UNIVERSIDAD DEL BÍO-BÍO

INGENIERÍA DE EJECUCIÓN EN COMPUTACIÓN E INFORMÁTICA

Facultad de Ciencias Empresariales

Optimización de software de simulación de dinámicas moleculares para sistemas de interacción fluido-grano

Memoria para optar al título de Ingeniero de Ejecución en Computación e Informática

---

Alumno:

Carlos RÍOS V.

Supervisor:

Sr. Patricio GALVEZ G.

Dr. Dino RISSO R.

Concepción, febrero de 2013

Resumen

Los medios granulares son elementos que están presentes en gran parte de nuestras vidas, y cuentan con un gran interés científico debido a su alto impacto en industrias como la minería, farmacéutica, y construcción entre otras.

El estudio de estos medios actualmente se ha llevado a cabo por medio de simulaciones computacionales, específicamente dinámicas moleculares. Este tipo de técnicas al utilizarlas en sistemas de muchas partículas requiere de una gran cantidad de poder de cálculo. Ante esta necesidad, los sistemas de computación paralela se muestran como una alternativa al problema, incluyendo nuevos métodos y tecnologías de computación paralela, donde últimamente destaca la utilización de tarjetas gráficas (***GPUs***).

El presente trabajo se presenta a *Adentu*, un framework Open Source para la creación de software de dinámicas moleculares híbridas, para el estudio de interacciones fluido-grano, el cual permite mezclar dinámicas moleculares conducidas por tiempo junto a dinámicas moleculares conducidas por eventos, haciendo uso de la tecnología de computación paralela por GPU llamado **CUDA** de NVIDIA.

Agradecimientos

Quisiera agradecer, primero que todo a mi familia, quienes me han dado las herramientas, el apoyo y el cariño que solo ellos podrían darme. Sin ellos no podría haber culminado este ciclo.

También quisiera, con un abrazo enorme, agradecer el apoyo y el aliento que me han dado mis amigos, los cuales en momentos de flaqueza me inyectaron de fuerzas para continuar.

Gracias a los profesores Dino Risso R. quien fue el que me abrió las puertas al mundo de la computación paralela y me acepto hacer la memoria con él. Y al profesor Patricio Galvez G. quien me ha apoyado durante todo mi periodo en la carrera.

Agradecer también al Laboratorio de Biofísica Molecular de la Universidad de Concepción, donde descubrí el mundo de la computación científica.

Finalmente agradecer a FONDECYT y al Proyecto Fondecyt Regular N° 1120775 dirigido por el Dr. Dino Risso R. con el cual su financiamiento hizo posible este desarrollo.

Gracias a todos.

Índice general

Índice de figuras

Capítulo 1

Introducción

Los medios granulares son elementos que están presentes en gran parte de nuestras vidas, podemos encontrar este tipo de medio en los granos de cereal que comemos, en la arena de la playa, y hasta en el cemento utilizado para construir las ciudades. Existe una gran cantidad de materiales que tienen la característica de materia (o medio) granular. Estos medios son colecciones de muchas partículas que tienen un tamaño suficientemente grande para que las fluctuaciones térmicas no sean importantes.

Este tipo de medio es el segundo más explotado en el planeta, antecedido sólo por el agua [8]. Entre los procesos e industrias que involucran materiales granulares están: la construcción, los cosméticos, la comida y la industria farmacéutica. Los materiales granulares poseen una innumerable cantidad de propiedades y comportamientos que los hacen atractivos para el estudio. Entre los comportamientos más estudiados podemos encontrar la ocurrencia de segregación (agrupación de elementos similares en una mezcla), los bloqueos de flujos y la formación de arcos en silos y toberas.

Estudiar fenómenos que ocurren en los granulados, en fluidos o en sistemas complejos en general, trae consigo un análisis y seguimiento de modelos y una gran cantidad de datos; labores que un ser humano no puede realizar, ya que la tarea de vuelve insostenible. Es ahí donde entra las *simulaciones computacionales*, las cuales nos permiten estudiar fenómenos dado un modelo simplificado del problema.

Típicamente simular sistemas de muchas partículas requiere de un gran cantidad de trabajo y

poder computacional. Dado el gran trabajo que requieren, nace la necesidad de buscar técnicas que ayuden a agilizar dichos procesos. Ante esta necesidad los sistemas de computación paralela se muestran como una alternativa real al problema. La programación paralela no es una técnica nueva, y ha sido utilizada en aplicaciones industriales y científicas, como una forma de resolver problemas de forma más rápida durante muchos años.

En 2006 NVIDIA, uno de los mayores fabricantes de GPUs, introdujo CUDA[1], la cual es una arquitectura de computación de propósito general usando GPUs (*General-purpose GPU,* ***GPGPU***). Este incluye un nuevo modelo de programación paralela, y propone resolver problemas de forma más eficiente que con una CPU.

La mayoría de las tarjetas gráficas son diseñadas para juegos de computadoras. Los últimos modelos de este tipo de hardware han tenido un aumento considerable en las operaciones de punto flotante por segundo (FLOP), que llegan a ser 10 veces mayor en comparación a CPUs de alto rendimiento [21]. CUDA permite la utilización de las capacidades computacionales usadas generalmente para gráficos, ahora en aplicaciones como la resolución de problemas científicos, como por ejemplo, las simulaciones computacionales.

Esta memoria es acerca de cómo optimizar un simulador, en específico el desarrollado por el ahora Ingeniero Civil Informático Danilo Sandoval, cuya memoria de pregrado, titulado *Desarrollo de Software Simulador Híbirdo vía Dinámica Molecular para sistemas con interación Fluido-Grano* [25], trabajo en el que utiliza los métodos de: *dinámica conducida por eventos* para la simulación de granos duros; y *Stochastic Rotation Dynamics* para la simulación de fluidos.

El anterior trabajo y el presente, forman parte de un proyecto Fondecyt regular número 1120775 de 4 años de duración, el cual lo dirige el Dr. Dino Risso del Departamento de Física de la Universidad del Bío-Bío, y en el que participan como co-investigadores el Dr. Patricio Cordero y el Dr. Rodrigo Soto de la Universidad de Chile. El proyecto, denominado *Dynamics of confined granular fluids*, tiene como objetivo -entre otros- construir un esquema teórico en la forma de un modelo hidrodinámico que reproduzca exitosamente los resultados experimentales obtenidos del Laboratorio de Física No-Lineal que lidera el Dr. Nicolás Mujica de la Universidad de Chile. En general pretende estudiar tres tipos de situaciones:

[1] CUDA es un acrónimo para *Compute Unified Device Architecture*.

1. Sistemas someros (delgados) que son fluidizados por vibraciones de la caja que los contiene.

2. Sistemas densos que son fluidizados gravitacionalmente - este tipo de sistemas permite el estudio de avalanchas y el flujo granular a través de toberas.

3. Sistemas granulares fluidizados por viento.

En relación a los dos primeros, el Grupo de Teoría Cinética de la Universidad de Chile ha desarrollado soluciones optimizadas [18, 7, 5, 24]. Con respecto al último punto, Danilo Sandoval en [25] ofrece una solución -aunque no optimizada- al problema.

El presente trabajo se presenta como una optimización del trabajo realizado por Danilo Sandoval, implementando mejoras en base a la tecnología CUDA.

En los siguientes capítulos se revisarán: las bases de la computación paralela, la tecnología CUDA, y los métodos de simulación utilizados en el trabajo en el cual esta memoria se basa. Finalmente se extraen registros y conclusiones sobre la optimización realizada.

Capítulo 2

Computación Paralela

Tradicionalmente el desarrollo de software ha sido dominado por la computación serializada. El algoritmo construido para resolver un problema es implementado como una serie de instrucciones que son ejecutadas por la CPU. Solo una instrucción puede ser procesada a la vez, cuando una termina la siguiente es ejecutada.

Por otra parte, la computación paralela consta de varias unidades de procesamiento que trabajan simultáneamente para resolver un problema. Esto se logra mediante la división del problema en partes independientes para que así, cada unidad de procesamiento, pueda ejecutar su parte del algoritmo paralelamente junto a las otras unidades.

2.1. Tipos de paralelismo

La principal condición de la computación paralela, es que deben haber partes del problema que se puedan ejecutar de forma simultánea. Esto es el caso en que a lo menos dos partes de la computación no tengan dependencia de datos, es decir, que los datos de entrada de cada parte no dependan de los resultados de la otra parte.

En [16], los autores Lin y Snyder distinguen dos formas de paralelismo:

- Paralelismo basado en tareas.
- Paralelismo basado en datos.

2.1.1. Paralelismo basado en tareas

Algunos cómputos pueden ser separados en varias tareas, es decir, una secuencia de instrucciones independientes que trabajan sobre ítemes de datos independientes. Este tipo de paralelismo lo podemos encontrar típicamente en los sistemas operativos. Los procesos que se pueden ejecutar son diversos y muchas veces no se relacionan. Un usuario puede estar escribiendo un archivo en LaTeX mientras que escucha su lista de reproducción favorita en segundo plano. Dado que las tareas son independientes unas de las otras, estas pueden ser ejecutadas en paralelo.

2.1.2. Paralelismo basado en datos

La idea atrás del paralelismo basado en datos, es que en vez de concentrarse en qué tareas hay que ejecutar, primero hay que mirar los datos y como estos necesitan ser transformados. Como ejemplo, pensemos en que debamos ejecutar una transformación sobre cuatro arreglos separados, de similar tamaño y no relacionados. Para esto constamos con una CPU de 4 núcleos. En la aproximación basada en tareas, podríamos asignar cada arreglo a cada uno de los núcleos de la CPU. De esta forma la descomposición del problema es ejecutada pensando sobre las tareas a ejecutar y no sobre los datos a transformar.

En una aproximación basada en datos, la descomposición del problema se llevaría separando cada arreglo en cuatro bloques y cada uno sería asignado a un núcleo de la CPU. Una vez completado, los 3 arreglos restantes serían procesados de similar forma.

2.2. La taxonomía de Flynn

En [9], Flynn clasifica las arquitecturas de las computadoras con respecto a su habilidad de explotar el paralelismo con respecto al flujo de instrucciones (*instruction stream* en inglés), que es la secuencia de instrucciones que son ejecutadas en la computadora, y el flujo de datos (*data stream* en inglés), que es la secuencia de datos que son procesados por un flujo de instrucciones. La clasificación es la siguiente:

- **SIMD** - Single instruction, multiple data: La computadora ejecuta un solo flujo de instrucciones sobre un flujo múltiple de datos.

- **MIMD** - Multiple instructions, multiple data: Instrucciones de diferentes flujos de instrucciones son aplicadas a diferentes flujos de datos.
- **SISD** - Single instruction, single data: La computadora solo sigue un flujo de instrucciones que opera sobre solo un flujo de datos.
- **MISD** - Multiple instructions, single data: Varias instrucciones de diferentes flujos son ejecutados sobre el mismo flujo de datos.

La programación serializada sigue el modelo SISD, donde CPUs de un núcleo ejecutan una tarea a la vez, sin embargo, es posible ofrecer la ilusión de poder ejecutar múltiples tareas gracias al *time-slicing*, lo que permite cambiar entre tareas rápidamente.

Las actuales CPUs multi-núcleos siguen el modelo MIMD, los cuales disponen de un grupo de hilos o procesos que el sistema operativo asigna a uno de los N núcleos disponibles, donde cada hilo o proceso tiene un flujo de instrucciones independiente.

Podemos encontrar el modelo SIMD en el paralelismo basado en datos, se sigue un solo flujo de instrucciones sobre múltiples flujos de datos.

2.3. Programación Paralela

Dentro de la programación paralela existen dos aproximaciones basadas en el uso de memoria: *memoria compartida* y *memoria distribuida*. El modelo basado en memoria es de especial interés, ya que dicta las bases en como los procesos interactúan entre ellos y como el paralelismo debería ser implementado en orden de conseguir los mejores resultados.

2.3.1. Memoria compartida

El modelo de memoria compartida (*shared memory* en Inglés) se basa en reservar una región de memoria en común entre todas las instancias del programa en ejecución. A esta región de memoria, las instancias del programa pueden leer de o escribir datos.

Una de las APIs más conocidas en implementar el modelo de memoria compartida es OpenMP [1] (*Open Multi-Processing*). OpenMP define un set de directivas para los compiladores de C/C++ y Fortran. Estas directivas le dicen al compilador como distribuir el trabajo de una región

específica sobre una serie de hilos. Como los hilos son ejecutados en un mismo espacio de memoria, estos tienen accesos a todo los datos del proceso que los invoca -el proceso padre. Si OpenMP no se encuentra disponible, una alternativa son los hilos POSIX (*Portable Operating System Interface*) [11], los cuales son muy útiles para implementar aplicaciones paralelas basadas en tareas.

En el modelo de memoria compartida hay dos puntos en el que hay que preocuparse: La primera es asegurarse de que es "seguro" acceder a secciones de memoria, esto es evitar las condiciones de carrera, donde dos o más instancias pueden acceder y modificar una sección de memoria al mismo tiempo provocando un efecto inesperado en la aplicación. Lo segundo es que la escalabilidad del programa es limitado ya que las CPUs adicionales aumentan el estrés de carga en el bus de datos al mismo tiempo que el trafico de memoria compartida aumenta [19].

2.3.2. Memoria distribuida

El modelo de memoria distribuida (*distributed memory* en Inglés) o también conocido como modelo de paso de mensajes (*message passing* en Inglés), por lo general se utiliza en clusters, donde los nodos trabajan en un problema sobre una red.

Usualmente es implementado como una biblioteca que ofrece una interfase de funciones que pueden ser invocadas por los procesos que se quieren comunicar. Generalmente funciones para enviar y recibir mensajes están disponibles. Estas funciones pueden ser de bloqueo (el remitente del mensaje espera a que el *cartero* tome el mensaje,y el destinatario espera a que el *cartero* envíe el mensaje) o de no-bloqueo (el *cartero* recibe y envía el mensaje, mientras que el remitente y destinatario son libres de continuar con su trabajo).

La interfase más común para el modelo de paso de mensaje, en la computación de alto rendimiento, es el estándar MPI (Message Passing Interface). Este estándar define una interfase para el paso de mensajes en los lenguajes C y Fortran. Implementaciones de este estándar son Open MPI [27] y MPICH2 [13].

2.3.3. Conceptos de la programación paralela

En esta sección se introducirá conceptos claves sobre arquitecturas paralelas y consideraciones varias.

Arquitecturas clásicas de la programación paralela

En la computación paralela hay varias técnicas que nos pueden ayudar a resolver un problema específico. Alguna de estas técnicas pueden ser más eficientes que otras dependiendo de nuestros requerimientos. A continuación se da revista a cuatro de las técnicas más importantes [4].

Maestro - esclavo En este tipo de diseño existe un proceso maestro que descompone el problema en pequeñas tareas que son asignadas a otros procesos, estos últimos son llamados esclavos. Las tareas son procesadas hasta cierto punto y luego se devuelve el resultado al proceso maestro, este último termina el procesamiento para obtener el resultado final. Con este diseño se pueden alcanzar un alto rendimiento pero se corre el riesgo de producir un cuello de botella por medio de la comunicación de todos los esclavos con el maestro.

Dividir para conquistar Es el típico diseño utilizado en la recursividad, donde un problema se divide en dos subproblemas y estos pueden volver a ser divididos. Los resultados obtenidos en cada división son combinados por el proceso que los dividió, para finalmente resolver el problema del proceso principal.

Pipelining Es un diseño donde dos o más procesos se combinan para resolver un problema, en donde la salida de un proceso es la entrada del otro.

Single Program, Multiple Data (SPMD) Esta técnica se aprovecha de los procesos que hacen una misma tarea sobre diferentes datos. Es el diseño mas común y se asocia al modelo SIMD visto anteriormente.

Conceptos

Hay una serie de conceptos que se encuentran al trabajar en programación paralela. A continuación se presentan algunos de estos junto a sus atributos:

Fork/join: Cuando un proceso genera un número de subprocesos es llamado *fork*. El programa ejecuta un flujo de instrucciones secuencialmente hasta que llega un punto donde el trabajo lo distribuye en N subprocesos que efectúan cálculos en paralelo. Una vez ejecutados

los cálculos estos convergen retornando al proceso padre, una vez retornado la ejecución secuencial continúa. Este último paso se le denomina *join*.

Sincronización: Es posible que los hilos o procesos que trabajan paralelamente vayan en diferentes velocidades y orden, por lo que aveces es necesario implementar sistemas de sincronización. Los sistemas de sincronización permiten que todos los hilos o subprocesos se ejecuten hasta un punto específico y quedan en modo de espera. Una vez que todos los procesos o hilos llegan a ese punto de espera, se les está permitido continuar con su trabajo. Las razones para utilizar la sincronización pueden ser variadas, pero muchas veces se utiliza para sincronizar el acceso a datos que son utilizados por otros hilos.

Paralelización de loops: Una de las primeras partes y más fáciles de paralelizar son los loops. Esta tarea es un ejemplo típico del diseño SPMD. El programador debe asegurarse que los índices -en el caso de trabajar con un arreglo de datos- son divididos entre los hilos disponibles.

Reducción: Es el proceso donde el conjunto de entrada de datos se procesan paralelamente para producir un conjunto de tamaño reducido.

2.4. Programación sobre GPU para propósito general

Las GPUs (*Graphics Processing Units*) son unidades de procesamiento altamente optimizadas para el procesamiento de gráficos 3D. Desde la década de los 90's las tarjetas gráficas se han convertido en un gran mercado gracias a los juegos. Gracias a la alta competitividad del mercado, el desarrollo de las tarjetas ha aumentado en su poder de cálculo y ha bajado en sus precios.

Ya en los 90's el poder de las GPUs se utilizaba en otras áreas más que para la computación gráfica [15]. En 2003 Mark Harris [12] propuso el término **GPGPU** (*General-purpose GPU*) para el uso de propósito general de las tarjetas gráficas. Al inicio, programar sobre GPUs era muy tedioso, los algoritmos tenían que ser implementados a través de instrucciones de tipo *shaders*[1]. La programación de propósito general fue mejorada cuando NVIDIA introduce una nueva arquitectura de instrucciones shaders unificada en el hardware, esto fue con la introducción de las tarjetas

[1] Un shader es un programa utilizado para la producción de niveles de luz y color apropiados en una imagen.

gráficas GeForce 8 Series en noviembre de 2006. El gran cambio que hizo NVIDIA fue desarrollar una arquitectura multi-núcleo implementando las instrucciones de tipo shaders via software. La nueva arquitectura de NVIDIA se llama **CUDA** (*Compute Unified Device Architecture*). En 2007, con las tarjetas Radeon R600 series, AMD lanzó una arquitectura unificada de shaders para trabajar sobre GPUs.

Para el desarrollo sobre GPUs AMD y NVIDIA ofrecen SDKs (software development kits) que dan acceso a la GPU desde el programa anfitrión que se está ejecutando sobre la CPU. Inicialmente ambos fabricantes usaban versiones modificadas del lenguaje de programación C, NVIDIA ofrecía el lenguaje CUDA-C, y AMD usaba Brook+, una versión mejorada de BrookGPU [3]. Actualmente ambos fabricantes ofrecen soporte para OpenCL [22]. OpenCL es una especificación de un framework de programación que consiste de un lenguaje de programación y una API, este facilita la tarea de escribir programas que se ejecuten en paralelo sobre ambientes computacionales heterogéneos que consistan, por ejemplo, de CPUs multi-núcleos con GPUs.

Capítulo 3

CUDA

Compute Unified Device Architecture (CUDA), es la plataforma desarrollada por NVIDIA para el desarrollo de computación paralela sobre GPUs. Como se explicó en apartados anteriores, CUDA se basa en la tecnología de las tarjetas gráficas. El mismo término GPU fue acuñado por NVIDIA en 1999 [20].

En este capítulo se dará revista a las características del hardware y se describirán conceptos claves para entender el modelo de programación bajo CUDA.

3.1. Introducción

CUDA C es diseñado para la computación paralela, y permite al programador manejar y programar grandes cantidades de hilos sobre GPUs compatibles. La idea atrás de la programación en CUDA es que los programas están estructurados de tal forma de que se ejecutan sobre la CPU, mientras que funciones específicas son ejecutadas en hilos sobre la GPU. El código que se ejecuta en la CPU se denomina código anfitrión (*host code*). El código anfitrión maneja las funciones que serán ejecutadas sobre la GPU, estas funciones se denominan *kernels*, en CUDA el código que se ejecuta en la GPU se denomina como *device code*. Un kernel puede ser lanzado utilizando miles o incluso millones de hilos “livianos” que se ejecutan sobre el dispositivo. Los hilos en CUDA se dicen que son livianos por las siguientes razones:

Un hilo en CUDA tiene prácticamente cero gasto en su creación, por lo que miles de hilos pueden crearse rápidamente. A bajo nivel, CUDA maneja los hilos con aceleración por hardware.

El programador es incentivado a “pensar en grande” para que maneje libremente los hilos que necesita. Mientras más hilos se utilicen, la posibilidad de esconder la latencia aumenta, esto es debido a la ganancia de rendimiento que ofrece el hardware.

Dentro del kernel, el mismo código es ejecutado en todos los hilos. Veamos el siguiente ejemplo que suma dos vectores:

```
int v1[3] = {1, 2, 3}, v2[3] = {4, 5, 6}, v3[3];
for (int i = 0; i < 3; ++i)
    v3[i] = v1[i] + v2[i];
```

En CUDA, ejecutar el código anterior provocaría que cada hilo ejecute el mismo loop, provocando sobre-escritura de los mismo resultados. La forma de ejecutar el código anterior en paralelo se alcanza asignándole a cada hilo un identificador único. El identificador puede ser utilizado para decidir qué trabajo debe ejecutar el hilo. En programas creados de forma eficiente en CUDA, los datos son ordenados de tal forma que los hilos pueden trabajar sobre ellos directamente. Un ejemplo de como un código en CUDA C implementaría el ejemplo anterior es el siguiente:

```
int v1[3] = {1, 2, 3}, v2[3] = {4, 5, 6}, v3[3];
id = threadIdx.x;
if (id < 3)
    v3[id] = v1[id] + v2[id];
```

El loop ha sido removido y cada hilo con identificador menor a 3 contribuye con su parte en la suma de vectores.

La GPU consiste en varios multiprocesadores, cada uno contiene un número de núcleos que ejecutan los hilos en paralelo. Los programas hechos en CUDA son portables entre las diferentes versiones de GPUs[1] y sin tener en cuenta el número de multiprocesadores o núcleos que estos posean, se logran ejecutar el mismo número de hilos. Para facilitar esto, CUDA ofrece un nivel de

[1] La portabilidad que proporciona CUDA es hacia atrás, las nuevas versiones soportan las características presentes en versiones previas, pero no al revés.

abstracción: los hilos se dividen conceptualmente en *bloques*, cada bloque de un kernel ejecuta el mismo numero de hilos; y el número total de bloques se ordenan en una *grilla*. La grilla consiste en todos los bloques, y por lo tanto, todos los hilos del kernel (Figura 3.1) [21].

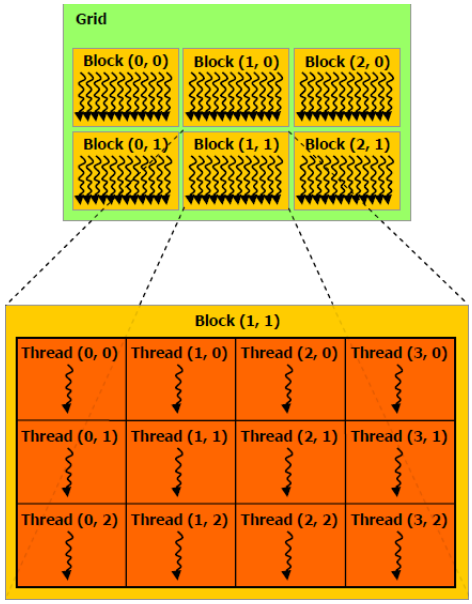


Figura 3.1: La grilla se divide en bloques los cuales contienen un número de hilos [21].

CUDA opera con una unidad de ejecución llamada *warp*. Un warp consiste en 32 hilos, cuyas instrucciones están siendo ejecutadas simultáneamente en el multiprocesador. Si uno o más hilos ejecutan secciones de código diferentes a los demás hilos dentro del warp (ya sea por medio de una bifurcación), estos se ejecutarán de forma serializada. Este fenómeno se conoce como divergencia de hilos.

3.2. Hardware CUDA

Hardware compatible con CUDA se puede encontrar en las GPUs que NVIDIA ha lanzado al mercado desde la introducción de la arquitectura G80. Este tipo de GPUs son comúnmente encontradas en equipos de medio y alto rendimiento, haciéndolos de cierta forma ubicuos. Se pueden encontrar versiones que están integradas a las tarjetas madres, pero estas comparten la memoria principal del sistema, por lo que su rendimiento es poco deseable. Otras versiones -de medio y alto rendimiento- vienen en forma de tarjetas, las cuales pueden ser insertadas dentro de

sistemas, y dependiendo del tipo de sistema, varias tarjetas pueden ser insertadas a la vez.

El hardware de una GPU es muy diferente al hardware de una CPU. La figura 3.2 esquematiza como se ve un sistema multi-GPU que es conectada en un bus PCI-E. En el diagrama se pueden identificar los siguientes componentes:

- Memoria (global, constante y compartida).
- Streaming multiprocessors (SMs).
- Streaming processors (SPs).

Hay que notar que la GPU realmente es un arreglo de SMs, la cual cada una posee N núcleos (Figura 3.3). Una GPU consiste en uno o más SMs. Si miramos en detalle, veremos que son varios los componentes que forman un SM. Lo más significante es que cada SM posee varios SPs. En la Figura 3.3 se muestran 8 SPs, pero en arquitecturas como Fermi[2], es posible encontrar entre 32-48 SPs y en la arquitectura Kepler hasta 192. Características que nos hacen pensar que en futuras versiones irá aumentando la razón entre SPs/SMs.

Cada SM tiene acceso a archivos de registros, esta es una memoria que es utilizada para guardar los registros en uso dentro de cada hilo corriendo en un SP, la velocidad de esta memoria es casi la misma que la velocidad de cada unidad SP, por lo que la espera es casi cero. En cada SM también encontramos un bloque de memoria compartida que solo tienen acceso los SPs que integran el SM; esta memoria puede ser utilizada como una cache manejada a nivel de software. A diferencia de la cache en CPU, esta memoria esta bajo el control del programador.

Cada SM tiene acceso a la memoria global, constante y de textura por medio de buses separados; la memoria de textura es una vista de la memoria global, la cual es útil para datos que quieran ser interpolados; la memoria constante es utilizada como una memoria de solo lectura, esta memoria igual es una vista de la memoria global. La memoria global es proporcionada por una versión especial de la memoria DDR (*Double Data Rate*), la GDDR (*Graphic Double Data Rate*). El bus puede ser de hasta 512 bits de ancho (256 bits en Fermi), con un ancho de banda de hasta 190 GB/s. Las Unidades de Propósito Especial (SPUs) son otros componentes que se pueden encontrar en cada SM, estas unidades se encargan de ejecutar instrucciones especiales en el hardware (como las operaciones de seno/coseno/exponencial).

[2]Arquitectura presente en las tarjeta Nvidia Quadro 4000.

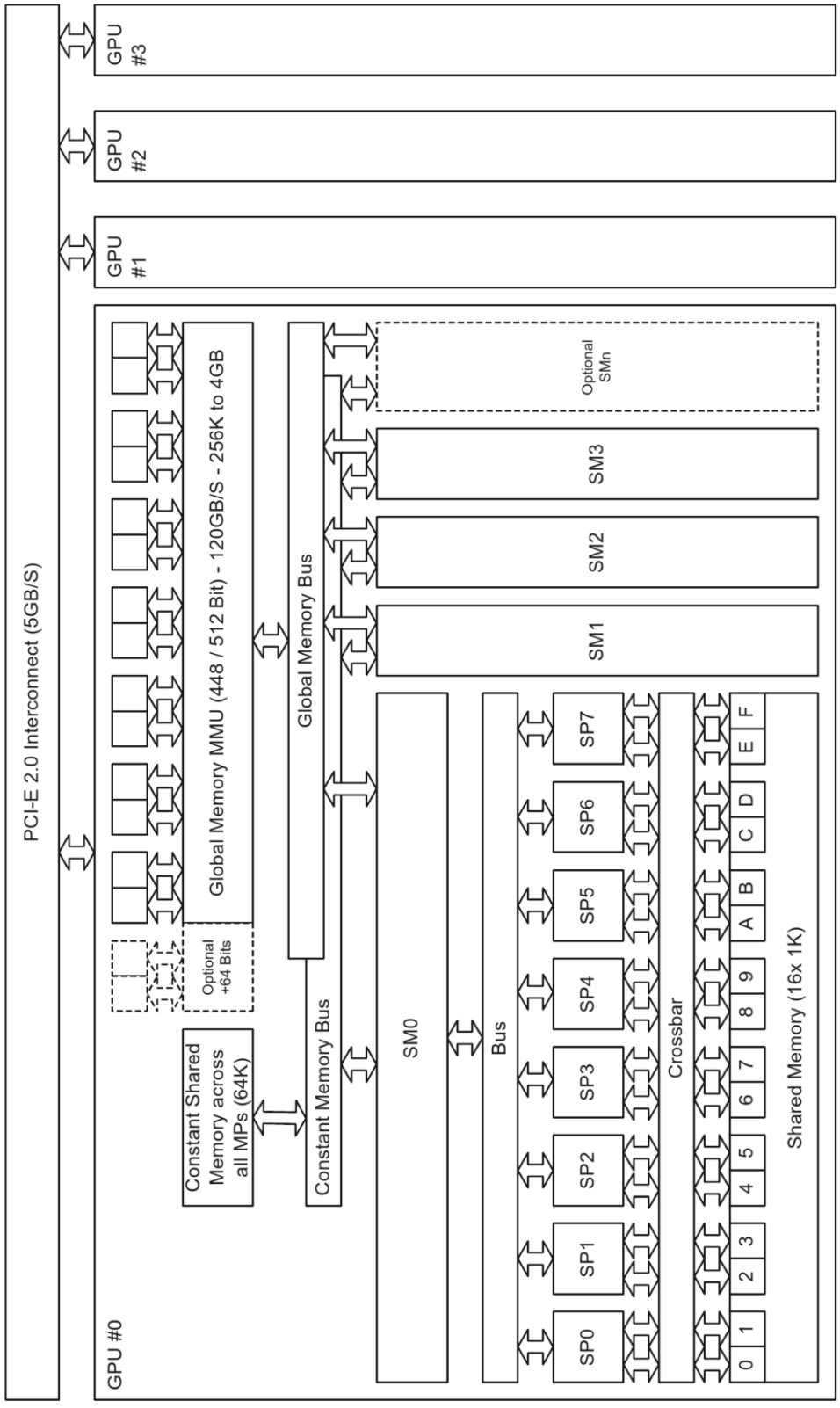


Figura 3.2: Diagrama de una GPU (G80/GT200) [6].

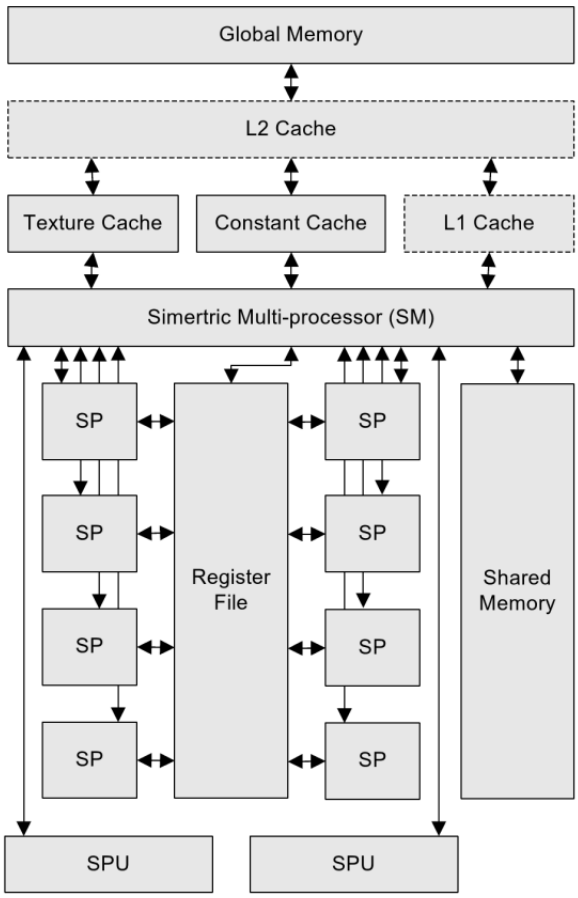


Figura 3.3: Diagrama de un SM [6].

3.3. Niveles de cómputo

A menudo que se liberan nuevas GPUs, nuevas capacidades y funcionalidades aparecen. Los programas pueden ser escritos para que aprovechen estas nuevas ventajas. CUDA viene en varias versiones, cada cual está ligada a diferentes generaciones de tarjetas gráficas. La serie G80 de tarjetas gráficas venía con la primera versión de CUDA. Cada generación de GPUs tiene un número identificador denominado *CUDA compute capability*. Al momento de escritura de este documento, existen 6 versiones de CUDA compute capability: 1.0, 1.1, 1.2, 1.3, 2.0, 2.1.

Una lista de las diferencias entre cada versión puede ser encontrada en el Apéndice G de [21], el cual viene distribuído como parte del CUDA SDK. A continuación se señalarán las diferencias más importantes entre las versiones 1.3 y 2.0:

- Mayores operaciones atómicas y más rápidas.
- Más memoria compartida y la posibilidad de configurar razones entre cache L1 y memoria compartida.
- Incremento en la velocidad con operaciones de punto flotante.
- Incremento en el numero de registros e hilos por cada SM.
- Incremento en la precisión de ciertas operaciones de punto flotante.

3.4. Programación paralela en CUDA

Por defecto CUDA soporta los lenguajes de programación C/C++ y Fortran. En el presente trabajo solo se enfocará en la implementación de CUDA C. Esta sección pretende describir conceptos generales, extensiones, y algunos usos de la API de CUDA C.

3.4.1. CUDA C

CUDA C es una extensión al lenguaje de programación C. Estas extensiones permiten el uso del dispositivo, aunque con algunas restricciones comparado con el estándar de C. Algunas de estas restricciones solo se encuentran en versiones de CUDA compute capability v1.x y no aplican a versiones iguales o superiores a la v2.0.

Calificadores de funciones

Calificadores de funciones adicionales son usados para especificar si las funciones deberían ser compiladas para GPU o CPU.

El calificador **__global__** identifica una función de GPU que es accesible desde y solo desde el programa anfitrión (CPU). Las funciones globales solo pueden retornar un tipo `void` y no pueden ser usados recursivamente. Los parámetros pueden ser pasados del anfitrión hacia el dispositivo. Punteros a funciones globales pueden ser utilizados solamente del lado del anfitrión. Cuando la función global es invocada, esta debe tener una configuración de lanzamiento especial. Estas funciones son los llamados kernels.

El calificador __**device**__ identifica a una función de GPU que es accesible solamente desde otras funciones de GPU. Desde versiones 2.0 estas funciones pueden ser llamadas recursivamente.

El calificador __**host**__ indica a una función que es compilada y ejecutada solo por la CPU. Si una función se define sin calificador, se asume que es de este tipo.

Calificador de variables

Al igual que las funciones, CUDA ofrece calificadores para indicar los tipos de variables, los cuales difieren dependiendo del ámbito y el uso. Estos no pueden ser utilizados dentro de estructuras o en variables locales dentro de funciones __host__.

El calificador __**device**__ indica una variable que estará almacenada en la memoria global de la GPU. Este tipo de variables es accesible directamente desde funciones de GPU o por medio del anfitrión usando funciones especiales ofrecidas por CUDA. Estas variables duran todo el tiempo que el programa dura.

El calificador __**constant**__ indica una variable que esta almacenada en la memoria constante de la GPU. Su duración es por toda la vida del programa. Una restricción de este tipo de variables es que solo puede ser escrita por funciones especiales de CUDA llamadas desde código anfitrión.

El calificador __**shared**__ indica una variable que esta almacenada en la memoria compartida de la GPU. Una variable en la memoria compartida tiene un periodo de vida de un bloque, y es solamente visible por los hilos pertenecientes a un bloque.

3.4.2. Grillas, Bloques e Hilos

Cada kernel es ejecutado en una grilla de hilos. Esta grilla es dividida en bloques de hilos y cada bloque es divido en hilos. En la Figura 3.4 se observa un ejemplo de una grilla que es divida en nueve bloques de hilos (3x3), cada bloque es también dividido por nueve hilos (3x3), lo que da un total de 81 hilos en la grilla. Si bien la imagen muestra una grilla de 2 dimensiones, tarjetas con Compute Capability mayor o igual a 2.0, soportan grillas de hasta 3 dimensiones. En cambio los bloques de hilos permiten ser configurados hasta 3 dimensiones sin importar que Compute Capability se posea.

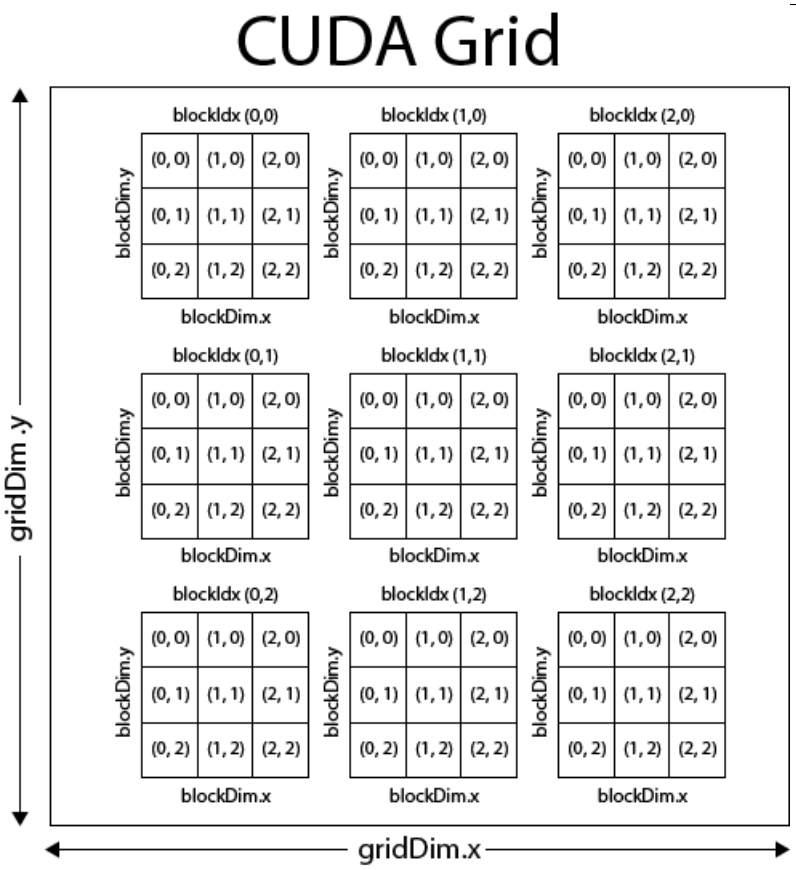


Figura 3.4: Grilla dividida en 9 bloques, los que a la vez están divididos en 9 hilos (81 hilos en total).

La forma básica para invocar un kernel, utiliza la siguiente sintaxis:

```
kernel<<<bloques, hilos>>>(argumento1, argumento2, ...)
```

El parámetro `hilos` es el número de hilos que se lanzarán, este se define con el tipo de dato `dim3` el cual permite definir valores con 3 dimensiones. El parámetro `bloques` es el número de bloques que se lanzarán, al igual que con el número de hilos, este es de tipo `dim3`.

El número de bloques dentro de una grilla puede ser determinada dentro de un kernel utilizando la variable `gridDim` y para determinar el número de hilos dentro de un bloque, se puede utilizar la variable `blockDim`. Cada bloque de hilos es únicamente identificado dentro de cada kernel usando la variable `blockIdx` y cada hilo perteneciente a un bloque, puede ser identificado usando la variable `threadIdx`. Cada una de estas variables -`gridDim, blockDim, blockIdx, threadIdx`-

son estructuras de componentes `.x, .y, .z`.

Cuando un kernel es invocado, este es ejecutado en la GPU por mucho hilos en paralelo. Cuando es lanzado el kernel, las variables `gridDim, blockDim, blockIdx, threadIdx` son inicializadas. En el Listado 3.1 se utiliza la variable `threadIdx`, la cual almacena el índice de cada hilo en ejecución. En el ejemplo se utiliza esta variable para leer de y escribir a una dirección de memoria en particular al utilizar los índices de los hilos como forma de acceder a los valores de un arreglo.

Listing 3.1: Invocación de un kernel.

```
__global__ void cuadrado (float *a) /* definicion del kernel */
{
    int idx = threadIdx.x;
    float v = a[idx];
    a[idx] = v * v;
}

int main (void)
{
    float *h_a, *d_a;
    int numVal = 256;
    h_a = (float *) malloc (numVal * sizeof (float));
    cudaMalloc ((void **)&d_a, numVal * sizeof (float));

    /* Inicializar h_a
     * ...              */

    cudaMemcpy (d_a, h_a, numVal * sizeof (float), cudaMemcpyHostToDevice);
    dim3 hilos (numVal, 1, 1);
    dim3 bloques (1, 1, 1);

    miKernel<<<bloques, hilos>>> (d_a);
    cudadMemcpy (h_a, d_a, numVal * sizeof (foat), cudaMemcpyDeviceToHost);

    /* Utilizar resultados
     * ...               */
```

```
    free (h_a);
    cudaFree (d_a);
}
```

Hay dos características del modelo de ejecución de tareas en GPUs. La primera es que por cada grupo de *Stream Processors* (SPs), las tareas que se ejecutan se hacen bajo un esquema de paso-a-paso, permitiendo ejecutar una misma tarea sobre diferentes datos (Figura 3.5). Esto significa que cada instrucción en la cola de instrucciones es enviada a cada SP dentro de un SM. La segunda característica es que debido al gran tamaño del archivo de registros, el cambio de contexto entre hilos significa una sobrecarga casi cero.

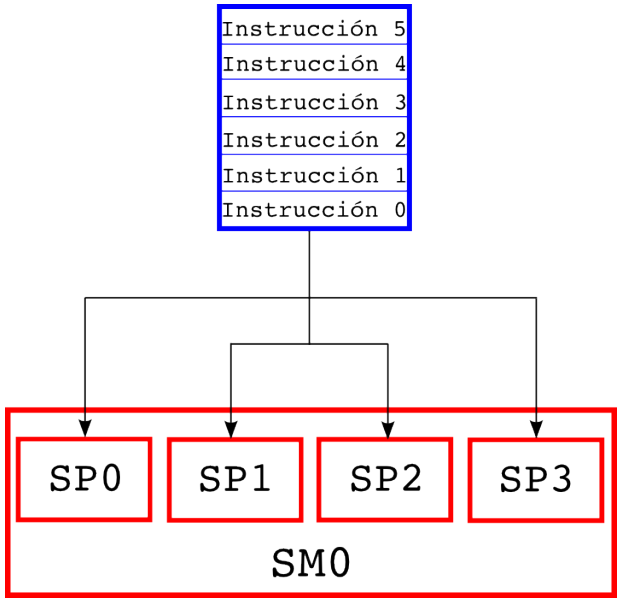


Figura 3.5: Esquema de ejecución paso-a-paso.

Capítulo 4

Simulaciones Computacionales

Realizar experimentación sobre ciertos fenómenos permite generar conocimiento sobre el objetivo de estudio con el fin de analizar sucesos del pasado y preveer sucesos del futuro.

Lo que diferencia la simulación computacional en general con otras formas de computación, si es que tal distinción se puede hacer, es la manera en la que el computador es usado: más allá de ejecutar cálculos, el computador se convierte en un laboratorio virtual en donde un sistema es estudiado - un experimento numérico [23].

En este capítulo se revisarán los métodos utilizados por Danilo Sandoval en [25] para el desarrollo del software de simulación fluido-grano.

4.1. Dinámica Conducida por Eventos

Este tipo de técnica de simulación es ideal para sistemas de partículas que interactúan vía potenciales discretos (dinámica de partículas duras), la técnica se preocupa de determinar para cada evento (una colisión por ejemplo): el instante en que éste ocurre -cronológicamente-; los objetos que intervienen; y las acciones asociadas al evento.

A diferencia de la Dinámica Molecular (DM), la cual evoluciona en pasos temporales regulares, la Dinámica Conducida por Eventos (DCE) evoluciona a través del salto desde la configuración del evento actual hacia la configuración del evento siguiente. Por lo tanto, los intervalos de tiempo por los cuales la simulación avanza, es determinada por la propia dinámica de la simulación.

4.1.1. Algoritmo de Simulación

El algoritmo de simulación conducida por eventos se puede esquematizar en dos pasos:

1. Determinar cuando ocurren las posibles colisiones (u otros eventos).
2. Actualizar el estado del sistema luego de cada colisión (o evento).

Detección de posibles colisiones

Para determinar la colisión entre dos partículas, hay que tener en cuenta que éstas: (a) Se están acercando; y (b) si acaso en su trayectoria se interceptan. En un sistema en ausencia de gravedad, la evolución libre está dada por:

$$\vec{r}_t(t) = \vec{r}_i(0) + \vec{v}_i(0)t$$

En donde $\vec{r}_i(0)$ y $\vec{v}_i(0)$ se refieren a la posición y velocidad iniciales. El instante de tiempo t para que ocurra la colisión entre dos partículas se determina exigiendo que se cumpla lo siguiente:

$$\|\vec{r}_2(t) - \vec{r}_1(t)\| = \sigma$$

Donde σ es la suma de los radios de ambas partículas.

A partir de:

$$\|\vec{r}_2(t) - \vec{r}_1(t)\|^2 = \sigma^2$$

Y desarrollando:

$$\left\| \underbrace{\vec{r}_2(0) - \vec{r}_1(0)}_{\vec{r}_{21}} + \underbrace{(\vec{v}_2(0) - \vec{v}_1(0))t}_{\vec{v}_{21}} \right\|^2 = \sigma^2$$

Resulta:

$$r_{21}^2 + 2(\vec{r}_{21} \cdot \vec{v}_{21}t) + v_{21}^2 t^2 = \sigma^2$$

Se obtiene una ecuación de segundo grado, la cual resolviéndola obtenemos el instante de colisión t:

$$v_{21}^2 t^2 + 2(\vec{r}_{21} \cdot \vec{v}_{21} t) + (r_{21}^2 - \sigma^2) = 0$$

$$t = \frac{-(\vec{r}_{21} \cdot \vec{v}_{21}) \pm \sqrt{(\vec{r}_{21} \cdot \vec{v}_{21})^2 - v_{21}^2 (r_{21}^2 - \sigma^2)}}{v_{21}^2}$$

Se encuentran dos soluciones que corresponden a: (1) al evento más próximo en el tiempo $t_{(-)}$; y (2) al evento más lejano en el tiempo $t_{(+)}$. Este último resultado no tiene interés físico por que significa que las partículas pudieran atravesarse sin interactuar.

$$t = \frac{-(\vec{r}_{21} \cdot \vec{v}_{21}) - \sqrt{(\vec{r}_{21} \cdot \vec{v}_{21})^2 - v_{21}^2 (r_{21}^2 - \sigma^2)}}{v_{21}^2}$$

Cambio de estado del sistema luego de cada colisión

Una vez predicho el momento de la colisión, se debe actualizar el estado del sistema hacia el estado mismo de colisión, esto se alcanza actualizando las posiciones y velocidades de las partículas involucradas en la colisión.

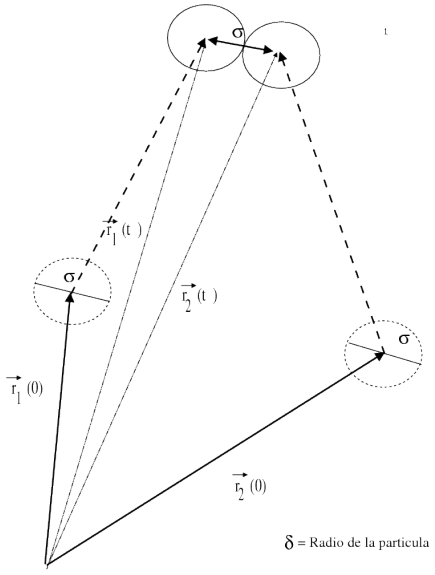


Figura 4.1: Momento en que el estado de las partículas se actualizan hasta la situación de colisión.

Eficiencia del algoritmo

La estrategia desarrollada por Marín, Risso y Cordero en [18] consiste en utilizar dos algoritmos para tratar la Lista de Eventos Futuros, denominados como *Estados Retardados* y *Mínimos Locales*. El algoritmo de estados retardados propone añadir al estado de las partículas el tiempo de la última colisión en las que estuvieron involucradas, de esta forma, para evolucionar el sistema no se avanzan todas las partículas, sino sólo aquellas que estuvieron involucradas en el choque. El algoritmo de mínimos locales propone crear, para cada partícula, una lista de eventos futuros, cada lista de eventos posee un mínimo local (el evento más próximo en ocurrir); Cada mínimo local es situado como una hoja final de un árbol binario completo, finalmente cada mínimo local compite con un par, el mínimo local de menor tiempo asciende en el árbol, trasformándose en el mínimo global y próximo evento de simulación.

4.2. Stochastic Rotation Dynamics

La simulación *Stochastic Rotation Dynamics* (SRD) -también conocida como *Multiple Particle Collision Dynamics (MPC)*- es un método que fue originalmente propuesto en [17]. El método emula el comportamiento mesoscópico de un fluido. En el algoritmo de simulación hay esencialmente dos etapas: (a) La traslación libre de las moléculas (*streaming*); y (b) la interacción de choque entre ellas. El método avanza haciendo un ciclo donde las moléculas -representadas por un modelo de pseudo-partículas- experimentan una interacción libre, seguido luego de una "colisión" de las pseudo partículas en todas las celdas.

A nivel mesoscópico, la interacción que produce la colisión conserva en cada celda de la grilla de simulación la masa, el momentum y la energía. De esta forma se asegura que a nivel macroscópico se satisfagan las ecuaciones de balance de masa (continuidad), balance de momentum y balance energía; es decir, da como resultado un comportamiento que puede ser modelado por las ecuaciones más generales de la hidrodinámica [14].

4.2.1. Streaming

La primera parte de este método consiste en el avance libre de las pseudo-partículas. Esto se logra haciendo avanzar las pseudo-partículas a través del tiempo. En la forma más básica, el

streaming está dado por:

$$\vec{x}_i(t+\tau) = \vec{x}_i(t) + \vec{v}_i(t+\tau)\tau$$

Donde t es el tiempo, y τ es el tiempo que dura cada paso de la dinámica.

Durante este paso, fuerzas adicionales pueden ser aplicadas. Por ejemplo, una aceleración gravitacional puede ser aplicada, para simular un fluido bajo la acción de la gravedad.

4.2.2. Regla de choque

Para colisionar las pseudo-partículas, una grilla de celdas es distribuida sobre la caja de simulación. Todas las pseudo-partículas se asignan a las celdas correspondiente, y el momentum total es sumado para cada celda. La colisión es una rotación de las velocidades de todas las pseudo-partículas en cada celda, relativa a la velocidad de centro de masa $\vec{u}$ de cada celda. Esto es hecho como una multiplicación de matrices de una matriz de rotación **R** para un ángulo α sobre el vector de velocidad relativo $\vec{v} - \vec{u}$, y es dado por:

$$\vec{v}(t+\tau) = \mathbf{R}(\vec{v}(t) - \vec{u}(t)) + \vec{u}(t)$$

Donde $\vec{u}$ es dado por:

$$\vec{u} = \frac{\sum_{i=1}^{M'} m_i \vec{v}_i}{m_t}$$

M' es el número de pseudo-partículas de una celda en un tiempo dado. m_t es la masa total de la celda, que está dada por:

$$m_t = \sum_{i=1}^{M'} m_i$$

- En el caso 2 dimensional **R** está dado por:

$$\mathbf{R} = \begin{pmatrix} \cos\alpha & \pm\sin\alpha \\ \pm\sin\alpha & \cos\alpha \end{pmatrix}$$

- En el caso 3 dimensional **R** está dado por:

$$\mathbf{R} = \begin{pmatrix} \cos\alpha + u_x^2(1-\cos\alpha) & u_x u_y(1-\cos\alpha) - u_z\sin\alpha & u_x u_z(1-\cos\alpha) + u_y\sin\alpha \\ u_y u_x(1-\cos\alpha) + u_z\sin\alpha & \cos\alpha + u_y^2(1-\cos\alpha) & u_y u_z(1-\cos\alpha) - u_x\sin\alpha \\ u_z u_x(1-\cos\alpha) - u_y\sin\alpha & u_z u_y(1-\cos\alpha) + u_x\sin\alpha & \cos\alpha + u_z^2(1-\cos\alpha) \end{pmatrix}$$

Dado que la matriz de rotación **R** es una matriz ortogonal, se satisface:

$$\mathbf{R}\mathbf{R}^{-1} = \mathbf{R}\mathbf{R}^{T} = \mathbf{I}$$

Para asegurar un balance detallado, la dirección de rotación es escogida de forma azarosa para cada celda. Esto es implementado escogiendo azarosamente un signo mas o menos en **R**.

El detalle de qué tipo de fluido se está modelando, tiene directa relación con los parámetros de: tamaño de la celda, el tiempo de streaming y las regla de choque entre las pseudo-partículas.

Este método a resultado apropiado para simular sistemas donde se forman remolinos detrás de un obstáculo cuando pasa un fluido. Incluso se ha usado para estudiar el movimiento de propulsión de los peces [10].

Capítulo 5

Implementación

El objetivo principal del cual nace este proyecto, es apoyar al *Grupo de Teoría Cinética*[1] -de la Universidad de Chile y la Universidad del Bío-Bío- en su trabajo de simulaciones, mediante la creación de un software de simulación híbrido, que implemente dinámicas moleculares conducidas por eventos y por tiempo que permitan estudiar las interacciones fluido-grano.

Para alcanzar el objetivo principal, se crearon tres objetivos específicos relacionados a continuar el desarrollo hecho en [25]:

- Revisar y solucionar problemas asociados al software de simulación desarrollado en [25].
- Implementar una mayor paralelización en el simulador utilizando CUDA.
- Finalizar implementación de la fluidización granular por viento.

Como consecuencia de la revisión del desarrollo hecho en [25], se decidió comenzar un nuevo proyecto[2]. A este nuevo desarrollo se le denominó *Adentu*, vocablo *Mapudungun* que significa modelo.

Adentu fue desarrollado como un conjunto de bibliotecas escritas en C y CUDA C, que proveen una API y las estructuras de datos necesarias para poder implementar simulaciones moleculares para sistemas fluido-grano. Además, Adentu fue licenciado con GPLv3[3], lo que significa que cualquier persona puede ayudar al mejoramiento de este.

[1] `http://www.dfi.uchile.cl/cinetica`

[2] Si bien es un proyecto complétamente nuevo, se heredan los algoritmos para los cálculos físicos de [25].

[3] `https://www.gnu.org/licenses/gpl-3.0.html`

En el desarrollo de este proyecto se utilizaron varias herramientas; Como se mencionó anteriormente los lenguajes de programación utilizados son C[4] y CUDA C. Para llevar un control sobre el desarrollo, se utilizo *Git*[5] como software de control de versiones. Utilizar Git significó poder implementar ideas y características de tal forma que el código estable nunca se viera comprometido, usando además el servicio que provee GitHub[6], el cual se utiliza como repositorio oficial del proyecto[7]. También se utilizaron las bibliotecas GLib[8] con la que se implementa listas enlazadas y otras estructuras, y GLUT[9]/FreeGlut[10] con las que se implementa la visualización gráfica.

5.1. Resumen de las características implementadas

Como base, Adentu debe contener las mismas características que se implementaron en el desarrollo anterior hecho en [25]. Adentu implementa dichas características, siendo capaz además, de agregar varias otras, todas desarrolladas tomando en cuenta la flexibilidad y el potencial que el cálculo en paralelo con CUDA proporciona.

Una de las características esenciales en Adentu es el sistema de partículas, esto permite tener sistema de miles partículas de grano y fluido, cada una con sus propias características, es decir, lar partículas pueden ser de diferentes tamaños o diferente masa por ejemplo.

Para poder mantener un software lo suficientemente flexible, es necesario convertir cada componente de Adentu en un módulo. Los módulos implementado han sido desarrollados con una mínima interdependencia, permitiendo utilizar cada uno de forma autónoma y junto a código de terceros.

Adentu consta de un motor de eventos que permite activar y desactivar eventos que el usuario quiera tener en la simulación. Debido a la modularidad con la que se cuenta, es posible agregar nuevos eventos de forma sencilla. Además, el motor de eventos es lo suficientemente flexible para integrar eventos relacionados con dinámicas moleculares conducidas por tiempo, por lo que implementar simulaciones híbridas -que hacen uso de simulaciones conducidas por eventos y

[4] Se desarrolla utilizando características del estándar *C99*.

[5] `http://git-scm.com/`.

[6] `http://www.github.com`.

[7] `https://github.com/crosvera/adentu`

[8] `https://developer.gnome.org/glib/`.

[9] `https://www.opengl.org/resources/libraries/glut/`.

[10] `http://sourceforge.net/projects/freeglut/`.

tiempo- sea más simple. Entre los eventos que se incluyen en Adentu encontramos: eventos por condiciones de borde (periódico, bounce-back, reflectante y fluidizado.); eventos de colisión grano-grano y grano-fluido; y eventos MPC.

5.1.1. Condiciones de borde

Una condición de borde corresponde a un evento en el cual una partícula de grano o de fluido alcanza el borde del espacio de simulación; para evitar que la partícula escape del sistema, existen varias aproximaciones para controlar este suceso y que describen el comportamiento que deben tomar las partículas ante este suceso, en Adentu existen las siguientes condiciones de bordes: condición de borde periódico, condición de borde *bounce-back*, condición de borde reflectante y condición de borde fluidizado.

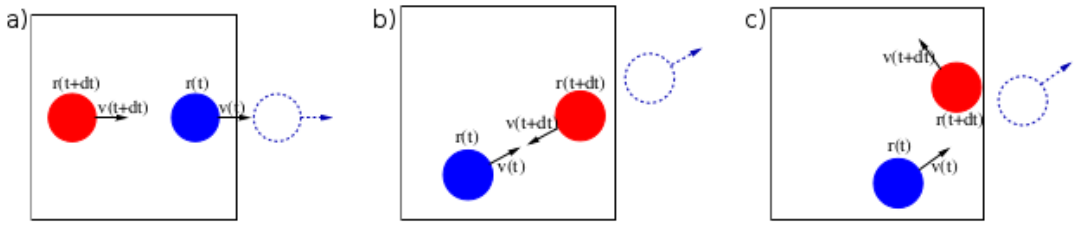


Figura 5.1: Tres de las condiciones de bordes que son implementadas en Adentu: a) condición de borde periódica; b) condición de borde *bounce-back*; y c) condición de borde reflectante. [26]

Condición de borde periódico

Muchas veces es necesario simular grandes sistemas, pero hacer estas simulaciones conllevarían grandes costos, por lo que se simula solo una parte del sistema haciendo que se comporte como si este fuera un sistema infinito. La condición de borde periódico determina que cuando una partícula alcanza el borde de la caja de simulación, este es reintegrado en la pared contraria de esta (Figura 5.1-a).

Condición de borde *bounce-back*

La condición de borde *bounce-back* además de su posición, cambia la velocidad de la partícula, por tanto la condición de borde *bounce-back* no conserva el momentum. Cuando una partícula alcanza el borde de la caja de simulación, las componentes paralelas y perpendiculares de la velocidad de la partícula son invertidas (Figura 5.1-b).

Condición de borde reflectante

La condición de borde reflectante (o reflexiva) permite obtener la reflexión que se puede observar con las bolas de villar, donde estas colisionan en un borde y el ángulo de reflexión es el mismo a la de incidencia. En esta condición de borde, tampoco se conserva el momentum, las componentes de la velocidad son invertidas (Figura 5.1-c).

Condición de borde fluidizado

Aveces se requiere simular sistemas fluidizados, donde un extremo del sistema simule la inyección acelerada de partículas. Cuando una partícula alcanza el borde de la caja de simulación, este es reintegrado en la pared contraria con posición aleatoria y una velocidad previamente definida (Figura 5.2).

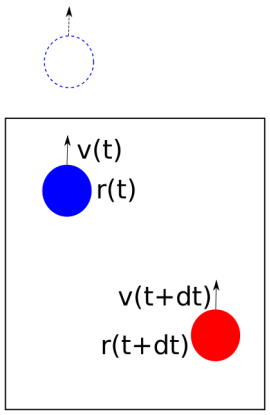


Figura 5.2: Condición de borde fluidizado.

5.2. Estructura de datos

Para poder realizar la simulación de sistemas de granos fluidizados, es necesario crear estructura de datos para poder almacenar y manejar datos sobre el sistema (granos, condiciones de borde, fuerzas, etc.) e inferir información al respecto de la naturaleza del sistema simulado. A continuación se describen las principales estructuras de datos implementadas en Adentu.

Consideraciones en el uso de memoria

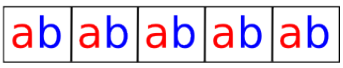
El ancho de banda y la latencia son consideraciones claves para el desarrollo de aplicaciones, pero en CUDA o más específicamente la programación en GPUs, estas consideraciones toman una mayor importancia. En la programación en GPUs nos concierne el ancho de banda en el sentido de cuál es la cantidad de datos que podemos mover de o hacia la memoria global, y el tiempo -latencia- que la operación toma en ejecutarse. En las GPUs, la latencia en la memoria está diseñada a ser 'invisible' dada la ejecución de otros warps. Cuando un warp accede a un espacio de memoria que no está disponible, el hardware lanza una petición -de lectura o escritura- hacia la memoria, esta petición es automáticamente combinada o unida con las peticiones de los otros hilos en el mismo warp, permitiendo a los hilos acceder a secciones de memoria adyacentes y con el área inicial de memoria propiamente alineada.

Las transacciones en la memoria global son de 128 bytes, y alineadas a 128 bytes. Como se mencionó anteriormente, las transacciones son ejecutadas siempre para un warp completo. Cuando un warp alcanza una función que ejecuta una transacción a la memoria global, digamos de 32 bit, el chip en ese momento realizará todas las transacciones necesarias para servir a los 32 hilos pertenecientes al warp. Entonces, si todos los hilos acceden a valores de 32 bit, una sola transacción de 128 bytes es realizada.

Una forma de implementar las estructuras de datos, siguiendo el pensamiento de la programación secuencial, sería utilizar arreglos de estructuras (*Array of Structures* en Inglés), lo cual mantiene las estructuras alineadas en memoria, pero el acceso de una misma propiedad en cada instancia de la estructura no es lineal (Figura 5.3-a), haciendo que el acceso a la propiedad por medio de diferentes hilos tenga que llevarse a cabo utilizando muchas transacciones. Por otro lado, utilizando estructuras de arreglos (*Structure of Arrays* en inglés), el acceso a una misma

propiedad de diferente instancias por medio de diferentes hilos es de forma lineal (Figura 5.3-b).

a)
```
struct foo {
    int a;
    int b;
} bar[5];
```



b)
```
struct foo {
    int a[5];
    int b[5];
} bar;
```

Figura 5.3: Organizar los datos en un (a) Arreglo de Estructuras, hace que el acceso de una propiedad entre múltiples instancias sea no lineal; mientras que organizar mediante una (b) Estructura de Arreglos, el acceso de una misma propiedad ante múltiples instancias sea de forma lineal.

5.2.1. Partículas

En este tipo de simulaciones grano-fluido, se es necesario especificar una simple estructura que sea capaz de representar tanto las partículas de fluido como las partículas de grano. Las partículas deben especificar: su posición (`pos`); su velocidad absoluta (`vel`); su radio (`radius`); su masa (`mass`); las número de veces que se ha visto involucrado en eventos (`nCol`); y en el caso si es un fluido, su velocidad relativa al respecto de la velocidad de centro de masa de la celda donde se ubica. Cabe destacar que en esta estructura -y en otras-, se utilizan dos variables para almacenar los mismos datos, pero el alcance se hace es que las variables que empiezan con `h_` son variables cuya memoria reside en la RAM del anfitrión, mientras que las variables que empiezan con `d_` son variables cuya memoria residen en la memoria global de la tarjeta de video. Con esta estrategia se ahorra tiempo en estar reservando memoria cada vez que se necesita utilizar una variable sobre la GPU o CPU.

Listing 5.1: Estructura `AdentuAtom` (`src/adentu-atom.h`).

```
typedef struct _AdentuAtom {
```

```
    AdentuAtomType type;
    int n;
    float *h_pos;
    float *h_vel;
    float *h_velRel;
    int *h_nCol;
    float *h_mass;
    float *h_radius;

    float *d_pos;
    float *d_vel;
    float *d_velRel;
    int *d_nCol;
    float *d_mass;
    float *d_radius;
} AdentuAtom;
```

5.2.2. Grilla de simulación

Para la regla de colisión en la simulación de fluidos y para la búsqueda de partículas vecinas, es necesaria la utilización de una grilla de celdas, en la cual las partículas se asignan a las celdas correspondientes. La grilla se define en la estructura `AdentuGrid` (Listado 5.2), la cual especifica: el tipo de grilla (`type`); el origen (`origin`); el tamaño (`length`); el número de celdas en cada eje (`nCell`); el total de celdas (`tCell`); el tamaño de las celdas (`h`); una estructura `AdentuCell` donde se almacena información referida a cada celda (`cells`); un arreglo que contiene el índice de la primera partícula encontrada en la celda (`head`); y un arreglo que almacena los índices de todas las partículas de forma de utilizar el método *Linked Cell List*[11] (`linked`).

Listing 5.2: Estructura `AdentuGrid` (`src/adentu-grid.h`).

```
typedef struct _AdentuGrid {
    AdentuGridType type;
    vec3f origin;
    vec3f length;
```

[11] Linked Cell List es una técnica utilizada para buscar todos los átomos o pares de átomos dentro de un área dada, esta área puede ser un set de celdas dentro de una grilla de simulación.

```
    vec3f h;
    vec3i nCell;
    int tCell;
    AdentuCell cells;
    int *h_head;
    int *h_linked;

    int *d_head;
    int *d_linked;
} AdentuGrid;
```

La estructura `AdentuCell` (Listado 5.3) almacena información sobre cada celda en una grilla de simulación, la cual especifica: el número de partículas en la celda (nAtoms); la velocidad de centro de masa de la celda (`vcm`); información si la celda se encuentra en un borde de la caja de simulación (`wall`); y el eje de rotación para MPC (`nhat`).

Listing 5.3: Estructura `AdentuCell` (`adentu-grid.h`).

```
typedef struct _AdentuCell {
    int *h_nAtoms;
    vec3f *h_vcm;
    int *h_wall;
    vec3f *h_nhat;

    int *d_nAtoms;
    vec3f d_*vcm;
    int *d_wall;
    vec3f *d_nhat;
} AdentuCell;
```

5.2.3. Eventos

En Adentu, sucesos como: colisiones entre partículas de grano; colisiones entre partículas de grano y partículas de fluido; condiciones de borde; y MPC, son considerados eventos. El usuario puede optar en activar o desactivar los eventos que quiera que interactúen en el sistema.

Cada evento consta de cuatro funciones básicas con las que interactúa en el flujo de simulación: una función de inicialización del evento (`event_init()`); una función que verifica que el

evento predicho es válido (`event_is_valid()`); una función que atienda a los eventos predichos (`event_attend`), y finalmente una función que haga predicción del siguiente suceso del evento (`event_get_next()`). Estas cuatro funciones básicas se *inscriben* definiendo una estructura del tipo `AdentuEventHandler`, la cual encapsula estas cuatro funciones básicas en forma de punteros a funciones (Listado 5.4).

Listing 5.4: Estructura `AdentuEventHandler` (`src/adentu-event.h`).

```
typedef struct _AdentuEventHandler {
    GSList      *(*event_init)     (AdentuModel *);
    int      (*event_is_valid)  (AdentuModel *, AdentuEvent *);
    void     (*event_attend)    (AdentuModel *, AdentuEvent *);
    AdentuEvent *(*event_get_next) (AdentuModel *);
} AdentuEventHandler;
```

Todo evento que pueda suceder en la simulación primero tiene que ser predicho. El resultado que se obtiene en cada predicción es una instancia de la estructura `AdentuEvent` (Listado 5.5), la cual describe: el tipo de evento (`type`); el tiempo al que debe ser atendido (`time`); la partícula dueña del evento (`owner`); la partícula compañera del evento (`partner`) (como en el caso de colisiones grano-grano o grano-fluido); también debe llevar un conteo de los números de eventos que la partícula dueña del evento a atendido (`nEvents`), de esta forma se puede discriminar los eventos válidos de los inválidos; y finalmente un campo para almacenar información extra del evento (`eventData`).

Listing 5.5: Estructura `AdentuEvent` (`src/adentu-event.h`).

```
typedef struct _AdentuEvent {
    char *type;
    double time;
    int owner;
    int partner;
    int nEvents;
    void *eventData;
} AdentuEvent;
```

5.2.4. Información del sistema

También es necesario contar con una estructura que englobe las propiedades generales del sistema a simular: la aceleración del sistema (`accel`); el tiempo de duración total de la simulación (`totalTime`); el tiempo transcurrido de la simulación (`elapsedTime`); la temperatura de las partículas (`gTemp, fTemp`); la velocidad de centro de masa de las partículas (`vcmGrain, vcmFluid`); las partículas en si (`grain, fluid`); las grillas de simulación (`gGrid, fGrid, mpcGrid`); las condiciones de borde del sistema (`bCond`); los *runnables*, funciones que se ejecutan antes o después de un evento (`pre_event_func, post_event_func`); y finalmente la cola de eventos (`eList`).

Listing 5.6: Estructura `AdentuModel` (`src/adentu-model.h`).

```
typedef struct _AdentuModel {
    vec3f accel;
    double totalTime;
    double elapsedTime;
    double gTemp;
    double fTemp;

    vec3f vcmGrain;
    vec3f vcmFluid;
    AdentuAtom *grain;
    AdentuAtom *fluid;

    AdentuGrid *gGrid;
    AdentuGrid *fGrid;
    AdentuGrid *mpcGrid;
    AdentuBoundaryCond bCond;

    GSList *pre_event_func;
    GSList *post_event_func;

    GSList *eList;
} AdentuModel;
```

5.3. Flujo de simulación

El flujo de una simulación con Adentu consta de cinco etapas: 1) definir las condiciones iniciales del sistema; 2) definir los manejadores de eventos que estarán activos en la simulación; 3) definir los *runnables* pre y post eventos; 4) inicialización de la cola de eventos; 5) inicio del ciclo de simulación.

5.3.1. Definición de condiciones iniciales

En esta primera etapa de simulación se debe hacer instancia de la estructura `AdentuModel`(Listado 5.6) e inicializar las condiciones iniciales.

Listing 5.7: Ejemplo de inicialización de la estructura `AdentuModel`.

```
AdentuModel m;
vecSet (m.accel, 0.0, 0.0, 0.0);
m.totalTime = 50;
m.gTemp = 1.0;
m.fTemp = 0.0;
vecSet (m.vcmGrain, 5., 0., 0.);
vecSet (m.vcmFluid, 2., 0., 0.);
vecSet (m.bCond, ADENTU_BOUNDARY_BBC,
ADENTU_BOUNDARY_BBC, ADENTU_BOUNDARY_BBC);
m.grain = NULL;
m.fluid = NULL;
m.gGrid = NULL;
m.fGrid = NULL;
m.mpcGrid = NULL;
m.pre_event_func = NULL;
m.post_event_func = NULL;
```

Para la creación de las partículas de grano y las (pseudo-)partículas de fluido, se dispone de la función `adentu_atom_create_from_config()` y de la estructura `AdentuAtomConfig` (Listado 5.8) para poder crear un set de partículas. Un ejemplo es el que vemos en el Listado 5.9, donde se crea un set de 200 partículas de granos, con masa constante `1.0` en unidades arbitrarias y radio constante de `0.50` en unidades arbitrarias.

Listing 5.8: Estructura `AdentuAtomConfig` (`src/adentu-atom.h`).

```
typedef struct _AdentuAtomConfig {
    int nAtoms;
    AdentuAtomType type;
    AdentuPropRange mass;
    AdentuPropRange radii;
} AdentuAtomConfig;
```

Listing 5.9: Ejemplo utilización de función `adentu_atom_create_from_config()`

```
AdentuAtomConfig ac;
ac.nAtoms = 200;
ac.type = ADENTU_ATOM_GRAIN;
ac.mass.from = ac.mass.to = 1.0;
ac.mass.rangeType = ADENTU_PROP_CONSTANT;
ac.radii.from = ac.radii.to = 0.50;
ac.radii.rangeType = ADENTU_PROP_CONSTANT;

AdentuAtom a;
adentu_atom_create_from_config (&a, &ac);
m.grain = &a;
```

Al igual que en la creación de las partículas, para la creación de las grillas de simulación para las partículas de grano, fluido, y MPC se dispone de la función `adentu_grid_set_from_config()` y la estructura `AdentuGridConfig` (Listado 5.10) que ayudan en la creación de las grillas de simulación (Listado 5.11).

Listing 5.10: Estructura `AdendtuGridConfig`.

```
typedef struct _AdentuGridConfig {
    vec3f origin;
    vec3f length;
    vec3i cells;
    AdentuGridType type;
} AdentuGridConfig;
```

Listing 5.11: Ejemplo de utilización de función `adentu_grid_set_from_config()`

```
AdentuGridConfig gc;
```

```
vecSet (gc.origin, 0.0, 0.0, 0.0);
vecSet (gc.length, 3.1, 3.1, 3.1);
vecSet (gc.cells, 3, 3, 3);
gc.type = ADENTU_GRID_DEFAULT;

AdentuGrid g;
adentu_grid_set_from_config (&g, &gc);
m.gGrid = &g;
```

5.3.2. Definir manejadores de eventos

Como se comentó anteriormente, la estructura AdentuEventHandler es utilizada para encapsular las funciones básicas de cada evento existente. Para definir los manejadores de eventos que serán utilizados en el ciclo de simulación, se debe crear un arreglo de estructuras AdentuEventHandler (Listado 5.12). Hay que notar que el último elemento en el arreglo es `NULL`, lo cual indica al motor de eventos que no hay más manejadores. Los manejadores de eventos están definidos en los archivos de cabecera de cada evento, los cuales se encuentran en el directorio `src/event/` dentro del árbol de archivos de Adentu, si se utiliza un manejador de evento su cabecera debe ser explícitamente incluida.

Listing 5.12: Definición del arreglo de manejadores de eventos.

```
#include "event/adentu-event-bc.h"
#include "event/adentu-event-mpc.h"
#include "event/adentu-event-ggc.h"
#include "event/adentu-event-gfc.h"

AdentuEventHandler *handler[] = {
    &AdentuMPCEventHandler,
    &AdentuBCGrainEventHandler,
    &AdentuBCFluidEventHandler,
    &AdentuGGCEventHandler,
    &AdentuGFCEventHandler,
    NULL
};
```

5.3.3. Definir *runnables* pre y post eventos

Muchas veces el usuario necesita acceder al estado del sistema en el momento anterior a la atención de un evento, y en el momento posterior de la atención de un evento. En Adentu, se implementa un sistema de dos listas ligadas en las cuales el usuario puede agregar una función que se ejecute antes (`adentu_runnable_add_pre_func()`) o después (`adentu_runnable_add_post_func()`) de cada evento respectivamente, estás funciones son llamadas *Runnables*.

Listing 5.13: Ejemplo de definición de un *runnable*.

```
void post_foo (const AdentuModel *m, const AdentuEvent *e)
{
    printf ("Evento de tipo %s, atendido en tiempo %f.\n",
            AdentuEventTypeStr[e->type], e->time);
}

int main (int argc, char **argv)
{
    AdentuModel m;
    /* ... */
    adentu_runnable_add_post_func (&m, post_foo);
    /* ... */
}
```

5.3.4. Inicialización de la cola de eventos

Como se mencionó anteriormente, la cola de eventos se define en la estructura *AdentuModel* (Listado 5.6). Esta es una lista ligada la cual es populada con los eventos que son predichos. La primera predicción de los eventos es ejecutada antes del inicio del ciclo de simulación y está encargada de las funciones `event_init()` de cada evento. La función que está a cargo de inicializar todos los eventos inscritos, es decir, la función encargada de ejecutar todas las funciones `event_init()` de los manejadores de eventos activos en la simulación, es la función `adentu_event_init()`.

Listing 5.14: Inicialización de cola de eventos.

```
int main (int argc, char **argv)
{
    AdentuModel m;
    /* ... */
    AdentuEventHandler *handler[] = {
        /* ... */
    };
    /* ... */
    m.eList = adentu_event_init (handler, &m);
}
```

5.3.5. Inicio del ciclo de simulación

Una vez inicializada la cola de eventos con los primeros eventos predichos, se inicia el ciclo de simulación. El ciclo de simulación consiste en atender todos los eventos que han sido predichos y están en la cola de eventos; luego de atender cada evento se vuelve a hacer una nueva predicción de eventos, lo que modifica la cola de eventos, esto se repite hasta que se atienda un evento de tipo `ADENTU_EVENT_END` el cual finaliza el ciclo de simulación.

El ciclo de simulación se puede iniciar de dos formas: utilizando la función `adentu_event_loop()` o la función `adentu_graphic_start()`. Utilizando la función `adentu_event_loop()` (Listado 5.15) inicializará el ciclo de simulación sin entorno gráfico (sólo se mostrará información por la salida estándar). Utilizar la función `adentu_graphic_start()` permite inicializar el ciclo con la capacidad de mostrar en modo gráfico -y en tiempo real- el estado del sistema.

Listing 5.15: Inicialización del ciclo de simulación con `adentu_event_loop()`.

```
int main (int argc, char **argv)
{
    AdentuModel m;
    /* ... */
    m.eList = adentu_event_init (handler, &m);
    m.eList = adentu_event_loop (handler, &m);
}
```

Listing 5.16: Inicio ciclo de simulación gráfica con `adentu_graphic_start()`.

```
int main (int argc, char **argv)
{
    AdentuModel m;
    /* ... */
    m.eList = adentu_event_init (handler, &m);
    adentu_graphic_init (argc, argv, &m, &handler);
    adentu_graphic_start ();
}
```

5.3.6. Implementanción de una simulación de prueba

Con el flujo de simulación ya descrito, a continuación se muestra un código que implementa una simulación simple que hace uso de Adentu.

Listing 5.17: Implementación de prueba de Adentu (`src/adentu.c`).

```
/*
    Adentu: An hybrid molecular dynamic software.
    https://github.com/crosvera/adentu

    Copyright (C) 2013 Carlos Rios Vera <crosvera@gmail.com>
    Universidad del Bio-Bio.

    This program is free software: you can redistribute it and/or
    modify it under the terms of the GNU General Public License
    version 3 as published by the Free Software Foundation.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

#include "adentu.h"

/* events */
#include "event/adentu-event-mpc.h"
#include "event/adentu-event-bc.h"
#include "event/adentu-event-gfc.h"
#include "event/adentu-event-ggc.h"


/* usr modules */
#include "usr/atoms-pos-cuda.h"
#include "usr/print-event-info.h"


AdentuEventHandler *handler[] = {
    &AdentuBCGrainEventHandler,
    &AdentuBCFluidEventHandler,
    &AdentuMPCEventHandler,
    //&AdentuGGCEventHandler,
    NULL,
};

int main (int argc, char *argv[])
{
    g_message ("Reseting CUDA Device.");
    adentu_cuda_reset_device ();

    g_message ("Initializing adentu.");
    //set seeds
    //ADENTU_SET_SRAND (time (NULL));
    ADENTU_SET_SRAND (1234567);
```

5.3. FLUJO DE SIMULACIÓN

```
/* crear modelo */
AdentuModel m;
vecSet (m.accel, 0.0, -1.0, 0.0);
m.totalTime = 0.1;
m.gTemp = 1.0;
m.fTemp = 0.0;
vecSet (m.vcmGrain, 0., 0., 0.);
vecSet (m.vcmFluid, 0., 0., 0.);
vecSet (m.bCond, ADENTU_BOUNDARY_BBC,
        ADENTU_BOUNDARY_BBC, ADENTU_BOUNDARY_BBC);
m.grain = NULL;
m.fluid = NULL;
m.gGrid = NULL;
m.fGrid = NULL;
m.mpcGrid = NULL;
m.pre_event_func = NULL;
m.post_event_func = NULL;


/* if using ADENTU_BOUNDARY_FBC as BC,
 * set the grain and fluid velocity */
vec3f gvel, fvel;
vecSet (gvel, 0.0, 0.0, 0.0);
vecSet (fvel, 0.0, 0.0, 0.0);
adentu_event_bc_set_fbc_vel (gvel, fvel);


/* if using MPC events, set the interval time
 * which it will generate events. Also set the
 * rotation angle.*/
adentu_event_mpc_set_dt (1.0);
adentu_event_mpc_set_alpha (3.141592653589793238462);


/* creating grain grid */
AdentuGridConfig gc;
vecSet (gc.origin, 0.0, 0.0, 0.0);
```

```
vecSet (gc.length, 100.1, 100.1, 100.1);
vecSet (gc.cells, 10, 10, 10);
gc.type = ADENTU_GRID_DEFAULT;

AdentuGrid g;
adentu_grid_create_from_config (&g, &gc);

/*set grid into the model*/
m.gGrid = &g;

/* creating fliud and MPC grid */
AdentuGrid fg;
adentu_grid_create_from_config (&fg, &gc);
m.fGrid = &fg;

gc.type = ADENTU_GRID_MPC;
AdentuGrid mpcg;
adentu_grid_create_from_config (&mpcg, &gc);

m.mpcGrid = &mpcg;



/* create grains */
AdentuAtomConfig ac;
ac.nAtoms = 0;
ac.type = ADENTU_ATOM_GRAIN;
ac.mass.from = ac.mass.to = 1.5;
ac.mass.rangeType = ADENTU_PROP_CONSTANT;
ac.radii.from = ac.radii.to = 4.00000;
ac.radii.rangeType = ADENTU_PROP_CONSTANT;

AdentuAtom a;
adentu_atom_create_from_config (&a, &ac);
adentu_atom_set_init_vel (&a, &m);
//adentu_atom_set_init_pos (&a, &g);
```

```
/*set grains into the model*/
m.grain = &a;

/* set atoms into grid */
//adentu_grid_set_atoms (&g, &a, &m.bCond);



/*************************************************/
/* creating fluid*/
ac.nAtoms =  256;
ac.type = ADENTU_ATOM_FLUID;
ac.mass.from = ac.mass.to = 0.5;
ac.mass.rangeType = ADENTU_PROP_CONSTANT;
ac.radii.from = ac.radii.to = 0.00000000000001;
ac.radii.rangeType = ADENTU_PROP_CONSTANT;

AdentuAtom f;
adentu_atom_create_from_config (&f, &ac);
adentu_atom_set_init_vel (&f, &m);
//adentu_atom_set_init_pos (&f, &fg);
m.fluid = &f;
//adentu_grid_set_atoms (&fg, &f, &m.bCond);
//adentu_grid_set_atoms (&mpcg, &f, &m.bCond);



adentu_usr_cuda_set_atoms_pos (&m);




/* General debug Info */
vec3f half, center;
vecScale (half, m.gGrid->length , 0.5);
center.x = m.gGrid->origin.x + half.x;
center.y = m.gGrid->origin.y + half.y;
center.z = m.gGrid->origin.z + half.z;

printf ("gGrid Origin: ");
```

```
    print_vec3f (&m.gGrid->origin);
    printf ("gGrid Length: ");
    print_vec3f (&m.gGrid->length);
    printf ("gGrid Half:   ");
    print_vec3f (&half);
    printf ("gGrid Center: ");
    print_vec3f (&center);


    printf ("Acceleration: ");
    print_vec3f (&m.accel);
    printf ("Boundary Conditions: ");
    printf ("(%s, %s, %s)\n", AdentuBoundaryTypeStr[m.bCond.x],
                              AdentuBoundaryTypeStr[m.bCond.y],
                              AdentuBoundaryTypeStr[m.bCond.z]);

    //adentu_runnable_add_pre_func (&m, print_pre_event);
    adentu_runnable_add_post_func (&m, print_event);
    //adentu_runnable_add_post_func (&m, print_post_event);

    /* setup event engine */
    m.eList = adentu_event_init (handler, &m);
    puts ("");


    /* run in graphic mode or text mode */
    if (argc == 2 && !strncmp (argv[1], "-g", 2))
        {
            adentu_graphic_init (argc, argv, &m, &handler);
            adentu_graphic_set_time_sleep (0);
            adentu_graphic_start ();
        }
    else
        m.eList = adentu_event_loop (handler, &m);

    return 0;
}
```

Capítulo 6

Conclusiones y futuras proyecciones

El presente trabajo ha cumplido con documentar y especificar el desarrollo de Adentu. Dentro de los logros alcanzados, se puede destacar el hecho de que ahora el Grupo de Teoría Cinética cuenta con una herramienta lo suficientemente flexible para apoyar el estudio de medios granulados. Agregar además que Adentu pasa a ser un desarrollo nacional con fines científicos, utilizando lo último en procesamiento de datos en paralelo con GPUs, permitiendo al Grupo de Teoría Cinética estar actualizados en el uso de nuevos métodos y tecnologías. Importante es también mencionar que este desarrollo al ser liberado como Software Libre tiene dos grandes ventajas: la primera que es que cualquier persona puede hacer uso de Adentu permitiendo beneficiar no solo al Grupo de Teoría Cinética; la segunda ventaja es que relacionado con que Adentu lo puede utilizar cualquier grupo o persona, estos pueden reportar posibles errores o agregar nuevas funcionalidades al proyecto, permitiendo así que exista un constante mejoramiento del proyecto.

6.1. Rendimiento

Con la utilización de CUDA es posible poder hacer cálculos predictivos de forma paralela. Utilizando la herramienta `nvprof` que viene incluido en el SDK de CUDA, es posible realizar perfiles de rendimiento de nuestras aplicaciones hechas en CUDA, permitiendo identificar las rutinas ejecutadas y el tiempo que toman para realizarse. La Figura 6.1 muestra el perfil de la rutina que hace la integración de posiciones y velocidades ante diferentes cantidades de partículas y la Figura 6.2 muestra el perfil de la rutina que hace la predicción de eventos de condición de borde

ante un número variable de partículas. Podemos observar que la escala de tiempo está en el rango de unos pocos microsegundos, alcanzando su tope en milisegundos cuando la cantidad de partículas llega a ser del rango de cientos de miles. Con estos dos gráficos podemos concluir que el trabajo con muchas partículas de forma paralela es *barato*, lo cual pone como único límite la cantidad de memoria ram con la que cuenten las tarjetas gráficas en las cuales se ejecute Adentu.

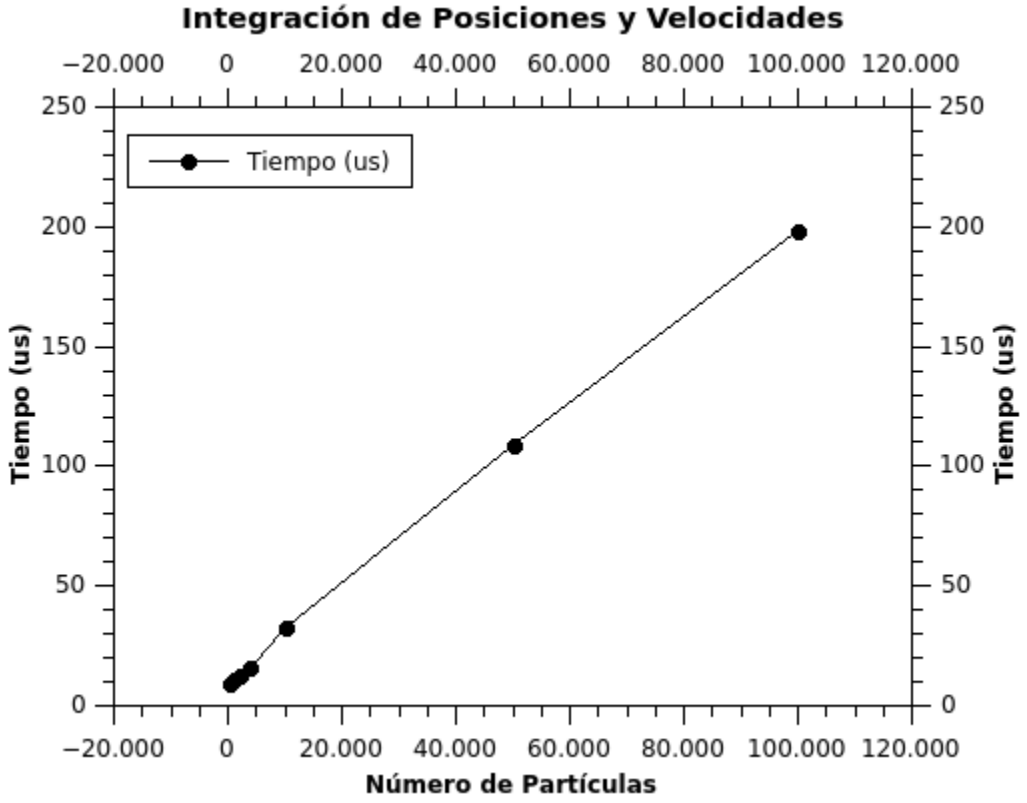


Figura 6.1: Tiempo de ejecución de la rutina que integra las posiciones y velocidades de las partículas.

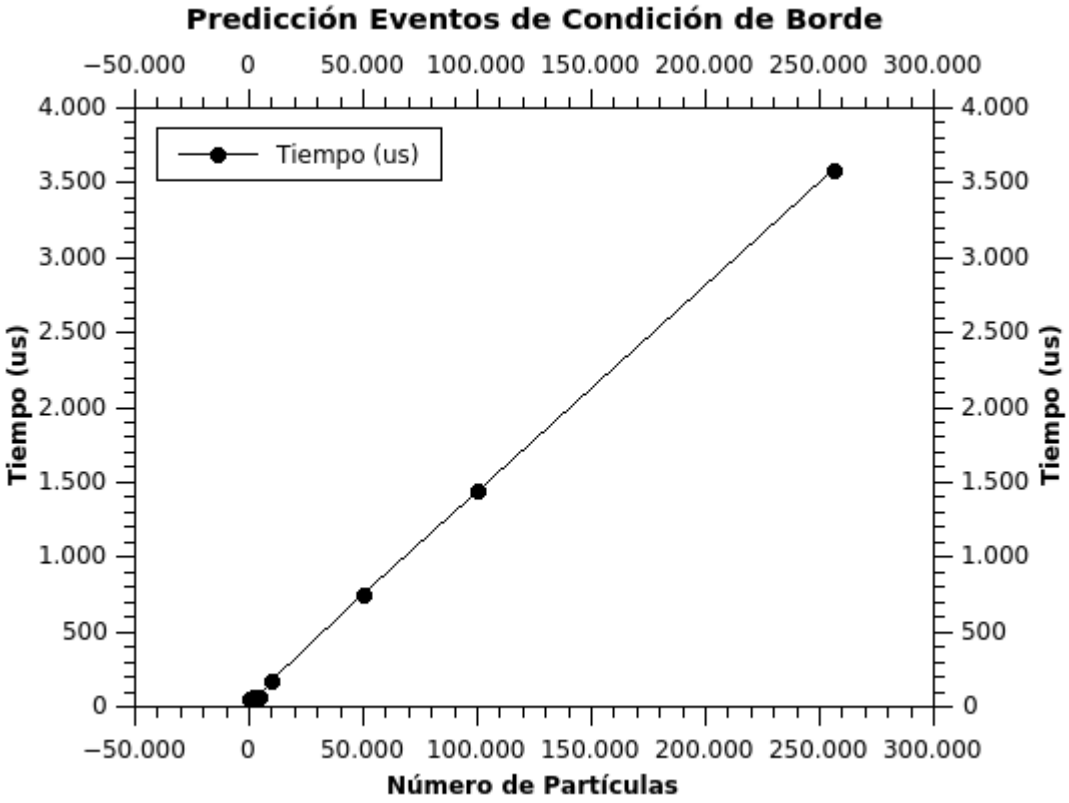


Figura 6.2: Tiempo de ejecución de la rutina de predicción de evetos de condiciones de borde ante varias cantidades de partículas.

6.2. Proyecciones y trabajos futuros

Si bien Adentu en su versión 0.2 cumple con todas las funcionalidades que le son requeridas, quedan muchas mejoras y características complementarias que se pueden agregar. Entre las características que se quieren agregar en futuras versiones, se encuentran:

Capacidad de agregar objetos a la simulación Es necesario integrar un módulo que permita integrar a la simulación geometrías o mallas, que describan un objeto con el que las partículas puedan interactuar, de esta forma el investigador puede generar una malla utilizando herramientas de tipo CAD e importarlas con Adentu para observar el comportamiento de las partículas ante este objeto.

Visualización gráfica Si bien actualmente Adentu consta con un módulo de visualización gráfica, este no es eficiente. Además una buena idea es integrar SiMon, desarrollo hecho por Julio Borja [2], el cual permite visualización gráfica de las simulaciones de forma remota.

Bindings para otros lenguajes Actualmente en el desarrollo científico se utiliza una variedad de lenguajes de programación. Para poder cubrir una mayor área de aplicación, se pueden crear bindings[1] de Adentu para otros lenguajes, tales como Python o Fortran.

[1] Un binding es código que hace de pegamento entre un programa o biblioteca como Adentu para que pueda ser utilizado en otro lenguaje de programación en el que no ha sido programado originalmente.

Bibliografía

[1] OpenMP Architecture Review Board. The openmp api specification for parallel programming, 2012.

[2] Julio Borja. Desarrollo de una aplicación gráfica en modalidad cliente-servidor para el monitoreo de simulaciones de dinámica molecular conducida por eventos, 2010.

[3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Pat Hanrahan, Mike Houston, and Kayvon Fatahalian. Brookgpu, 2004.

[4] Rajkumar Buyya. *High Performance Cluster Computing: Programming and Applications*, volume 2. Prentice Hall, 1999.

[5] Marcel G. Clerc, Patricio Cordero, J. Dunstan, K. Huff, Nicolás Mujica., Dino Risso, and G. Varas. Liquid–solid-like transition in quasi-one-dimensional driven granular media. *Nature Physics*, 2008.

[6] Shane Cook. *CUDA Programming. A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann, 1 edition, 2012.

[7] Patricio Cordero, Mauricio Marín, and Dino Risso. Efficient simulations of microscopic fluids: Algorithm and experiments. *Chaos, Solitons and Fractals*, 1995.

[8] Jacques Duran. *Sands, powders, and grains: An introduction to the physics of granular materials*. Springer, 1999.

[9] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901 – 1909, dec. 1966.

[10] G. Gompper, T. Ihle, D.M. Kroll, and R.G. Winkler. Multi-particle collision dynamics a particle-based mesoscale simulation approach to the hydrodynamics of complex fluids. *Advanced Computer Simulation Approaches for Soft Matter Sciences III, Advances in Polymer Science*, 221, 2009.

[11] The Open Group. Posix.1-2008, 2008.

[12] Mark Jason Harris. Real-time cloud simulation and rendering, 2003.

[13] Argonne National Laboratory. Mpich high-performance and portable mpi, 2012.

[14] L. D. Landau, E. M. Lifshitz, J. B. Sykes, and W. H. Reid. *Fluid Mechanics*. Pergamon Press Oxford, England, 1959.

[15] Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. In *In Proc. SIGGRAPH*, pages 327–335, 1990.

[16] Calvin Lin and Larry Snyder. *Principles of Parallel Programming*. Addison-Wesley, 2008.

[17] Anatoly Malevanets and Raymond Kaparal. Mesoscopic model for solvent dynamics. *J. Chem. Phys.*, 110, 1999.

[18] Mauricio Marín, Dino Risso, and Patricio Cordero. Efficient algorithms for many-body hard particle molecular dynamics. *Journal of Computational Physics*, 109:306–317, 1993.

[19] Truls Mosegaard and Jon Nielsen. Parallel programming with cuda. Master's thesis, Roskilde Universitet, 2012.

[20] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi. Whitepaper, 2011.

[21] NVIDIA. Cuda c programming guide, 2012.

[22] Khronos Opencl and Aaftab Munshi. The opencl specification version: 1.2 document revision: 15, 2011.

[23] Dennis C. Rapaport. *The Art of Molecular Dynamics Simulation*. Cambridge University Press, 2004.

[24] Nicolas Rivas, Suomi Ponce, Basile Gallet, Dino Risso, Rodrigo Soto, Patricio Cordero, and Nicolas Mujica. Sudden chain energy transfer events in vibrated granular media. *Physical Review Letters*, 2011.

[25] Danilo Sandoval. Desarrollo de software simulador híbrido vía dinámica molecular para sistemas con interacción fluido-grano, 2012.

[26] Annika Schiller and Godehard Sutmann. Mesoscopic simulations of hydrodynamic interactions using a particle-based model on cell broadband engine. In *Mitteilungen - Gesellschaft für Informatik e.V., Parallel-Algorithmen und -Rechnerstrukturen*, volume 26, pages 17 – 26, December 2009.

[27] Indiana University, Indiana University Research, and Technology orporation. Open mpi: Open source high performance computing, 2013.