada por un proceso MPI (ver Figura 8).

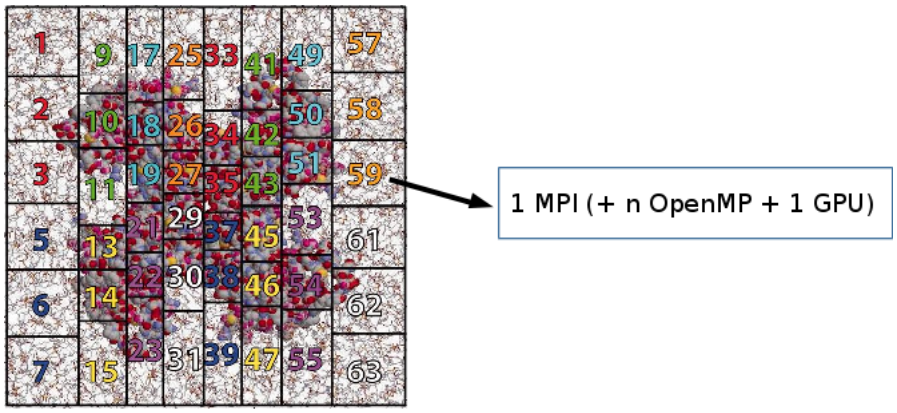


Figura 8: Átomos de la molécula distribuidos en una malla irregular.

La organización interna de Gromacs se basa en un modelo maestro/esclavo. El proceso maestro (rango 0) se encarga de la distribución inicial de los átomos del sistema, de calcular los valores totales del sistema tales como el nivel de energía o temperatura, así como todas las operaciones de escritura. Durante la fase inicial, el proceso maestro distribuye los átomos a los procesos esclavo. Durante una iteración de escritura, la posición, velocidad y fuerza de los átomos son recopiladas de manera síncrona por el proceso maestro y guardadas en disco. Las operaciones de distribución y recopilación

tienen un costo en el rendimiento de la simulación: bloquean la simulación y no escalan apropiadamente para un número grande de procesos. Sin embargo, Gromacs usa estas operaciones a una frecuencia relativamente baja, lo que a su vez implica que la escritura de archivos de salida empleando estas operaciones debe ser limitada. Típicamente, para simulaciones extensas, sólo una iteración entre 1000 a 5000 iteraciones es guardada en disco.

A medida que la simulación avanza, los átomos pueden desplazarse y cambiar de celda. Es por ello que se necesitan comunicaciones entre procesos para intercambiar las posiciones de los átomos a lo largo de las celdas de cada proceso. Cada N iteraciones, una fase de cálculo de vecindades se ejecuta. Durante esta fase, los átomos huéspedes pueden cambiar de procesos y pueden surgir áreas fantasma. Gromacs cuenta con un sistema de balanceo de carga dinámico que ajusta las dimensiones de la malla en tiempo de ejecución. El objetivo es equilibrar lo más posible el tiempo empleado para cada proceso en el cálculo de las fuerzas que representa la mayor carga computacional.

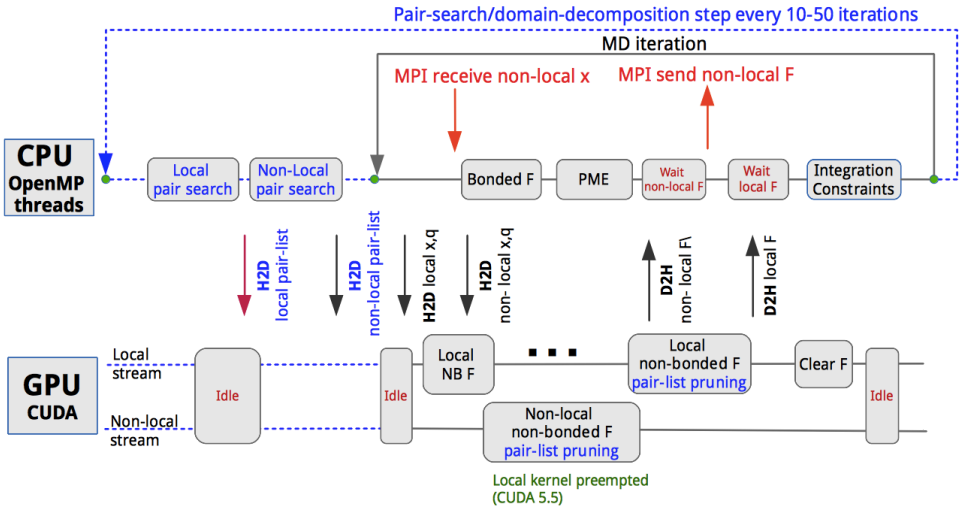


Figura 9: Gromacs: Flujo de cómputo concurrente entre la CPU y la GPU.[1]

Gromacs puede usar múltiples GPUs por nodo durante el cómputo de procesos. Cada GPU es asignada a un proceso MPI. En caso de que el número de núcleos en la CPU sea mayor que el número de GPUs en el nodo, los núcleos restantes son gestionados con OpenMP para acelerar el cómputo en la CPU. La computación más intensiva de Gromacs es la de las fuerzas: interacciones enlazadas y no enlazadas. Las interacciones enlazadas son aquellas entre átomos que comparten un enlace y se ejecutan en la CPU.

[1]Fuente: http://www.gromacs.org/GPU_acceleration

Las interacciones no enlazadas son entre átomos distantes, típicamente requieren mayor cómputo y por tanto se ejecutan en la GPU. Estos dos tipos de interacciones se ejecutan de manera concurrente (ver Figura 9). El sistema de balanceo de carga dinámico también se encarga de monitorear la diferencia en el tiempo de cómputo entre la CPU y la GPU y ajustar las dimensiones de la malla consecuentemente.

Si bien el cómputo de fuerzas ejecutado por la CPU y la GPU se realiza de manera concurrente, la CPU se encarga de realizar cómputos adicionales. Por tanto, es de esperarse que haya periodos de ocio para la GPU durante la simulación. Un problema potencial adicional es que el sistema de balanceo de carga podría no ser capaz de balancear apropiadamente el cómputo entre la CPU y la GPU.

Estas observaciones indican que Gromacs no estaría aprovechando completamente la GPU. Una serie de experimentos fueron diseñados para evaluar la utilización real de la GPU por Gromacs. Estos experimentos también permitirán hacer un análisis del balance CPU-GPU e identificar factores que pueden afectar dicho balance.

3.2 Contexto Experimental

Los experimentos se realizaron en el cluster GUANE-1 (GpUs Advanced eNviromEnt) de la Universidad Industrial de Santander. Cada nodo cuenta con 8 núcleos de procesamiento Intel® Xeon® CPU E5640 @ 2.67 GHz, distribuidos en 2 zócalos con 4 núcleos cada uno y *hyper-threading* activado (16 núcleos lógicos), 103 GB de RAM y 8 Nvidia Tesla M2050 GPUs de 448 núcleos GPU cada una. La comunicación entre procesos se realiza a través de una red Ethernet de 1GB. Para todos los experimentos Gromacs ejecuta una simulación de MARTINI (simulación de agregación atómica) con un conjunto de 54000 lípidos que representa alrededor de 2,1 millones de partículas de grano grueso [8]. Gromacs es muy sensible a la calidad de la red debido a su alta frecuencia. Por lo tanto, se utilizó solo un nodo para ejecutar los experimentos y así evitar la comunicación entre nodos. Para todos los experimentos, el método nativo de escritura de Gromacs fue deshabilitado.

Las medidas consideradas para cada experimento son el rendimiento de la simulación y la utilización de la GPU. La métrica de rendimiento es iteraciones por segundo (mayor es mejor). La duración de cada experimento es de por lo menos 1000 pasos iterativos para evitar la inestabilidad en el rendimiento debido al balanceo dinámico de carga en la fase inicial. La utilización de la GPU es medida con la herramienta nvidia-smi[2] de Nvidia. La utilización GPU indica el porcentaje de tiempo en el pasado segundo

[2] https://developer.nvidia.com/nvidia-system-management-interface

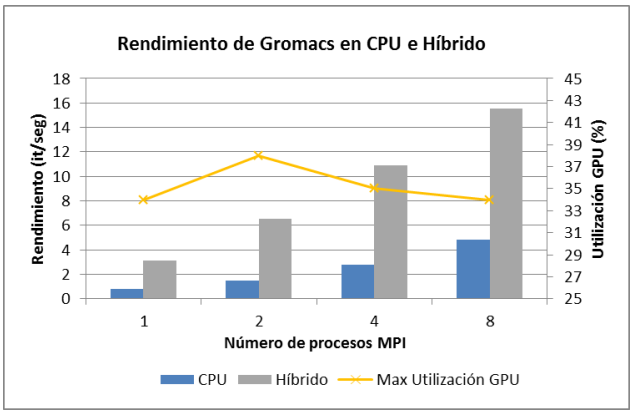


Figura 10: Rendimiento de Gromacs nativo y utilización de la GPU cuando se incrementa el número de procesos MPI

durante el cual uno o más *kernels* fueron ejecutados en cada GPU. El valor tenido en cuenta fue el máximo valor de utilización de GPU encontrado para cada experimento de un total de 5 medidas por experimento.

3.3 Resultados de Gromacs nativo

Se diseñó un conjunto de pruebas para determinar qué configuración (número de hilos, procesos MPI, GPUs) proporciona el mejor rendimiento para Gromacs nativo (sin modificaciones en el código) en un nodo. Luego se estudia la utilización de recursos para esta configuración en particular. Con estas pruebas también se busca determinar si Gromacs puede aprovechar las 8 GPUs disponibles en el nodo.

La Figura 10 presenta los resultados obtenidos para Gromacs utilizando dos configurciones: únicamente CPU y en modo híbrido usando GPUs (1 GPU por cada proceso MPI). Para ambos casos, se utilizó 2 hilos OpenMP por cada proceso MPI, con los dos hilos mapeados en el mismo núcleo de procesamiento físico (*hyperthreading*). La versión híbrida supera la versión CPU en todos los casos por un factor que va desde 3.2 para 8 procesos MPI hasta 4.38 para 2 procesos MPI.

Durante la simulación, el cómputo de la CPU y la GPU se ejecuta de manera concurrente. Este cómputo es balanceado por Gromacs en tiempo de ejecución. De acuerdo con los registros internos de Gromacs, para cada configuración híbrida, la CPU no está esperando por la GPU. Esto puede indicar que la GPU está inactiva mientras que la CPU completa su cómputo.

La utilización GPU de la Figura 10 indica el máximo porcentaje de tiempo donde al menos un kernel de CUDA está activo. En el mejor de los casos, para 2 procesos MPI, la GPU se usa a lo sumo 38 % del tiempo. Adicionalmente, cuando los kernels de la simulación están en ejecución, la ocupación de la GPU es sólo del 60 %[3].

3.4 Discusión sobre los resultados

Si bien Gromacs se beneficia significativamente del uso de múltiples GPUs, estos recursos son subutilizados por la simulación. Estos resultados indican que, en el caso de Gromacs, la CPU podría ser el factor limitante en la computación. Esta observación está soportada por los archivos internos de Gromacs, en los cuales se aprecia que la CPU no debe esperar a que la GPU termine el cómputo. Si bien esta observación es la mas probable para el caso presentado, la subutilización de recursos puede ser causada por otros factores como limitaciones de ancho de banda o deficiencia en la escalabilidad de la implementación GPU de Gromacs. Sin embargo el tamaño del modelo usado y el algoritmo seleccionado para la simulación (grano grueso) hacen estas hipótesis menos probables.

Gromacs escala con 8 GPUs con un incremento significativo en el rendimiento de la simulación. Sin embargo, el balance de carga entre la CPU y la GPU no es eficiente debido a las secciones del algoritmo que se ejecutan únicamente en la CPU. Durante estos periodos la GPU espera por el envío desde la CPU de procesamiento a realizar. Debido al imbalance de carga, la GPU es subutilizada por la simulación. Estos resultados confirman el análisis previo: En la plataforma empleada, la CPU es el cuello de botella de la simulación, causando un bajo uso de las GPUs.

La cantidad de recursos computacionales subutilizados por Gromacs es bastante significativa. Estos recursos podrían acelerar el proceso científico si se usaran de manera apropiada. Los siguientes capítulos de este trabajo se centran en proponer estrategias para mejorar la utilización global de recursos en arquitecturas híbridas y en consecuencia, acelerar el proceso científico.

El enfoque del trabajo es considerar el flujo de trabajo científico completo como aplicación, el cual está compuesto por la simulación y el análisis. El paradigma in situ permite ejecutar análisis en los mismos recursos computacionales de la simulación. De esta manera, al ejecutar tareas extra junto con la simulación, es posible obtener beneficios de recursos subutilizados por la simulación.

[3]Medido con nvprof

4. DESCRIPCIÓN DE LA SOLUCIÓN IN SITU

El paradigma de procesamiento in situ es una posible solución para mejorar la utilización global de recursos de una aplicación científica. En el contexto in situ, análisis y simulación se ejecutan en los mismo recursos computacionales. En este capítulo, se selecciona una herramienta para crear aplicaciones in situ y evaluar su costo en el rendimiento de la simulación.

4.1 *Middleware* FlowVR

FlowVR [6] es un middleware para crear aplicaciones in situ asíncronas implementando el paradigma de programación por componentes. FlowVR Permite describir una aplicación como un grafo, donde los vértices son operaciones de datos y las aristas son canales de comunicación. Un vértice se denomina módulo y está equipado con puertos de entrada y de salida que permiten el intercambio de mensajes con los demás nodos de la aplicación. Una aplicación instrumentada con FlowVR puede ser ejecutada en un nodo de cómputo o en un sistema distribuido. La coordinación de los módulos y la gestión de los canales de comunicación es responsabilidad de un proceso específico denominado demonio. Los canales de comunicación pueden pasar a través de memoria compartida para comunicación intra nodo o a través de la red para comunicación entre nodos.

4.1.1. Construcción de un módulo

Un módulo es un proceso o hilo que ejecuta un loop infinito. En cada iteración, un módulo puede recibir datos, procesarlos, y enviar los datos procesados al resto de la aplicación. Un loop está implementado con tres funciones principales:

- *wait()*: bloquea el módulo hasta que haya por lo menos un mensaje en todos los puertos de entrada.
- *get()*: devuelve el mensaje más antiguo de la cola de un puerto de entrada.
- *put()*: envía un mensaje a un puerto de salida.

Las funciones *get()* y *put()* son funciones no bloqueantes. Un módulo no tiene conocimiento del origen o destino de los datos. Los canales de datos son descritos en un script Python que declara la aplicación global y crea los enlaces entre los módulos. Cada módulo es asignado a un huésped y posiblemente a un conjunto de núcleos en el huésped. Un demonio es alojado en cada nodo en la aplicación. El demonio alberga un segmento de memoria compartida en el que los módulos leen y escriben los datos.

Si dos módulos están alojados en el mismo huésped, el demonio hace un simple intercambio de apuntadores en el espacio de memoria compartida. De los contrario, el demonio local envía el mensaje al demonio que aloja al módulo remoto, quien escribirá los datos en su espacio de memoria compartida y pasará el apuntador a su módulo.

FlowVR no impone restricciones en los recursos usados por un módulo. Cada módulo puede usar OpenMP o GPUs para acelerar sus procesamientos. Sin embargo, FlowVR no provee protección en caso de que varios módulos estén usando los mismo recursos de manera intensiva.

4.1.2. Construcción de un grafo

La declaración de un grafo se realiza en tres etapas:

1. Describir todos los módulos que se van a utilizar: La entidad básica de un grafo de FlowVR es el componente. Un componente puede incluir uno o varios módulos. Cada componente tiene definidos una serie de parametros: nombre, un lanzador (ssh, mpirun), un conjunto de nodos o máquinas que ejecutarán los módulos, así como el detalle de los módulos del componente con sus puertos.

2. Instanciar todos los módulos de la aplicación: En esta etapa se definen los parametros correspondientes a nombre, hostnames, y conjunto de núcleos de procesamiento.

3. Crear enlaces desde los puertos de entrada hacia los puertos de salida: El desarrollador tiene que enlazar los puertos de los módulos entre sí. Un canal de comunicación puede ser creado únicamente desde un puerto de salida hacia uno o varios puertos de entrada (*broadcast*).

Estas tres etapas se realizan en una secuencia de comandos de Python.

Por defecto, los canales de comunicación son FIFO (*First In First Out*). Cada puerto de entrada es asociado con una cola. Cuando un mensaje llega, es insertado en la cola del puerto correspondiente. Cuando el módulo hace un llamado a *wait()*, el mensaje en la cabeza de la cola es enviado al módulo y removido de la cola.

Para definir la política de gestión de mensajes, el desarrollador dispone de varios componentes llamados filtros y sincronizadores. Los filtros son módulos ligeros especializados para manipular mensajes en las colas. Esto permite, por ejemplo, crear políticas de muestra en las colas de mensajes.

4.2 Integración de Gromacs en el framework

El código de simulación, Gromacs, fue instrumentado con un método similar a trabajos previos [6, 5]. Para cada proceso MPI, se declaró un módulo con un puerto de salida para extraer las posiciones de los átomos. Este enfoque permite preservar el mismo nivel de paralelismo de los datos, que puede ser usado después por el análisis in situ.

El loop principal de Gromacs fue modificado para extraer periódicamente la posición de los átomos. Cada x iteraciones, siendo x un parámetro definido por el usuario, cada módulo ejecuta un *wait()* FlowVR, copia la posición de los átomos locales en un mensaje FlowVR, y coloca (*put()*) el mensaje. Las posiciones de los átomos son entonces enviadas al resto de la aplicación, si los puertos de salida de los módulos están conectados a otros módulos como módulos de análisis in situ. De lo contrario, los datos son eliminados debido a que no serán usados por ningún módulo. Los módulos de Gromacs no tienen puertos de entrada, por tanto la función *wait()* retorna inmediatamente y no bloquea la simulación. En orden de minimizar el ruido en el rendimiento de la simulación, se deshabilitó el sistema nativo de escritura de Gromacs.

4.3 Resultados de la instrumentación de Gromacs con FlowVR

Para este conjunto de pruebas se utilizó la misma configuración de la plataforma y sistema molecular descrito en la sección 3.2. Se midió el rendimiento tanto de la versión nativa de Gromacs como de la versión instrumentada con FlowVR. Para cada proceso MPI, se asignaron 2 hilos OpenMP y una GPU como se hizo previamente. Para la versión instrumentada, se extrajeron los datos cada 100 iteraciones de simulación.

Para el peor de los casos, la instrumentación incrementa el tiempo de la simulación por 0.5 % para 2 procesos MPI. El impacto en la utilización de la GPU es también despreciable. Estos resultados demuestran que el método de instrumentación presentado no impacta el comportamiento de la simulación. Este costo es significativamente menor que en resultados previos, lo cual puede ser explicado por una frecuencia de salida más baja. En los resultado previos, los datos se extraían cada 10 ms. En este caso, se extraen datos cada 6 segundos. Debido a que la instrumentación bloquea la simulación por cerca de 0.2 ms cada paso de salida (*wait()* y copia de las posiciones de los átomos), el costo de la instrumentación es despreciable.

4.4 Discusión sobre los resultados

Para la creación de la aplicación in situ, se seleccionó el middleware FlowVR, el cual permite procesamiento asíncrono in situ. Una de las ventajas de FlowVR, comparado con otras herramientas, es la flexibilidad en la ubicación del análisis, que puede ser definido por el usuario. Los recursos en los cuales se puede ubicar el análisis son en un conjunto de núcleos dedicados, nodos dedicados, o en los mismos recursos de la simulación. FlowVR permite la integración de códigos C/C++ o Python para el procesamiento in situ. Esta caracteristica permite que haya una amplia gama de códigos de análisis que pueden ser procesados in situ, sin que se deba hacer un paso extra de portabilidad de dichos códigos.

5. FRAMEWORK IN SITU PARA ARQUITECTURAS HÍBRIDAS

El procesamiento in situ de aplicaciones que se ejecutan solo en CPU permite la aceleración de las aplicaciones al mismo tiempo que mejora la utilización de recursos. En el caso de aplicaciones híbridas no es claro como el procesamiento de análisis in situ puede afectar el balance CPU-GPU de la simulación y por ende el rendimiento de la misma. En la primera parte de este capítulo se presenta el estudio del impacto de estrategias in situ comunes en la simulación. Para implementar dichas estrategias, se diseñó un framework para realizar procesamiento in situ empleando el middleware seleccionado, FlowVR. En la segunda parte del capítulo se propone una estrategia de ubicación in situ para mejorar la utilización global de recursos en el contexto de arquitecturas híbridas CPU-GPU.

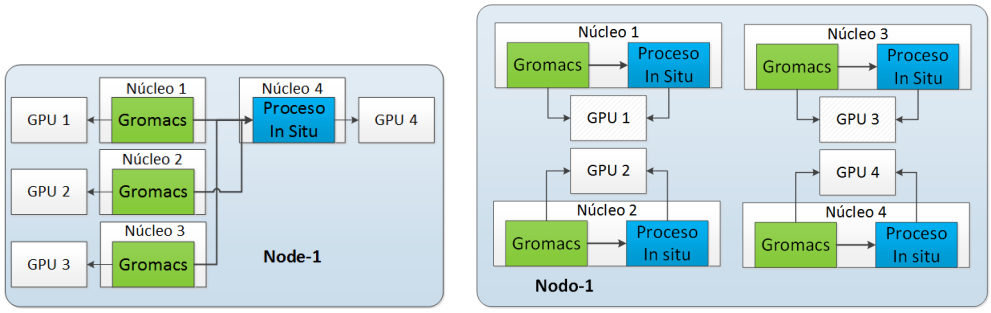
5.1 Descripción del framework

Se implementó un framework para ejecutar análisis in situ basado en dos estrategias diferentes de ubicación: *helper core* y *overlapping*. La simulación y el análisis tienen la posibilidad de usar GPUs para las dos estrategias.

La estrategia *helper core* reserva un núcleo de procesamiento por nodo de cómputo para ejecutar análisis in situ (ver Figura 11(a)). Los datos son recolectados en cada nodo de manera asíncrona y enviados a una o varias tareas in situ alojadas en el núcleo dedicado. Una GPU fue asignada para cada proceso en la simulación y una GPU para el análisis.

La estrategia *overlapping* se ejecuta en los mismos recursos de la simulación (ver Figura 11(b)). Para éste caso, el número de tareas in situ instanciadas es igual al número de procesos MPI por nodo de cómputo. Por lo tanto, cada proceso MPI de la simulación envía los datos a la tarea in situ ubicada en el mismo núcleo de procesamiento. Las tareas in situ se ejecutan de manera asíncrona a la simulación. Cada GPU es compartida entre un proceso de la simulación y una tarea in situ. En tiempo de ejecución, los *kernels* de la simulación y el análisis se ejecutan concurrentemente en cada GPU.

El análisis in situ es desplegado cada vez que la simulación realiza un paso de salida de datos. Los diferentes análisis usados son descritos es la siguiente sección. En estas dos configuraciones en particular, los canales de comunicación son canales FIFO entre los módulos de la simulación y el análisis. Si el análisis no sigue el paso de la simulación puede ocurrir desbordamiento. Para este framework, esta situación es aceptable dado



(a) (b)

Figura 11: (a) Estrategia *helper core*. Un núcleo dedicado es asignado para análisis in situ. Los datos son recopilados por el nodo y enviados de manera asíncrona a las tareas in situ. (b) Estrategia *overlapping*. Una tarea in situ es instanciada por cada proceso MPI de la simulación.

que los datos producidos son relativamente pequeños y sólo se ejecutan unos cuantos pasos de salida. Para escenarios con aplicaciones reales, componentes especiales pueden ser agregados para muestrear los datos de salida desde la simulación. También es posible bloquear la simulación en el siguiente paso de salida si el paso anterior no ha sido aún analizado.

5.2 Aplicación de estrategias in situ comunes

Se diseñó un micro benchmark para estresar la CPU y evaluar el impacto de tareas in situ CPU en el balance CPU/GPU de Gromacs. Este benchmark está disponible para las estrategias *helper core* y *overlapping*. Con estos experimentos se busca analizar el rendimiento de la simulación y la utilización de recursos cuando la simulación es perturbada por cómputo extra ejecutado en la CPU.

5.2.1. Descripción del benchmark diseñado para CPU

PI El benchmark PI, usado por Zheng et al. en [34], estresa las unidades de punto flotante de los procesadores. Cuando PI es activado, x iteraciones, siendo x un parámetro definido por el usuario, son ejecutadas para estimar el valor de π. Para las dos estrategias, *helper core* y *overlapping*, se ejecutan el mismo número de iteraciones PI. Con la estrategia *helper core*, únicamente un proceso in situ computa todas las iteraciones (x). En el caso de la estrategia *overlapping*, N procesos in situ son usados. Las x iteraciones se distribuyen de manera uniforme entre todas las tareas in situ (x/N).

Este benchmark perturba la CPU mientras la CPU y la GPU son usadas de manera intensiva por la simulación. La simulación balancea la carga computacional tanto de la CPU como de la GPU. Por lo tanto, perturbar la CPU debe afectar la computación de la CPU y la GPU de la simulación.

5.2.2. Resultados de procesamiento in situ en CPU

Se midió el rendimiento de Gromacs al ejecutar análisis in situ asíncrono en la CPU. Se usó el benchmark PI descrito en las sección 5.2.1 para las estrategias *helper core* y *overlapping*. La salida de datos de Gromacs se realizó cada 100 iteraciones de la simulación. Por cada salida de Gromacs, se desplegaron x iteraciones de PI en total. La Figura 12 muestra el rendimiento de la simulación y la utilización GPU para las dos estrategias mientras se varía x.

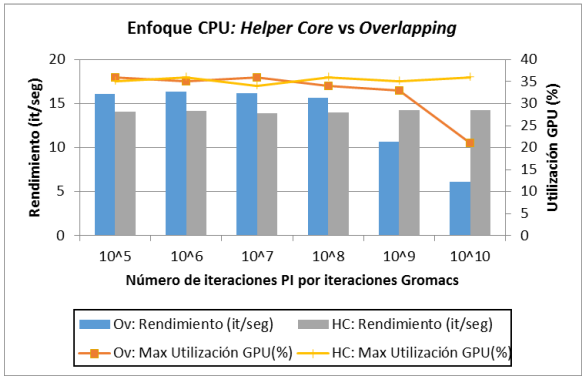


Figura 12: Estrategias *helper core* (HC) y *overlapping* (Ov) cuando se incrementa el número de iteraciones PI ejecutadas por iteración de salida de Gromacs.

Con la estrategia *overlapping* se obtiene el mejor rendimiento siempre y cuando el cómputo extra no sea intensivo. Para un número de iteraciones menor que 10^8, la simulación es desacelerada por menos del 4 % mientras que la utilización GPU permanece al mismo nivel que Gromacs nativo. Sin embargo, para un número mayor de iteraciones PI, el rendimiento de la simulación decrece a medida que x incrementa. Para 10^9 iteraciones, la degradación en el rendimiento de la simulación is superior al 30 % mientras que la utilización GPU decrece un 3 %. Para 10^{10} iteraciones, la degradación tanto del rendimiento de la simulación como la utilización GPU es aún más drástica.

La estrategia *helper core* muestra un comportamiento más estable. Se observó que el costo inicial en el rendimiento de la simulación es de 13.53 % comparado con la versión nativa de Gromacs (12.5 % de los recursos es tomado de la simulación). El

costo inicial con un número reducido de iteraciones PI es mayor que para la estrategia *overlapping*. Este resultado es esperado dado que un núcleo de procesamiento y una GPU son removidos de los recursos de la simulación. Sin embargo, como las tareas in situ no están albergadas en los mismos recursos de la simulación, el incremento en la carga computacional no afecta el rendimiento de la simulación. La Figura 12 muestra que entre 10^8 y 10^9 iteraciones, la estrategia *helper core* se vuelve más eficiente que la estrategia *overlapping*.

5.2.3. Discusión sobre los resultados

La utilización GPU es reducida por el análisis in situ a pesar de que el benchmark PI no hace uso de la GPU. Para la estrategia *helper core*, una GPU no es usada durante la simulación mientras que para la estrategia *overlapping*, la utilización GPU es menor al 30 % para todas las pruebas realizadas. Gromacs realiza balanceo de cómputo entre la CPU y la GPU. Sin embargo, resultados previos (ver sección 3.3) muestran que la CPU es el cuello de botella de la simulación. Debido a que el benchmark PI estresa la CPU, Gromacs requiere más tiempo para lanzar los kernels GPU, dando lugar a más tiempo de inactividad en la GPU con la estrategia *overlapping*.

En síntesis, análisis in situ tradicional ejecutado en la CPU falla en mejorar el uso global de recursos de Gromacs en modo híbrido. Con la estrategia *helper core*, una GPU no es usada. Con la estrategia *overlapping*, el cuello de botella de la simulación es estresado aún más por el análisis dando lugar a más tiempo de inactividad en la GPU. Por tanto, se requiere de otras estrategias para mejorar el uso global de recursos.

5.3 Proposición: Portando las estrategias in situ a la GPU

Estrategias in situ comunes en CPU resultan inapropiadas para ejecutar cómputo extra cuando la CPU es el cuello de botella de la simulación, como es el caso de Gromacs. Dado que para ambas estrategias, *helper core* y *overlapping*, hay recursos GPU subutilizados, se propone el uso de las GPUs para ejecutar procesamiento in situ. Se adaptaron las dos estrategias para realizar análisis in situ en las GPUs. Se diseñaron dos micro benchmarks para estresar las GPUs y evaluar el impacto en el rendimiento de la simulación y la utilización de recursos.

5.3.1. Descripción de los benchmarks diseñados para GPU

Matrix Multiplication El kernel CUDA multMatrix de Nvidia es un kernel de cómputo intensivo que multiplica dos matrices varias veces. Los tamaños de las dos

matrices son 640x320 y 320x320, respectivamente. El número y de multiplicaciones realizadas durante una iteración del benchmark es definido por el usuario. Con la estrategia *helper core*, uno de los procesos in situ realiza todas las multiplicaciones de matrices (y). En el caso de la estrategia *overlapping*, N procesos in situ realizan y/N multiplicaciones de matrices.

Este benchmark ocupa las unidades de procesamiento de la GPU. Los kernels de la simulación tendrán menos multiprocesadores disponibles para ser calendarizados causando retrasos de los cálculos de simulación. El impacto de los benchmarks en el rendimiento de la simulación depende no solo de los benchmarks sino también del balance computacional CPU/GPU adoptado por la simulación. Si la CPU es el factor limitante de la simulación, los benchmarks GPU tendrán menor impacto y viceversa si la GPU es el factor limitante.

Bandwidth El kernel CUDA bandwidth de Nvidia estresa la comunicación entre la CPU y la GPU enviando y recibiendo paquetes de datos varias veces. El tamaño s del mensaje es definido por el usuario. Para la estrategia *helper core*, una GPU recibe mensajes de tamaño s. En el caso de la estrategia *overlapping*, cada una de las N GPUs recibe mensajes de tamaño s/N.

Los intercambios de datos se realizan con frecuencia entre la CPU y la GPU durante la simulación. Perturbar la transmisión de datos de la GPU puede retrasar la computación de la GPU de la simulación a la espera de transferencias de datos.

5.3.2. Resultados de procesamiento in situ en GPU

En lugar de estresar la CPU, que es el cuello de botella de la simulación, se propone ejecutar análisis in situ en la GPU. Primero se usó el benchmark multMatrix descrito en la sección 5.3.1. Como anteriormente, la salida de datos de Gromacs se realiza cada 100 iteraciones. Por cada salida de Gromacs, y multiplicaciones de matrices son ejecutadas en total.

La Figura 13 muestra el rendimiento de la simulación y la utilización GPU para las dos estrategias. Igual que para el benchmark en CPU, la estrategia *overlapping* es más eficiente para computación ligera. Para un número de matrices menor que y=3120 por salida de Gromacs, el rendimiento de la simulación se reduce en menos del 12 %, mientras que la utilización GPU permanece al mismo nivel que Gromacs nativo. Sin embargo, para un mayor número de multiplicaciones, el rendimiento decrece hasta en un 20 %, pero la utilización GPU incrementa hasta el 99 %.

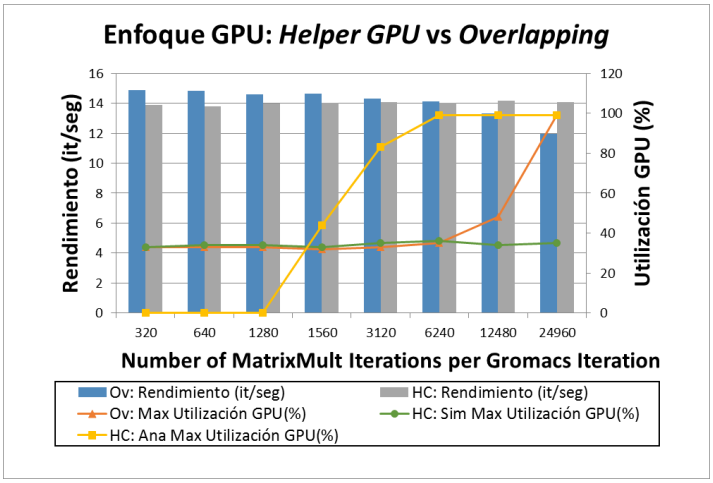


Figura 13: Estrategias *helper core* (HC) y *overlapping* (Ov) cuando se incrementa el número y de multiplicaciones de matrices ejecutadas por iteración de salida de Gromacs. La utilización GPU de la estrategia *helper core* esta dividido en dos curvas. Sim Max Utilización GPU indica la utilización máxima de las siete GPUs usadas por la simulación. Ana Max Utilización GPU indica la utilización de la GPU usada por multMatrix in situ.

La estrategia *helper core* (referido como HC), muestra un comportamiento más estable como en el caso de benchmark en CPU. El costo inicial con un número reducido de multiplicaciones de matrices es mayor que para la estrategia *overlapping*. Esto se debe a que las tareas in situ no están alojadas en los mismos recursos de la simulación, sin embargo el incremento en la carga computacional no afecta aún más el rendimiento de la simulación. La Figura 13 muestra que entre 6240 y 12480 multiplicaciones de matrices, la estrategia helper GPU mantiene un costo fijo y supera a la estrategia *overlapping*. Esta estrategia también permite el incremento en la utilización GPU hasta el 99 % de la GPU dedicada.

Se observaron las siguientes tendencias generales. Primero, las estrategias *overlapping* y *helper core* tienen comportamientos similares con análisis in situ en CPU y GPU. El costo de *overlapping* incrementa con un número creciente de computación in situ. La estrategia *helper core* tiene un costo inicial mayor para un número reducido de multiplicaciones pero no afecta aún más el rendimiento de simulación para un mayor coste computacional. En segundo lugar, realizar análisis in situ en la GPU mejora la utilización GPU. Debido a que la simulación no usa completamente las GPUs, otros kernels pueden ser lanzados con un impacto limitado en el rendimiento de la simulación.

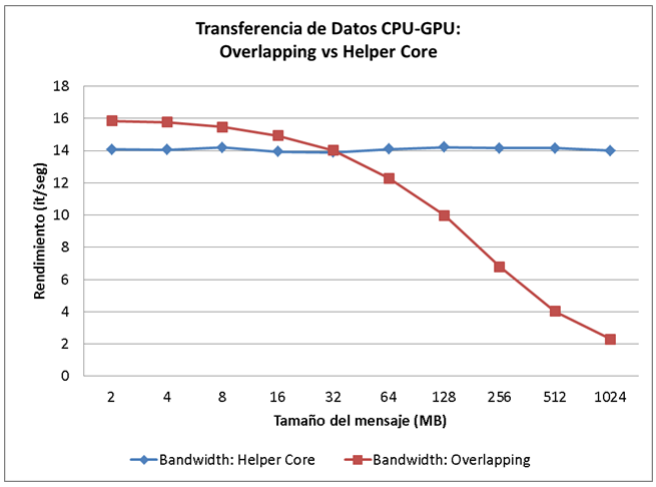


Figura 14: Estrategias *helper core* y *overlapping* cuando se incrementa el tamaño del mensaje transferido.

El benchmark multMatrix ejecuta operaciones pero no realiza transferencia de datos entre la CPU y la GPU. Solamente las unidades de procesamiento son estresadas. Sin embargo, cuando se realiza análisis intensivo de datos, la transferencia de datos debe ser considerada también. En la plataforma empleada, ocho GPUs están conectadas a tres puertos PCI express. Es decir, comparten el mismo bus para transferir datos desde/hacia la GPU. El benchmark Bandwidth descrito en la sección 5.3.1 se usó para evaluar el impacto del análisis intensivo de datos in situ en el rendimiento de la simulación. Como previamente, Gromacs efectúa salida de datos cada 100 iteraciones. Por cada salida de Gromacs, cinco ciclos de transferencia de datos fueron ejecutados cada uno con un tamaño de mensaje dado.

La Figura 14 muestra el rendimiento de la simulación para las dos estrategias. Para este benchmark, no se incluye la utilización GPU dado que el benchmark no usa las GPUs. El impacto de la estrategia *overlapping* en el rendimiento de la simulación es menos notorio cuando el tamaño del mensaje transferido es menor que 32MB. Sin embargo, para un tamaño mayor del mensaje, el rendimiento decrece hasta en un 85 % para mensajes de tamaño de 1GB. La estrategia *helper core* conserva un buen aislamiento entre la simulación y el análisis in situ y mantiene un costo fijo. Este resultado puede ser explicado por dos factores. Primero, las GPUs de la plataforma están conectadas a los buses PCI express en un patrón 3-3-2. La GPU dedicada fue ubicada en el tercer bus que solo alberga dos GPUs. Por lo tanto, el análisis in situ solo perturba una GPU de la simulación en el bus que es menos estresado. En segundo lugar, los códigos de dinámica molecular son códigos de no uso intensivo de datos. El modelo molecular completo representa solamente unos pocos MB de datos a transferir hacia/desde la

GPU. Esto deja espacio en los buses para transferir datos de otras tareas.

5.3.3. Discusión sobre los resultados

Con los experimentos realizados, se demostró que es posible aplicar las mismas estrategias de ubicación para análisis in situ en GPU como para CPU y observar comportamientos similares. Adicionalmente, en el caso de Gromacs, usar las GPUs para análisis in situ mejora la utilización GPU mientras se mantiene un costo similar a las estrategias CPU. Esto es posible por dos razones. Primero, el cuello de botella de Gromacs es la CPU en la configuración utilizada. Esto deja más espacio en la GPU que en la CPU. En segundo lugar, Gromacs no es una aplicación de uso intensivo de datos lo que la hace poco sensible a otras transferencias de datos.

Los experimentos también muestran que, en el caso de simulaciones híbridas, hay una necesidad de flexibilidad en la ubicación para procesar el análisis in situ. Para una simulación netamente CPU, la estrategia de ubicación depende del costo computacional y de la naturaleza del análisis. Sin embargo, para una simulación híbrida, el balance computacional CPU/GPU es otro parámetro a considerar. Dependiendo de si la CPU o la GPU es el cuello de botella de la simulación, el análisis in situ debe ser ejecutado en el recurso menos cargado si es posible.

La configuración de la simulación utilizada para los experimentos deja las GPUs inactivas durante un periodo de tiempo significativo. Esto permite calendarizar tareas in situ computacionalmente pesadas en las GPUs con la estrategia *overlapping* por un costo limitado. Sin embargo, otro tipo de simulaciones pueden usar las GPUs más intensivamente. Para tales escenarios, es probable que para las mismas tareas computacionales in situ, la estrategia *helper core* se vuelva más eficiente. Se espera que, a medida que la utilización de las GPUs por la simulación incremente, la carga computacional manejable a un costo razonable por la estrategia *overlapping* decrezca en favor de la estrategia *helper core*.

En este capítulo, se mostró que el análisis in situ en GPU puede ser una alternativa viable a las estrategias in situ en CPU. Las estrategias *helper core* y *overlapping* tienen un costo similar en el rendimiento de la simulación mientras que el análisis in situ en GPU permite usar de manera más eficiente las GPUs. Sin embargo, no es posible realizar una comparación directa del costo de estas estrategias debido a que se utilizan diferentes benchmark sintéticos. En el siguiente capítulo, se propone un caso real de análisis in situ implementado en la CPU y la GPU.

6. GROMACS Y QUICKSURF: CASO DE ESTUDIO

En este capítulo se propone realizar un estudio de un caso real de análisis in situ implementado en la CPU y la GPU. En la primera parte se presenta una descripción del algoritmo empleado, Quicksurf. En la segunda parte, se propone una implementación GPU del algoritmo Quicksurf. El objetivo es comparar la implementación GPU propuesta con la implementación CPU existente en términos de impacto en el rendimiento de la simulación y utilización de recursos. Las estrategias in situ *helper core* y *overlapping* se usan como configuración para la comparación.

6.1 Descripción del algoritmo Quicksurf

Una aplicabilidad común del procesamiento in situ es generar imágenes del estado actual de la simulación. En dinámica molecular, sin embargo, una imagen ofrece información limitada del fenómeno simulado. Por ende, resulta más apropiado una representación 3D.

La representación atomística de grandes sistemas moleculares resulta en el despliegue de grandes volúmenes de partículas que dificulta su análisis por parte del usuario. En estos casos, es conveniente una representación de la superficie molecular que permita tener una percepción 3D de la forma de una molécula y de posibles cambios en su estructura. El algoritmo Quicksurf, propuesto por Krone et al[cite], permite la visualización de la superficie molecular de sistemas con grandes aglomeraciones de partículas. Quicksurf permite la construcción de una malla de triángulos que representa la superficie del sistema molecular.

La estructura del algoritmo Quicksurf está comprendida por tres pasos. En un primer paso, la escena es inmersa en una grid regular 3D. Como segundo paso, para cada celda de la grid, se calcula la densidad Gaussiana como una función del número de átomos contenidos en la celda y en sus vecindades. Por último, se extrae la isosuperficie del gráfico usando el algoritmo Marching Cube.

Dreher et al presentan una implementación CPU del algoritmo Quicksurf usando FlowVR, comprendida por cuatro módulos paralelos que se describen a continuación (ver figura 15):

- **Módulo 1 (Morton):** Calcula el código Morton para la posición de cada átomo previamente asociado a una celda. El código Morton corresponde al índice de una celda de la grid y para calcularlo, el módulo tiene las dimensiones de la grid regular así como su posición en el espacio. Finalmente los átomos son ordenados

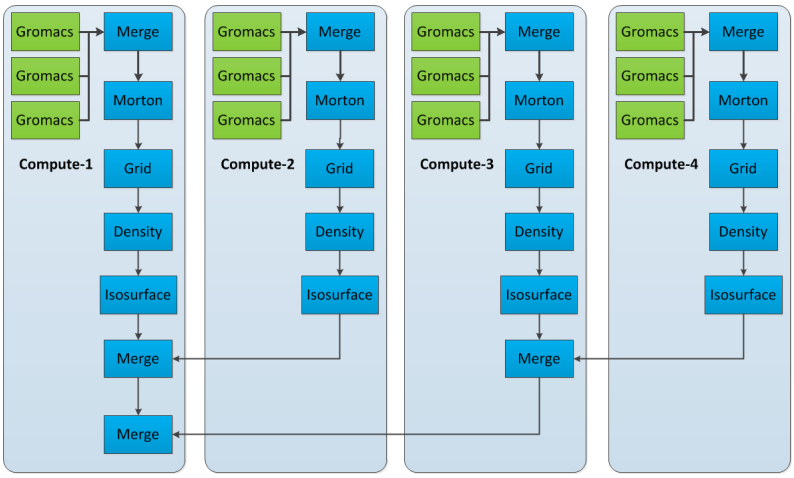


Figura 15: Implementación CPU del algoritmo Quicksurf usando FlowVR.

por su código Morton. Las posiciones de los átomos y sus respectivos índices Morton son enviados al siguiente módulo sin copiar.

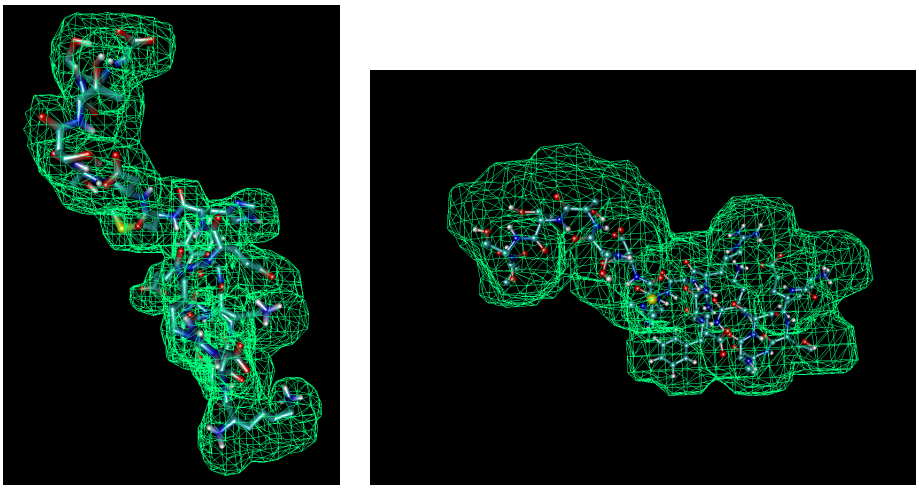
- **Módulo 2 (Grid):** Se encarga de construir la estructura 3D de la grid. Para cada celda de la grid, se combinan dos enteros. El primer entero representa el índice del primer átomo bandera en la celda. El segundo, es el número de átomos que pertenecen a la celda. La posiciones de los átomos, sus índices Morton y la estructura de la grid son enviados al siguiente módulo sin copiar.
- **Módulo 3 (Density):** Calcula los valores de densidad de la celda. La densidad de una celda está basado en el número de átomos, sus posiciones y sus diámetros en la celda actual y sus 27 vecinos. Finalmente, la escritura de la grid y las densidades son enviados sin copiarse al último módulo.
- **Módulo 4 (Isosurface):** Ejecuta el algoritmo Marching Cube. El último paso es recolectar todos los productos de triángulos para completar la isosuperficie.

La figura 16 muestra el resultado final del procesamiento y visualización de un péptido con Gromacs y Quicksurf respectivamente.

6.2 Implementación GPU del algoritmo Quicksurf

6.2.1. OpenACC

OpenACC (**O**pen **Acc**elerators) es un estándar de programación de cómputo paralelo para sistemas heterogéneos CPU/Acelerador desarrollado por Cray, CAPS, Nvidia and PGI. OpenACC tiene soporte para aceleradores como Nvidia GPU, x86 o ARM CPU,



(a) (b)

Figura 16: Procesamiento y visualización de un péptido con Gromacs y Quicksurf.

Intel Xeon Phi y AMD Radeon, es usado para acelerar códigos seriales en C, C++ y Fortran mediante directivas de compilación identificadas con `#pragma acc [clausulas]`. Estas directivas especifican al compilador las regiones paralelas a ejecutarse en el acelerador.

El cómputo intensivo puede ser gestionado a través de dos tipos de constructores: *kernels* (`#pragma acc kernels [argumentos]`) y *parallel* (`#pragma acc parallel [argumentos]`). Un bucle o un conjunto de bucles pueden ser gestionados por un constructor. Bucles anidados serán ejecutados en paralelo. Cuando hay una cantidad de bucles anidados superior a tres, los bucles externos serán ejecutados de manera serial en la CPU y los tres bucles internos se ejecutarán en el acelerador. El constructor *kernels* especifica una función denominada kernel a ejecutarse en el acelerador, donde el compilador se encarga de definir parametros tales como dimensiones de la grid de acuerdo a las características del acelerador. Mientras que para el constructor *parallel*, es el programador quien se encarga de definir el entorno de ejecución.

La figura 17 se presenta la función `saxpy` ($AX+Y$ en precisión simple) implementando directivas de OpenACC para la ejecución del bucle en paralelo. La línea 8 del código indica al compilador que el bucle deberá ser ejecutado en el acelerador.

Debido a que la CPU y el acelerador cuentan con memorias físicamente separadas, se requiere establecer una comunicación CPU/acelerador. El modelo de programación de OpenACC permite realizar transferencia de datos entre la memoria de la CPU y del acelerador a través de un bus PCI-Express. La transferencia de datos es definida por el

```
// Funcion saxpy empleando directivas OpenACC
void saxpy_parallel(int n,
  float a,
  float *x,
  float *restrict y)
{
// Constructor kernels
#pragma acc kernels
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
```

Figura 17: Función saxpy usando directivas OpenACC

programador. Si bien se puede obtener un mejor rendimiento al usar un sistema híbrido CPU/acelerador, este rendimiento puede verse opacado por el costo en tiempo de la transferencia.

En la figura 18 se presenta transferencias de datos CPU/acelerador que se define dentro de un constructor tipo kernels. Las funciones `copyin` y `copyout` realizan asignación de memoria en la GPU. `copyin` indica al compilador que los vectores `a` y `b` deberán ser copiados desde la memoria de la CPU hacia el acelerador, siendo `a` y `b` vectores previamente inicializados. Por otra parte, `copyout` indica al compilador que el vector `c`, una vez calculado, deberá ser copiado desde la memoria de la GPU hacia el acelerador.

```
// Transfencia de datos CPU/Acelerador
#pragma acc data copyin(a[0:n],b[0:n]), copyout(c[0:n]){
  #pragma acc kernels
  for(i=0; i<n; i++){
    c[i] = a[i] + b[i];
  }
}
```

Figura 18: Transferencia de datos entre la memoria de la CPU y del acelerador

6.2.2. Librería Thrust

Thrust es una librería CUDA basada en Standard Template Library (STL) que provée una colección de primitivas de datos paralelas tales como `scan`, `sort` y `reduce`.

Función `thrust::sort()`:
`sort` es una función disponible dentro de la biblioteca de Thrust que permite el orde-

namiento ascendente de datos, empleando el operador menor que (<). Al invocar la función `thrust::sort()`, los datos a ordenar son transferidos desde la memoria de la CPU hacia la GPU. Los datos son ordenados en la GPU y el resultado es copiado de regreso hacia la CPU. En la figura 19 se muestra el ordenamiento del vector `A` de 6 elementos, utilizando la función `sort`. Los parametros de entrada de la función son el inicio y final de la secuencia a ordenar.

```
#include <thrust/sort.h>
...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
thrust::sort(A, A + N);
// A es ahora {1, 2, 4, 5, 7, 8}
```

Figura 19: Función `thrust::sort()`

Una de las variantes de la función `sort`, es `sort_by_key`. Esta función permite ordenar dos vectores, de acuerdo al ordenamiento ascendente del vector llave empleando el operador menor que (<). Los parametros de entrada de la función son el inicio y final de la secuencia llave, y el inicio de la secuencia de valores. En la figura 20 se muestra el ordenamiento de los vectores `keys` y `values`, ambos vectores con longitud 6 elementos. El vector `keys` es ordenado de manera ascendente, mientras que cada elemento `j` del vector `values` es ordenado manteniendo la relación inicial con el respectivo a cada elemento `i` del vector `keys`.

```
#include <thrust/sort.h>
 ...
 const int N = 6;
 int    keys[N] = { 1,   4,   2,   8,   5,   7};
 char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
 thrust::sort_by_key(keys, keys + N, values);
 // keys es ahora   { 1,   2,   4,   5,   7,   8}
 // values es ahora {'a', 'c', 'b', 'e', 'f', 'd'}
```

Figura 20: Función `thrust::sort_by_key()`

6.2.3. Quicksurf en GPU

Para la implementación GPU del algoritmo quicksurf que se propone, se toma como base la implementación CPU presentada en la sección 6.1. De los cuatro módulos que componen la versión CPU, se referencia el módulo 1 y módulo 3 como los módulos en los

que realiza el cómputo más intensivo del algoritmo. EL módulo 1, denominado *Morton*, calcula el código Morton para cada átomo de la molécula y realiza un ordenamiento de los átomos. El módulo 3, denominado *Density*, calcula los valores de densidad de cada celda de la grid teniendo en cuenta las vecindades.

En la figura 21 se presenta un fragmento del código correspondiente al módulo Morton. En la línea número dos, la directiva `#pragma acc data` indica al compilador asignar espacio de memoria para las variables `positions2` y `morton2` en la GPU, y posteriormente, se especifican las transferencias de datos de las dos variables, que se realizarán al inicio y/o final del bloque de instrucciones, entre la memoria de la CPU y la memoria de la GPU.

El vector `positions2` de `nbElement` número de elementos es copiado desde la CPU hacia la GPU el principio del bloque de instrucciones y posteriormente copiado de regreso hacia la CPU al final del bloque, usando la clausula `copy`. La clausula `copyout` indica al compilador que una vez sea calculado en la GPU, el vector `morton2` de `nbElement` número de elementos, deberá ser copiado hacia la CPU. El vector `positions2` almacena las posiciones de los átomos de la molécula. El vector `morton2` almacena el código Morton de cada uno de los átomos y es calculado en la GPU. En la línea número tres, se especifica al compilador que el bucle deberá ser ejecutado en paralelo en la GPU. Dentro de este bucle se encuentra la función `Morton_3D_Encode_10bit` (línea 17) que realiza el cálculo del código Morton para cada átomo.

Una vez calculado, el vector `morton2` debe ser ordenado. En la línea 21 se indica que las variables `morton2` y `positions2`, previamente alocadas en la memoria de la GPU serán utilizadas en el bloque actual. Esta instrucción es necesaria para asegurar la interoperabilidad entre OpenACC y Thrust. En la línea 22 se invoca la función `sort` que implementa a la función `sort_by_key`. Para realizar el ordenamiento se toma como llave el vector de códigos Morton.

En la figura 22 se presenta un fragmento del código correspondiente al módulo *Density*. En la línea número uno, la directiva `#pragma acc data` indica al compilador asignar espacio de memoria para las variables `density` y `grid` en la GPU. Posteriormente se especifican las transferencias de datos de las variables. El vector `density` de `nbCells` número de elementos es copiado desde la GPU hacia la CPU al final del bloque de instrucciones. Este vector almacena los valores de densidades de cada una de las celdas de la grid. El vector `grid`, será copiados únicamente al inicio del bloque desde la CPU hacia la GPU.

```
//Computation of the morton codes of the atoms
#pragma acc data copy(positions2[0:nbElement]), copyout(morton2[0:nbElement]){
  #pragma acc kernels
  for(unsigned int i = 0; i < nbElement; i++){

    //Using cast from float to unsigned int to keep the lower int
    unsigned int cellX = (unsigned int)((positions2[i].posX - xmin) / unitX);
    unsigned int cellY = (unsigned int)((positions2[i].posY - ymin) / unitY);
    unsigned int cellZ = (unsigned int)((positions2[i].posZ - zmin) / unitZ);

    //Clamping the cells to the bbox. Atoms can move away from the box, we count them
    in the nearest cell (although it's not correct)
    cellX = cellX >= (dX)?(dX-1):cellX;
    cellY = cellY >= (dY)?(dY-1):cellY;
    cellZ = cellZ >= (dZ)?(dZ-1):cellZ;

    //Computing the corresponding morton code
    morton2[i] = Morton_3D_Encode_10bit(cellX,cellY,cellZ);
  }

//Sorting according to the morton code.
  #pragma  acc host_data use_device(morton2, positions2){
    sort(morton2, positions2, nbElement, stream);
  }
}
```

Figura 21: Fragmento del módulo *Morton*: Implementación GPU con OpenACC y Thrust

```
#pragma acc data copyout(density[0:nbCells]), copyin(grid[0:nbCells])
{
    #pragma acc kernels
    for(int id = 0; id < nbCells; id++)
    {
      if(grid[id].second > 0)
        density[id] = 1.0f;
      else
        density[id] = 0.0f;
    }
}
```

Figura 22: Fragmento del módulo *Density*: Implementación GPU con OpenACC

Por simplicidad, se removieron del código de ejemplo el cómputo de las celdas vecinas. Dicho cómputo está compuesto por 27 `if` para testear si hay átomos presentes en las celdas vecinas. La versión CPU implementa tres bucles anidados para tal fin. Sin embargo, debido a los movimientos de desplazamiento en los bucles, OpenACC fue incapaz de paralelizar esta sección. Los bucles fueron reemplazados para testear a mano todos los casos.

6.3 Implementación CPU vs GPU del algoritmo Quicksurf

En esta sección se presenta una comparación directa del costo de la versión CPU existente del algoritmo Quicksurf y la versión GPU propuesta en la sección 6.2.3 en el rendimiento de la simulación y la utilización de recursos. Como configuración para realizar la comparación se emplean las estrategias *helper core* y *overlapping*. Para la ejecución de las pruebas se utilizó la misma configuración de la plataforma y sistema molecular descrito en la sección 3.2. La salida de datos de Gromacs se realizó cada N iteraciones de la simulación, variando desde $N = 10$ hasta $N = 250$. Nótese que contrario a la sección anterior, las gráficas son presentadas con una carga computacional decreciente.

6.3.1. Resultados para la estrategia *overlapping*

En la figura 23 se presenta el rendimiento de la simulación con la estrategia *overlapping* para las implementaciones CPU (azul) y GPU (rojo) del algoritmo Quicksurf. Se tomó el rendimiento de Gromacs *stand alone* (verde) como parámetro de referencia. Para esta configuración, tanto la simulación como el análisis tienen acceso a 8 núcleos CPU y 8 GPUs disponibles en la máquina. Para $N = 10$, tanto la versión CPU como GPU tienen un rendimiento inferior a la versión Gromacs *stand alone*. En el caso de la versión CPU en un 38 %, mientras que la versión GPU en un 23 %, siendo éste el peor de los casos observados. A medida que aumenta en valor de N, el rendimiento de la simulación para las dos implementaciones tiende hacia el rendimiento de Gromacs *stand alone*. Para $N = 250$, se obtiene un rendimiento inferior de 0.7 % en el caso de la versión CPU comparado con Gromacs *stand alone*, mientaras que la versión GPU no tiene costo en el rendimiento de la simuación. Comparado con los resultados de la versión de Gromacs *stand alone*, no hay un incremento en la utilización de las GPUs para las dos implementaciones.

6.3.2. Resultados para la estrategia *helper core*

En la figura 24 se presenta el rendimiento de la simulación con la estrategia *helper core* para las implementaciones CPU (azul) y GPU (rojo) del algoritmo Quicksurf. Se tomó el rendimiento de Gromacs *stand alone* (verde) como parámetro de referencia, ejecutado en 7 núcleos CPU de un total de 8 núcleos y 7 GPUs de un total de 8 GPUs disponibles en la máquina. Esta configuración de Gromacs *stand alone* tiene un costo de 12 % en el rendimiento de la simulación, equivalente al porcentaje de recursos que son tomados de la simulación para la ejecución del análisis in situ -un núcleo CPU y una GPU-. Las pruebas realizadas muestran que el rendimiento de la simulación es similar para las implementaciones CPU y GPU de Quicksurf. Para $N = 10$, el costo

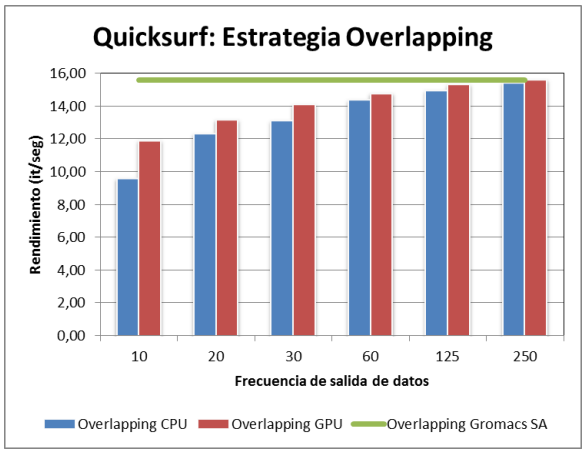


Figura 23: Comparación de las implementaciones CPU y GPU del algoritmo Quicksurf usando la estrategia *overlapping*

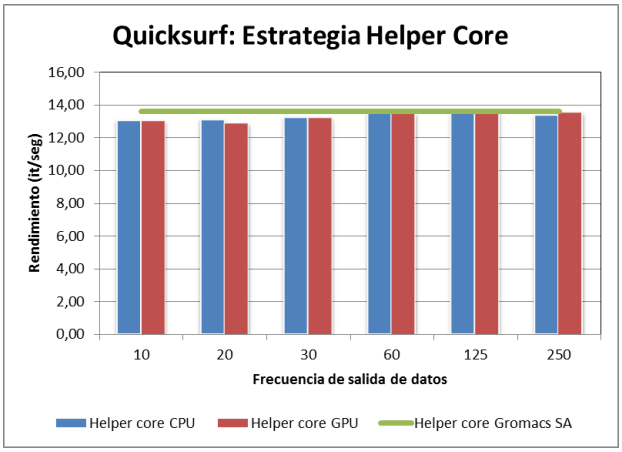


Figura 24: Comparación de las implementaciones CPU y GPU del algoritmo Quicksurf usando la estrategia *helper core*

de las dos implementaciones es de 4 % comparado con Gromacs *stand alone*. Para $N = 250$, el costo de la implementación CPU es de 2 %, mientras que el costo de la implementación GPU es cercano a cero. Comparado con los resultados de la versión de Gromacs *stand alone*, no hay un incremento en la utilización de las GPUs para las dos implementaciones.

6.4 Discusión sobre los resultados

Con los experimentos realizados se logró realizar una comparación directa del costo de la versión CPU existente del algoritmo Quicksurf y la versión GPU que se propone en

este trabajo en el rendimiento de la simulación y la utilización de recursos.

Con la estrategia *helper core* se observó un comportamiento constante en el rendimiento de la simulación. El procesamiento in situ de las implementaciones CPU y GPU del algoritmo Quicksurf tienen un costo fijo en el rendimiento de la simulación. Estos resultados son esperados debido a que para realizar procesamiento de análisis in situ se reduce la cantidad de recursos a usar por la simulación. Sin embargo, la implementación GPU permite el uso de más recursos computacionales en el procesamiento del análisis, lo cual puede permitir la aceleración de la fase de análisis.

La estrategia *overlapping* usando GPUs para el análisis in situ parece más prometedora en el caso del estudio presentado. Para cada frecuencia de salida de datos, la implementación GPU supera la versión CPU, especialmente para frecuencias altas. Nótese que solamente la mitad del flujo de ejecución del algoritmo Quicksurf se ejecuta en las GPUs. Se espera reducir aún más el costo con una implementación completa en las GPUs.

En general, la implementación GPU se muestra como la solución más eficiente para el caso de estudio. Para todos los casos, el análisis en GPUs genera un costo en la simulación igual o inferior que la implementación CPU mientras permite el uso de más recursos computacionales para el análisis.

Tanto para la estrategia *helper core* como *overlapping* se observó que el porcentaje de utilización de las GPU permanece constante para las dos implementaciones CPU y GPU. Este resultado es inesperado. Como el análisis in situ es ejecutado en la GPU, se esperaba una mejora en el porcentaje de utilización de la GPU. Este resultado se debe a que los *kernels* diseñados para el algoritmo Quicksurf son cortos y su tiempo de ejecución es despreciable. También se observó que la mitad del tiempo de procesamiento del análisis in situ es usado en transferencia de datos entre la CPU y la GPU. La implementación CPU del algoritmo emplea 4 módulos, cada uno de los cuales requiere enviar datos hacia la GPU y transferir de regreso hacia la CPU una vez han sido computados. Debido a esta implementación modular, se genera un gran número de transferencias de datos entre la CPU y la GPU.

Se espera que con una implementación del algoritmo que emplee uno en lugar de cuatro módulos, se disminuya las transferencias de datos y por ende el tiempo de cómputo del procesamiento in situ. Otro beneficio sería el incremento de la utilización de las GPU debido a la unificación de *kernels* cortos en un único *kernel*. Esta alternativa se plantea como trabajo futuro a este proyecto.

Una de las principales dificultades encontradas durante el desarrollo del trabajo fue el uso de OpenACC para la aceleración del algoritmo Quicksurf. OpenACC parecía una solución apropiada ya que la mayoría del cómputo a realizar consiste en múltiples bucles. Sin embargo, la ausencia de visibilidad en el código final (código generado por OpenACC) y la falta de precisión en los mensajes de error dificultaron el proceso de desarrollo y depuración. Adicionalmente, el patrón de bucles en el módulo Density debió ser simplificado para hacer posible la paralelización por OpenACC.

OpenACC puede ser una buena solución para bucles simples como el presentado en el módulo Morton. En el caso de códigos más complejos, CUDA C puede ser una solución más apropiada pues el esfuerzo requerido en la codificación es compensado por la ganancia en flexibilidad, visibilidad completa del código y métodos de depuración más simples.

Futuras máquinas Exascale integrarán GPUs or Xeon Phi, el cual es otro tipo de acelerador. Una interesante investigación que se puede conducir es extender este estudio al caso de procesadores Xeon Phi con fines comparativos.

7. CONCLUSIONES

Las aplicaciones in situ actuales están enfocadas en simulación y análisis ejecutado solamente en la CPU. Las aplicaciones híbridas CPU/GPU contemplan en su diseño el balanceo de carga entre la CPU y la GPU. Para este tipo de aplicaciones, no es claro como el procesamiento in situ puede afectar el balance y por ende, el rendimiento de la simulación y la utilización de recursos. Para el desarrollo de este trabajo se realizó un estudio de la utilización de recursos por aplicaciones in situ en el caso de simulaciones híbridas CPU/GPU. Como caso de estudio se utilizó Gromacs, una aplicación para la simulación de dinámica molecular. Para conducir el estudio, se aplicaron dos estrategias comunes de procesamiento in situ en CPU. Los resultados obtenidos muestran que el balanceo de carga entre la CPU y la GPU de Gromacs es insuficiente para gestionar los recursos disponibles y por ende, hay recursos que están siendo subutilizados. En este trabajo se propuso el uso de recursos subutilizados por la simulación para la ejecución de análisis in situ, empleando el algoritmo Quicksurf, un algoritmo de renderización, para tal fin. Debido a que en el caso de Gromacs, la CPU es el cuello de botella de la simulación, se encontró que es más eficiente el uso de GPUs para procesamiento in situ. Esta configuración permitió aprovechar las GPUs para acelerar la fase de análisis con un costo mínimo en el rendimiento de la simulación. Se espera que con este proyecto de maestría se haya mostrado que, cuando se consideran simulaciones híbridas en el contexto de aplicaciones in situ, el balance computacional entre la CPU y la GPU debería ser tenido en cuenta cuando se considera análisis in situ. Esto hace un llamado a la necesidad de mayor flexibilidad en la ubicación del análisis in situ (CPU o GPU) y a completar las estrategias de ubicación in situ existentes.

Para la implementación GPU del algoritmo Quicksurf que se planteó en este trabajo, solo se paralelizaron dos de los cuatro módulos que conforman la versión CPU del algoritmo. Como trabajo futuro se plantea unificación de los cuatro módulos en un único módulo de manera tal que haya solo un kernel que requiera de mayor poder computacional. Se espera que esta implementación permita disminuir las operaciones de transferencia de datos entre la CPU y la GPU, mejorar la utilización de las GPUs y reducir aún más el impacto en el rendimiento de la simulación.

En el contexto del caso de estudio presentado, se evidencia que Gromacs tiene la particularidad de subutilización de recursos GPU. Sin embargo, estas observaciones son específicas de la simulación y no pueden ser generalizadas. Una interesante investigación que se puede conducir a partir de los resultados prestados en este proyecto es extender este estudio al procesamiento in situ en GPU de otras aplicaciones híbridas CPU/GPU.

Portar el análisis a la GPU puede ser una tarea laboriosa. Una herramienta interesante para justificar este proceso sería contar con un modelo analítico que, basado en la utilización de recursos, la configuración de la máquina y otros parametros, pueda estimar el rendimeinto potencial a ganar al portar el análisis en las GPUs.

BIBLIOGRAFÍA

[1] Allard, J., Gouranton, V., Lecointre, L., Limet, S., Melin, E., Raffin, B., and Robert, S. FlowVR: A Middleware for Large Scale Virtual Reality Applications. In *Proceedings of Euro-par 2004* (Pisa, Italia, August 2004).

[2] Docan, C., Parashar, M., and Klasky, S. DataSpaces: an Interaction and Coordination Framework for Coupled Simulation Workflows. *Cluster Computing 15* (2012).

[3] Dorier, M., Antoniu, G., Cappello, F., Snir, M., and Orf, L. Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-Free I/O. In *CLUSTER - IEEE International Conference on Cluster Computing* (Sept. 2012), IEEE.

[4] Dorier, M., Sisneros, Roberto, R., Peterka, T., Antoniu, G., and Semeraro, Dave, B. Damaris/Viz: a Nonintrusive, Adaptable and User-Friendly In Situ Visualization Framework. In *LDAV - IEEE Symposium on Large-Scale Data Analysis and Visualization* (Atlanta, United States, Oct. 2013).

[5] Dreher, M., Piuzzi, M., Ahmed, T., Matthieu, C., Baaden, M., Férey, N., Limet, S., Raffin, B., and Robert, S. Interactive Molecular Dynamics: Scaling up to Large Systems. In *International Conference on Computational Science, ICCS 2013* (Barcelone, Spain, June 2013), Elsevier.

[6] Dreher, M., and Raffin, B. A Flexible Framework for Asynchronous In Situ and In Transit Analytics for Scientific Simulations. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on* (May 2014).

[7] Fabian, N., Moreland, K., Thompson, D., Bauer, A., Marion, P., Geveci, B., Rasquin, M., and Jansen, K. The Paraview Coprocessing Library: A Scalable, General Purpose In Situ Visualization Library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on* (Oct 2011).

[8] Fowler, P. W. Gromacs 4.6: Scaling of a very large coarse-grained system. http://philipwfowler.wordpress.com/2013/10/23/gromacs-4-6-scaling-of-a-very-large-coarse-grained-system/, 2015.

[9] Hagan, R., and Cao, Y. Multi-gpu load balancing for in-situ visualization. In *The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications* (2011).

[10] Hess, B., Kutzner, C., van der Spoel, D., and Lindahl, E. GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation. *Journal of Chemical Theory and Computation* (2008).

[11] Humphrey, W., Dalke, A., and Schulten, K. VMD – Visual Molecular Dynamics. *Journal of Molecular Graphics 14* (1996), 33–38.

[12] Klasky, S., Ethier, S., Lin, Z., Martins, K., Mccune, D., and Samtaney, R. Grid-Based Parallel Data Streaming implemented for the Gyrokinetic Toroidal Code. In *In Supercomputing Conference (SC 2003* (2003), IEEE Computer Society.

[13] Krone, M., Stone, J. E., Ertl, T., and Schulten, K. Fast Visualization of Gaussian Density Surfaces for Molecular Dynamics and Particle System Trajectories. In *EuroVis 2012 Short Papers* (2012), vol. 1.

[14] Levine, B. G., Stone, J. E., and Kohlmeyer, A. Fast analysis of molecular dynamics trajectories with graphics processing units Radial distribution function histogramming. *Journal of Computational Physics 230*, 9 (2011).

[15] Li, M., Vazhkudai, S. S., Butt, A. R., Meng, F., Ma, X., Kim, Y., Engelmann, C., and Shipman, G. Functional Partitioning to Optimize End-to-End Performance on Many-core Architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society.

[16] Lofstead, J. F., Klasky, S., Schwan, K., Podhorszki, N., and Jin, C. Flexible IO and Integration for Scientific Codes Through the Adaptable IO System (ADIOS). In *6th international workshop on Challenges of large applications in distributed environments* (2008).

[17] Lorendeau, B., Fournier, Y., and Ribes, A. In-Situ Visualization in Fluid Mechanics Using Catalyst: A Case Study for Code Saturne. In *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on* (Oct 2013).

[18] Moreland, K. Oh, $#! Exascale! The Effect of Emerging Architectures on Scientific Discovery. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* (Nov 2012).

[19] Phillips, J. C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R. D., KalÃ©, L., and Schulten, K. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry 26*, 16 (2005), 1781–1802.

[20] Pronk, S., Pall, S., Schulz, R., Larsson, P., Bjelkmar, P., Apostolov, R., Shirts, M. R., Smith, J. C., Kasson, P. M., van der Spoel, D., Hess, B., and Lindahl, E. Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics* (2013).

[21] Singh, A., Balaji, P., and Feng, W.-c. GePSeA: A General-Purpose Software Acceleration Framework for Lightweight Task Offloading. In *Proceedings of the 2009 International Conference on Parallel Processing* (Washington, DC, USA, 2009), ICPP '09, IEEE Computer Society.

[22] Soumagne, J., and Biddiscombe, J. Computational Steering and Parallel Online Monitoring Using RMA Through the HDF5 DSM Virtual File Driver. In *Proceedings of the International Conference on Computational Science, ICCS 2011* (Singapore, June 2011), vol. 4.

[23] Stone, J. E., Hardy, D. J., Ufimtsev, I. S., and Schulten, K. Gpu-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modelling 29*, 2 (2010).

[24] Stone, J. E., Kohlmeyer, A., Vandivort, K. L., and Schulten, K. Immersive molecular visualization and interactive modeling with commodity hardware. In *ISVC'10* (2010), Springer-Verlag.

[25] Stone, J. E., McGreevy, R., Isralewitz, B., and Schulten, K. Gpu accelerated analysis and visualization of large structures solved by molecular dynamics flexible fitting. *Faraday Discuss. 169* (2014).

[26] Stone, J. E., Vandivort, K. L., and Schulten, K. Gpu-accelerated molecular visualization on petascale supercomputing platforms. In *Proceedings of the 8th International Workshop on Ultrascale Visualization* (New York, NY, USA, 2013), UltraVis '13, ACM.

[27] Tu, T., Yu, H., Ramirez-Guzman, L., Bielak, J., Ghattas, O., Ma, K.-L., and O'Hallaron, D. From Mesh Generation to Scientific Visualization: An End-to-End Approach to Parallel Supercomputing. In *SC 2006 Conference, Proceedings of the ACM/IEEE* (Nov 2006).

[28] Vishwanath, V., Hereld, M., and Papka, M. Toward Simulation-Time Data Analysis and I/O Acceleration on Leadership-Class Systems. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on* (Oct 2011).

[29] Whitlock, B., Favre, J. M., and Meredith, J. S. Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System. In *Proceedings of*

the 11th Eurographics Conference on Parallel Graphics and Visualization (2011), EGPGV '11, Eurographics Association.

[30] Yu, H., Wang, C., Grout, R., Chen, J., and Ma, K.-L. In situ visualization for large-scale combustion simulations. *Computer Graphics and Applications, IEEE* (2010).

[31] Zhao, G., Perilla, J. R., Yufenyuy, E. L., Meng, X., Chen, B., Ning, J., Ahn, J., Gronenborn, A. M., Schulten, K., and Aiken, C. Mature HIV-1 Capsid Structure by Cryo-electron Microscopy and All-Atom Molecular Dynamics, 2013.

[32] Zheng, F., Abbasi, H., Cao, J., Dayal, J., Schwan, K., Wolf, M., Klasky, S., and Podhorszki, N. In-situ I/O Processing: A Case for Location Flexibility. In *Proceedings of the Sixth Workshop on Parallel Data Storage* (New York, NY, USA, 2011), PDSW '11, ACM.

[33] Zheng, F., Abbasi, H., Docan, C., Lofstead, J., Liu, Q., Klasky, S., Parashar, M., Podhorszki, N., Schwan, K., and Wolf, M. PreDatA - Preparatory Data Analytics on Peta-Scale Machines. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (2010).

[34] Zheng, F., Yu, H., Hantas, C., Wolf, M., Eisenhauer, G., Schwan, K., Abbasi, H., and Klasky, S. Goldrush: Resource Efficient In Situ Scientific Data Analytics Using Fine-Grained Interference Aware Execution. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013), SC '13, ACM.

[35] Zheng, F., Zou, H., Eisenhauer, G., Schwan, K., Wolf, M., Dayal, J., Nguyen, T.-A., Cao, J., Abbasi, H., Klasky, S., Podhorszki, N., and Yu, H. FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing* (2013), IPDPS '13, IEEE Computer Society.