Proyecto Fin de Carrera

Extensión OmpSS para CUDA y OpenCL

(OmpSS extension for CUDA and OpenCL)

Para acceder al Titulo de

INGENIERO EN INFORMÁTICA

Autor: Florentino Sainz Manteca

Septiembre – 2012

INGENIERÍA EN INFORMÁTICA

CALIFICACIÓN DEL PROYECTO FIN DE CARRERA

Realizado por: Florentino Sainz Manteca

Director del PFC: José Luis Bosque Orero

Título: "Extensión OmpSS para CUDA y OpenCL"

Title: "OmpSS extension for CUDA and OpenCL**"**

Presentado a examen el día:

para acceder al Título de

INGENIERO EN INFORMÁTICA

Composición del Tribunal:

Presidente (Apellidos, Nombre): Michael González Harbour

Secretario (Apellidos, Nombre): María del Carmen Martínez Fernández

Vocal (Apellidos, Nombre): Rafael Menéndez de Llano Rozas

Vocal (Apellidos, Nombre): Pablo Sánchez Barreiro

Vocal (Apellidos, Nombre): Roberto Sanz Gil

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: Vocal

Fdo.: Vocal

Fdo.: Vocal

Fdo.: El Director del PFC

ÍNDICE

TABLA DE CONTENIDOS

FIGURAS Y GRÁFICOS

AGRADECIMIENTOS

He de expresar mi agradecimiento a las personas que de forma directa han hecho posible este proyecto y las cuales me han ayudado en el aprendizaje tanto de OmpSS, como de CUDA y OpenCL.

Además de agradecer las molestias que se han tomado para seguir un calendario de reuniones semanales, que dadas sus obligaciones, no siempre ha sido fácil cumplir.

-**Jose Luis Bosque,** mi director de proyecto, por brindarme la oportunidad de realizar este proyecto y encontrar el tiempo suficiente para guiarme durante la realización del mismo.

-**Vicenç Beltran**, por ser mi principal vía de comunicación con el BSC y los demás integrantes del proyecto, además de sus aportaciones al mismo y la organización de las reuniones de seguimiento.

-Xavier Martorell, Judit Planas, Roger Ferrer, Rosa María Badia y Speziale Ettore por su asistencia a las reuniones y la resolución de dudas respecto a las partes del proyecto en las que coincidimos.

Además agradezco a mi familia su apoyo durante mis estudios y a los profesores que durante toda mi vida, y especialmente en este periodo universitario, me han instruido y entregado parte de los conocimientos de los que dispongo actualmente y gracias a los cuales he sido capaz de realizar este proyecto.

RESUMEN

El presente proyecto tiene por objeto mejorar la compatibilidad del modelo de programación OmpSS del BSC-CNS con la ejecución en aceleradores.

OmpSS está compuesto por dos componentes diferenciados, un runtime (Nanox) y un compilador (Mercurium), los cuales han de ser modificadas de acuerdo a las necesidades del proyecto.

El proyecto está estructurado en dos partes correspondientes a dos tecnologías diferentes para ejecutar código en aceleradores, CUDA y OpenCL.

La primera parte del proyecto, consiste en una extensión de una tecnología ya soportada por OmpSS, **CUDA**, para permitir una programación más sencilla de la misma, en la que se persigue abstraer al usuario de las llamadas de bajo nivel. Para ello se ha modificado la implementación ya existente del **compilador**.

La segunda parte del proyecto, consiste en añadir soporte para ejecutar código **OpenCL**, con el objetivo de poder programar en este lenguaje una forma muy similar a CUDA, y por consiguiente a cualquier otro dispositivo. Para ello hemos de crear la implementación tanto en el **compilador**, como en el **runtime**, para que sea capaz de ejecutar tareas OpenCL.

Ambas partes serán verificadas con diferentes pruebas, creadas en la realización del proyecto o adaptadas de pruebas ya existentes.

SUMMARY

This Project's goal is improving BSC-CNS's OmpSS Programming Model compatibility with accelerators.

OmpSS is composed with two different components, Nanox Runtime and Mercurium Compiler, which have to be modified according to our requirements.

This Project is structured in two different parts, corresponding to two different technologies, CUDA and OpenCL.

First part, consisting in extending current OmpSS implementation of CUDA, in order to make it easier to use, we have to remove the need to use low-level calls to invoke CUDA *tasks,* to reach this goal we have to modify Mercurium Compiler.

Second part, consisting in adding OpenCL support to OmpSS, *tasks* are going to be defined pretty similar to CUDA definitions, therefore similar to other devices, to reach this goal we have to add OpenCL device implementation in Mercurium Compiler and Nanox Runtime.

Both parts will be verified with different tests, created during the development of this Project or adapted from already existing ones.

PALABRAS CLAVE

En este documento se utilizan varias palabras clave que han de ser definidas, referentes tanto a OpenCL, CUDA como a OmpSS. Por lo tanto se irán explicando en las correspondientes secciones, junto a estas tecnologías.

Algunos **conceptos clave** generales:

-**GPU** (Graphics Processor Unit): Circuito electrónico diseñado para manipular y alterar imágenes para ser mostradas en una pantalla, conocido coloquialmente como **Tarjeta gráfica**.

-**GPGPU** (General-Purpose Computing on Graphics Processing Units): Utilización de **GPUs** para realizar procesamientos de propósito general que tradicionalmente corresponderían a una **CPU**.

Además en las siguientes referencias se pueden encontrar los conceptos clave de cada tecnología utilizada en el proyecto debidamente separados, no es necesaria su lectura hasta el momento en que aparecen.

- Palabras clave CUDA

- Palabras clave OpenCL

- Palabras clave OmpSS

CAPÍTULO 1. INTRODUCCIÓN Y OBJETIVOS

1.1 MOTIVACIÓN

Tradicionalmente se han utilizado CPUs en las computadoras destinadas al cálculo, sin embargo, en entornos de alto rendimiento en los que se necesitaba la mayor capacidad de procesamiento posible, se empezó a investigar la posibilidad de usar Tarjetas Gráficas (GPUs), que como su nombre indica estaban destinadas a la generación de gráficos 3D, pero que disponen de una gran capacidad de procesamiento, que puede ser del orden de 40 veces mayor que una CPU. Este concepto se conoce como **GPGPU** [1]**.**

Programar estos dispositivos tenía una complejidad elevada, especialmente porque había que hacerlo en lenguaje ensamblador. Actualmente, y con el apoyo de los dos principales fabricantes de GPUs (NVIDIA y AMD) se han desarrollado extensiones a los lenguajes C/C++ que facilitan la programación de estos dispositivos. Estas extensiones se conocen como **CUDA** [2]**,** propiedad de NVIDIA y que solo puede ejecutarse en GPUs de esta marca, y su alternativa abierta, **OpenCL**[5] que puede ejecutarse tanto en GPUs (AMD, NVIDIA o Intel) como en CPUs.

Aunque con **CUDA** y **OpenCL** se posibilita la programación **GPGPU** utilizando lenguajes de alto nivel, aunque se ha de tener en cuenta que las GPU inicialmente no estaban diseñadas para este uso, por lo cual tienen sus limitaciones a la hora de ser programadas. Especialmente en el acceso a memoria, que está más restringido que en una CPU.

Como consecuencia de ello, en el **BSC-CNS** (Barcelona Supercomputing Center – Centro Nacional de Supercomputación) se inició un proyecto conocido como OmpSS. En el cual se pretende integrar la programación GPGPU en su propio entorno de programación y simplificar este tipo de programación.

OmpSS es un paradigma de programación basado en la anotación de bloques de código como tareas (referidas durante este documento como *tasks*). La anotación se realiza mediante directivas (P.E. *#pragma omp task device(..)*). Estas t*asks* se ejecutan de forma asíncrona, sin embargo, tienen un orden predeterminado ya que existen dependencias entre ellas. Estas dependencias serán indicadas por el programador en la directiva, pudiendo ser de entrada (*input)* o de salida (*output)*.

Este entorno de programación se descompone en dos partes diferenciadas:

-El runtime (**Nanox**), el cual proporciona soporte para la ejecución y planificación de tareas y sincronización de las mismas a través de dependencias de datos en entornos paralelos. Además de dar soporte a la ejecución de las mismas en diversos tipos de dispositivos, como GPUs y en un futuro, sistemas distribuidos.

-El compilador (**Mercurium**), el cual se utiliza para transformar directivas de compilación sencillas que el programador incluye en su código en llamadas al runtime **Nanox**, más complejas, para ejecutar estas tareas.

El principal motivo que ha despertado el interés por la programación GPGPU es la diferencia de la arquitectura de las propias GPUs respecto a las CPUs tradicionales, que si bien no es tan flexible como estas últimas, provoca que para determinados cálculos o algoritmos sea mucho más eficiente.

Esta arquitectura se puede ver en la siguiente figura de una forma simplificada.

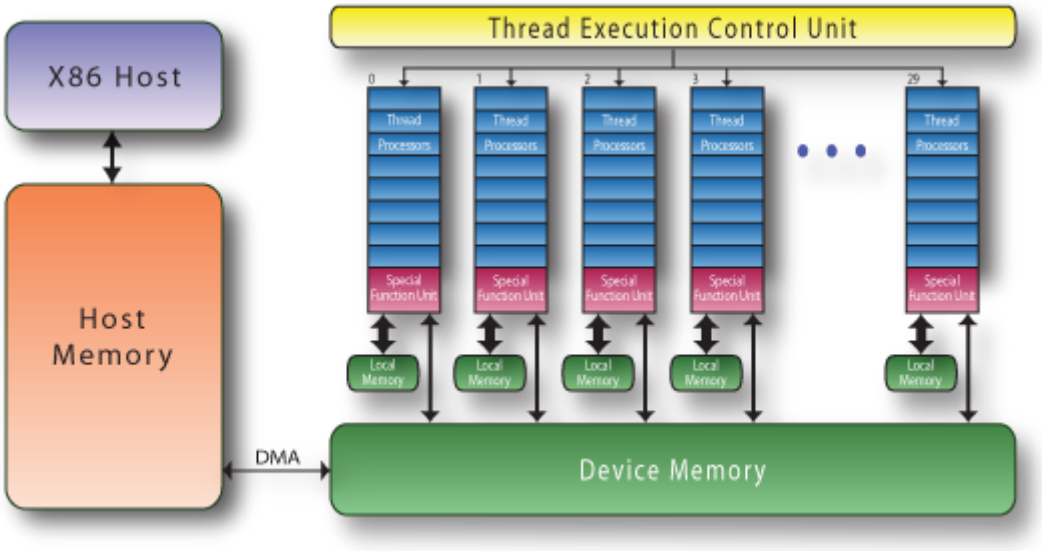


Figura 1: Arquiectura GPU.

Como vemos en la figura, la **GPU** *(device)* es una unidad de ejecución totalmente separada del sistema tradicional de **CPU** y **Memoria** *(host)* ya que tiene su propia memoria y sus propias unidades de ejecución, viendo esto podemos intuir que la programación de estos dispositivos no es similar a los sistemas tradicionales.

Identificándolo por colores, las figuras azules son las unidades de procesamiento de la GPU, las figuras verdes son la memoria de las mismas y la zona naranja es la memoria de la CPU.

Vemos que la memoria del *device* es independiente de la memoria del *host* y que únicamente se pueden comunicar entre ambas memorias, es decir, la CPU no puede acceder a la memoria de la GPU directamente, ni las unidades de ejecución de la GPU pueden acceder a la memoria del host.

Por lo tanto para ejecutar código en la GPU hemos de realizar los siguientes pasos:

1- Copiar los datos de la memoria de la CPU a la memoria del dispositivo.
2- Lanzar el código en las unidades de procesamiento del dispositivo
3- Copiar los datos de la memoria del dispositivo a la memoria de la CPU.

Como se puede ver en la figura 1, la GPU dispone de una gran cantidad de unidades de procesamiento agrupadas en diferentes grupos los cuales comparten una memoria local de acceso rápido (únicamente visible para su grupo). Además de compartir la memoria del dispositivo entre todas las unidades de procesamiento, aunque el acceso a la misma es más lento.

Para aprovechar la potencia de estas unidades de procesamiento, se lanza gran cantidad de threads ligeros, para aprovechar el elevado paralelismo que nos ofrecen este tipo de dispositivos, normalmente estos threads procesan un solo elemento de los datos.

Esto provoca que en ocasiones no sea eficiente utilizar las GPU, debido a que el algoritmo que se desea ejecutar no se adapte bien a utilizar tal cantidad de threads ligeros, o porque no merezca la pena realizar la copia de datos a memoria.

1.2 OBJETIVOS

Los objetivos de este proyecto de fin de carrera son los de modificar el compilador Mercurium para ofrecer soporte a programación **GPGPU.** Este compilador está desarrollado por el **BSC-CNS,** este proyecto se realizará en colaboración con el **BSC** que ofrecerá soporte sobre las partes ya existentes del compilador y del runtime.

Estos objetivos generales quedan concretados en los siguientes pasos:

1- Extender el compilador para ofrecer soporte a **dispositivos OpenCL**, para que al indicar mediante una directiva que una función pertenece a este dispositivo, el compilador sea capaz de generar el código para que el runtime pueda lanzarla en un dispositivo con soporte **OpenCL** (GPU o CPU) cada vez que el programador invoque a la misma.
2- Extender la implementación actual del compilador que da soporte a **dispositivos CUDA**, la cual actualmente permite marcar mediante una directiva las funciones como pertenecientes a este dispositivo, pero ha de ser el programador quien desarrolle parte del código que llama a la función. Nuestro objetivo será generar este código automáticamente en el compilador.
3- Apoyándonos en los dos objetivos anteriormente mencionados y en la implementación ya existente, soportar la posibilidad de que el programador realice varias implementaciones de una misma función para varios dispositivos y sea el runtime (**Nanox**) el encargado elegir la más conveniente dependiendo de la máquina donde se ejecute.
4- Adaptación de pruebas ya existentes y creación de pruebas que verifiquen el cumplimiento de los objetivos.

CAPÍTULO 2. TECNOLOGÍAS Y HERRAMIENTAS UTILIZADAS

En este capítulo describiremos todas las tecnologías y herramientas utilizadas en la realización del proyecto además de los conceptos clave necesarios para comprender la implementación del mismo.

2.1 CUDA

En este capítulo explicaremos todo lo necesario para poder entender la primera parte del proyecto en lo referente a CUDA.

CUDA son las siglas de Compute Unified Device Architecture (Arquitectura de Dispositivos de Cómputo Unificado) que hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por nVidia que **permiten a los programadores usar una variación de algunos lenguajes de programación, como C y C++, para codificar algoritmos en GPU de nVidia** [3][4].

CUDA intenta explotar las ventajas de las GPU frente a las CPU de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un altísimo número de hilos simultáneos. Por ello, si una aplicación está diseñada utilizando numerosos hilos que realizan tareas independientes (que es lo que hacen las GPU al procesar gráficos, su tarea natural), una GPU podrá ofrecer un gran rendimiento en campos que podrían ir desde la biología computacional a la criptografía por ejemplo.

A continuación se definirán varios **conceptos clave relativos** a CUDA**:**

1- **Kernel:** Función escrita en código C/C++ que se llama desde la CPU y se ejecuta en un dispositivo CUDA (GPU), estas funciones tienen ciertas limitaciones derivadas del lenguaje, no se pueden utilizar punteros a funciones, objetos…
2- **Thread:** Instancias de ejecución de un kernel, todos los threads de un mismo kernel ejecutan el mismo código, exceptuando que cada uno tiene una ID propia que se puede utilizar para tomar decisiones de control.
3- **Device:** Nomenclatura que se utiliza para referirse al dispositivo que ejecuta el kernel CUDA (GPU).
4- **Host:** Nomenclatura que se utiliza para referirse al dispositivo que lanza el kernel CUDA (CPU).
5- **Bloque:** Agrupación de threads que pueden compartir datos mediante la memoria compartida y sincronizar su ejecución, puede tener de 1 a 3 dimensiones.
6- **Grid:** Cantidad de bloques que se ejecutan por en cada kernel, es decir, en cada kernel se ejecutan “número de grids*tamaño de bloque” threads, puede tener de 1 a 3 dimensiones que han de coincidir con las del bloque.

*La distribución en grids y bloques recibirá el nombre de **ndrange**, por similitud con OpenCL.

CUDA intenta aprovechar el gran paralelismo, y el alto ancho de banda de la memoria en las GPU en aplicaciones con un gran coste de cómputo frente a realizar numerosos accesos a memoria principal, lo que podría generar un cuello de botella.

El modelo de programación de CUDA está diseñado para que se creen aplicaciones que de forma transparente escalen su paralelismo para poder incrementar el número de núcleos computacionales. Este diseño contiene tres puntos clave, que son la jerarquía de grupos de hilos, las memorias compartidas y las barreras de sincronización.

La estructura que se utiliza en este modelo está definido por un grid, dentro del cual hay bloques de hilos que están formados por 512 hilos distintos como máximo.

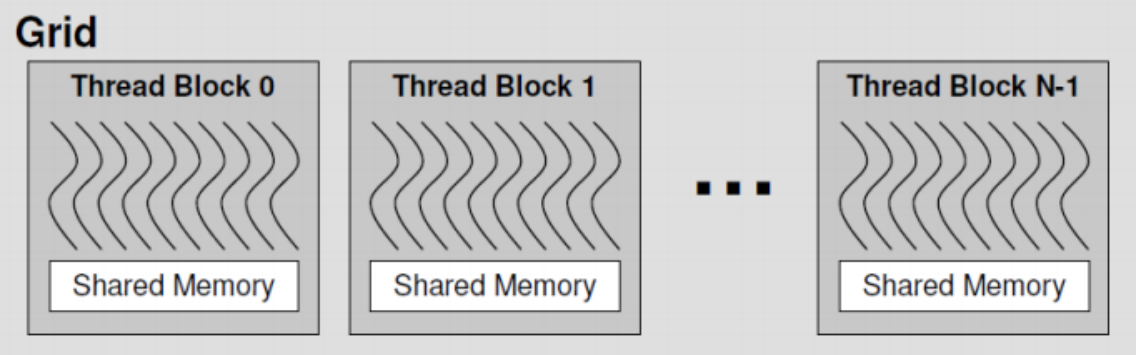


Figura 2: Distribución de threads en un kernel CUDA

Cada hilo dentro de un bloque está identificado con un identificador único, que se accede con la variable threadIdx. Esta variable se utiliza para repartir el trabajo entre distintos hilos y tiene 3 componentes (x, y, z), coincidiendo con las dimensiones de bloques de hilos.

Al igual que los hilos, los bloques se identifican mediante blockIdx (en este caso con dos componentes x e y). Otro parámetro útil es blockDim, para acceder al tamaño de bloque. Por ejemplo, para obtener la **id global del thread** (en la dimensión 1), utilizaríamos la siguiente expresión, **threadIdx.x + (blockIdx.x*blockDim.x).**

Los kernel son funciones C, pero tienen ciertas limitaciones:

- Solo pueden acceder a la memoria de la GPU.
- Deben ser funciones que retornen void.
- No pueden tener un número variable de argumentos.
- No pueden ser recursivos.
- No pueden utilizar variables estáticas.

Ahora que ya conocemos como son los **kernel** en CUDA y que se lanzan en un número de threads determinados por el tamaño de bloque y el tamaño de grid, explicaremos el proceso necesario para lanzar un kernel. Como ejemplo tomaremos el siguiente[1]:

```
__global__ void square_array(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

Código 1: Kernel CUDA

El proceso necesario para lanzar el kernel lo vemos en la siguiente figura:

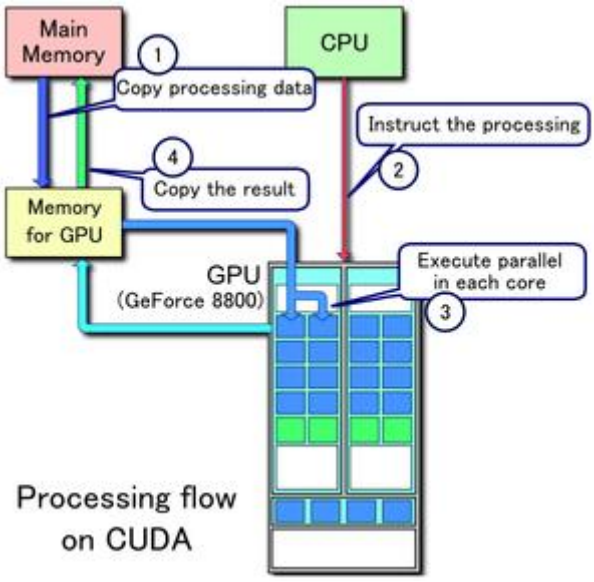


Figura 3: Pasos para lanzar un kernel CUDA

1- **Copiar los datos a procesar:** Copiar los datos que van a ser procesados en el kernel a la memoria de la GPU, es decir, copiar el array a la GPU.
   Esto será posible gracias a las instrucciones CUDAMalloc y CUDAMemCpy en las cuales se reserva un espacio en la memoria de la GPU y se copian los datos en el. De este proceso ha de encargarse el programador de aplicaciones CUDA.
2- **Lanzar el kernel:** La CPU llama al kernel. Con la dirección de los datos en la memoria de la GPU que se obtuvo en el paso 1, y debe indicar el tamaño de bloque y tamaño de grid. La llamada es la siguiente:

   square_array <<< n_blocks, block_size >>> (a_d, N);

3- **Ejecutar el kernel:** Este paso es transparente al usuario, CUDA ejecuta los kernel solicitados en el paso 2. Mientras los threads se están ejecutando, la CPU puede hacer otras acciones, o simplemente esperar a que finalice la ejecución.

[1] http://llpanorama.wordpress.com/2008/05/21/my-first-CUDA-program/

4- **Copiar el resultado:** Una vez que se ha ejecutado el kernel y tenemos el resultado en la memoria de la GPU, hemos de realizar el paso inverso al paso 1, es decir, mediante la instrucción CUDAMemCpy, copiar los datos de la GPU a la CPU para poder seguir trabajando con ellos.

Un ejemplo sencillo de como ejecutaría el kernel anterior de acuerdo a los pasos anteriormente explicados es el siguiente:

```
int main(void)
{
  //Tamaño
  const int N=1024;
  //Punteros memoria principal del host (CPU) y del device (GPU)
  float *a_h, *a_d;
  size_t size = N * sizeof(float);
  //Reservar espacio en el host
  a_h = (float *)malloc(size);
  for (int i=0; i<N; i++) a_h[i] = (float)i;
  //PASO 1
  //Reservar espacio en la GPU
  cudaMalloc((void **) &a_d, size);
  //Copiar array a la memoria de la GPU
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  //PASO 2
  //Lanzar el kernel (calculando tamaño de bloque y de thread,
  //este tamaño debe ser acorde al número de datos a procesar)
  int block_size = 4;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  square_array <<< n_blocks, block_size >>> (a_d, N);
  //PASO 3, ESPERAR
  //PASO 4
  //Copiar resultado a la CPU de nuevo (espera a que finalice el kernel)
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  //Imprimir resultados
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
  //Liberar
  free(a_h); cudaFree(a_d);
}
```

Código 2: Ejemplo de código C que lanza un kernel CUDA

2.2 OPENCL

A continuación explicaremos el lenguaje de programación OpenCL en el cual se realizará la segunda parte del proyecto. Aunque la **API** de OpenCL es muy diferente a la de CUDA, realmente ambos siguen la misma filosofía, y portar un programa de un lenguaje a otro suele consistir en traducir las llamadas correspondientes [6].

OpenCL (**Open Computing Language**, en español lenguaje de computación abierto) consta de una interfaz de programación de aplicaciones y de un lenguaje de programación. Juntos permiten crear aplicaciones con paralelismo a nivel de datos y de tareas que pueden ejecutarse tanto en unidades centrales de procesamiento como unidades de procesamiento gráfico. El lenguaje está basado en C99, eliminando cierta funcionalidad y extendiéndolo con operaciones vectoriales [7][8].

La especificación original fue creada por Apple y desarrollada en conjunto con AMD, IBM, Intel y nVidia. Aunque posteriormente fue estandarizado por el grupo Khronos (www.khronos.org), encargado de mantener el estándar actualmente.

La principal diferencia con CUDA es que OpenCL puede ser ejecutado en cualquier dispositivo que implemente el estándar, que al ser abierto no es únicamente nVidia, sino que hay implementaciones de Intel, AMD y nVidia, que incluyen tanto CPU como GPU. Es decir, que **OpenCL además de poder ejecutarse únicamente en GPUs, puede ser ejecutado en CPUs.** Aunque esto no parezca muy útil, ya que para ejecutar un software en CPU es más sencillo y versátil hacerlo en los lenguajes tradicionales, nos ofrece mayor portabilidad al asegurarnos que aunque no se disponga de una GPU nuestro programa va a poder ejecutarse correctamente.

Otra importante diferencia con CUDA es la existencia de Colas/Queues, que se asocian a un dispositivo OpenCL. Estas colas controlan las acciones que ha de realizar el dispositivo de una forma básica, ya que si se configuran para funcionar en orden, ninguna operación comienza hasta que han terminado las que fueron declaradas anteriormente. Si el programador lo desea, se pueden programar para un funcionamiento fuera de orden, en cuyo caso se ha de garantizar el orden de las acciones (en caso de ser necesario) mediante eventos.

A continuación, aunque como ya hemos dicho es muy similar a CUDA, se definirán varios **conceptos clave** referentes a OpenCL:

1- **Kernel:** Función escrita en código C/C++ que se llama desde la CPU y se ejecuta en un dispositivo OpenCL (GPU o CPU). Estas funciones tienen ciertas limitaciones derivadas del lenguaje, no se pueden utilizar punteros a funciones, objetos…
2- **Thread:** Instancias de ejecución de un kernel. Todos los threads de un mismo kernel ejecutan el mismo código, exceptuando que cada uno tiene una ID propia que se puede utilizar para tomar decisiones de control.
3- **Device:** Nomenclatura que se utiliza para referirse al dispositivo que ejecuta el kernel OpenCL (normalmente GPU).
4- **Host:** Nomenclatura que se utiliza para referirse al dispositivo que lanza el kernel OpenCL (CPU).
5- **Grupo:** Agrupación de threads que pueden compartir datos mediante la memoria compartida y sincronizar su ejecución. Puede tener de 1 a 3 dimensiones. Su tamaño se denomina **local_size.**
6- **Ndrange:** Distribución de los grupos que se ejecutan para cada kernel, es decir, en cada kernel se ejecutan "(global_size/local_size)" grupos. El número de elementos (global_size) puede tener de 1 a 3 dimensiones que han de coincidir con las del local_size del grupo. A diferencia de CUDA, en OpenCL global_size es el número total de threads que se ejecutarán, mientras que en CUDA es el número de grupos.

La estructura del ndrange en el que se lanzan los kernel es la siguiente:

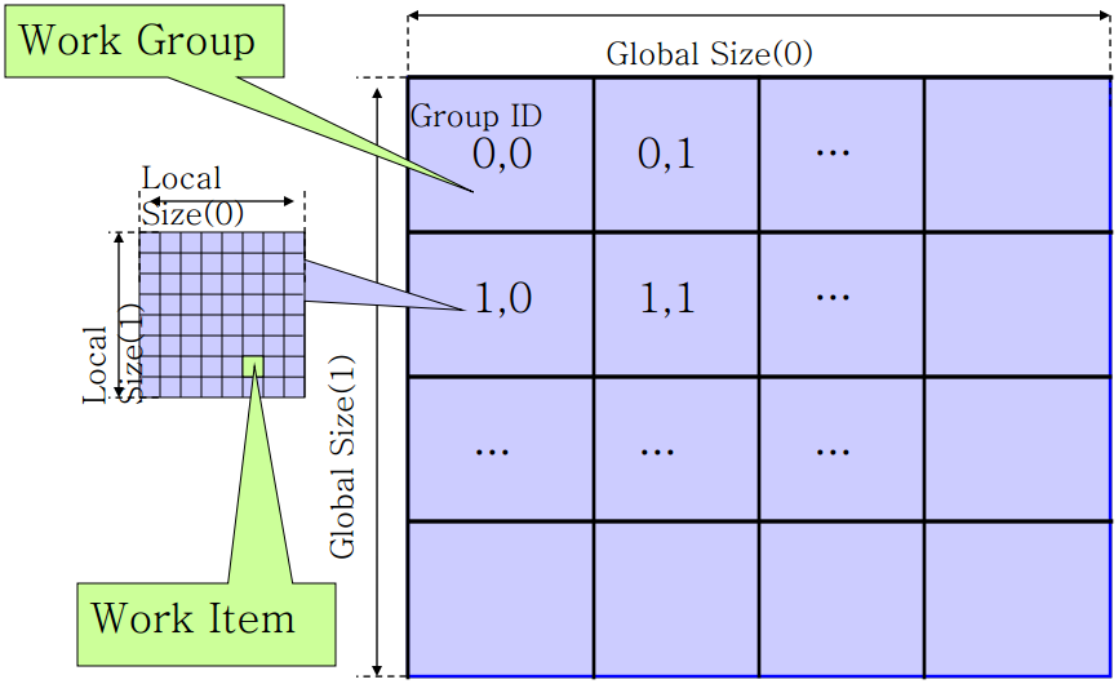


Figura 4: Distribución threads (ndrange) en OpenCL

Al igual que en CUDA podemos obtener un identificador que identifica a cada thread con el que repartimos el trabajo entre los hilos. Sin embargo en OpenCL se obtiene llamando dentro del kernel a una función que proporciona la API. Para obtener la id del bloque actual, utilizaremos **get_num_groups(),** y para obtener la id global del thread, se utiliza la función **get_global_id().**

Los kernel al igual que en CUDA son funciones C, pero tienen ciertas limitaciones:

- Solo pueden acceder a la memoria del *device*, incluso si éste es la CPU, han de copiarse los datos al device para obtener el identificador de memoria OpenCL, aunque internamente la API de OpenCL no haga la copia.
- Deben ser funciones que retornen void.
- No pueden tener un número variable de argumentos.
- No pueden ser recursivos.
- No pueden utilizar variables estáticas.

A continuación explicaremos como lanzar el ejemplo anterior en OpenCL en CUDA.

```
_kernel void square_array(_global float *a, int N)
{
  //Get id thread global
  int idx = get_global_id(0);
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

Código 3: Kernel OpenCL

Nos apoyaremos en el mismo gráfico del apartado anterior:

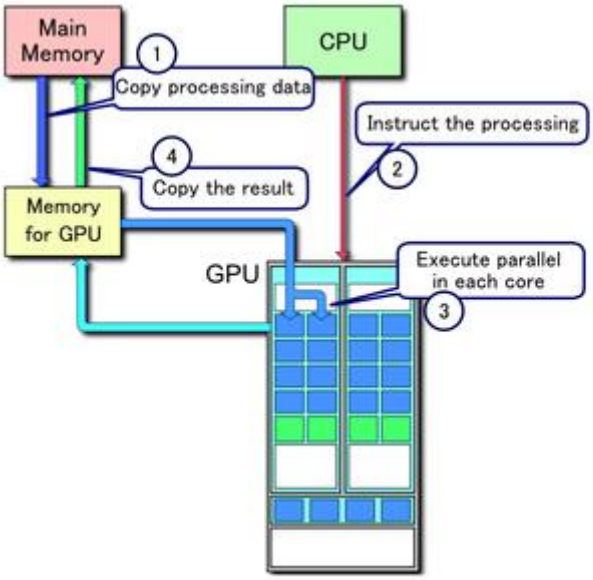


Figura 5: Pasos para lanzar un kernel OpenCL

1- **Copiar los datos a procesar:** Copiar los datos que van a ser procesados en el kernel a la memoria de la GPU, es decir, copiar el array a en la GPU.
   Esto será posible gracias a las instrucciones clCreateBuffer() y clEnqueueWriteBuffer() en las cuales se reserva un espacio en la memoria de la GPU y se copian los datos en el.
2- **Lanzar el kernel:** La CPU llama al kernel, debe llamarlo con la dirección de los datos en la memoria de la GPU que se obtuvo en el paso 1, y debe indicar el tamaño de bloque y tamaño de grid. La llamada se realiza invocando a la función clEnqueueNDRangeKernel().
3- **Ejecutar el kernel:** Este paso es transparente al usuario. OpenCL ejecuta los kernel solicitados en el paso 2. Para ello ha de elegir en que dispositivos de los que dispone en la cola de dispositivos del contexto OpenCL será más conveniente ejecutarlo. Mientras las tareas se están ejecutando la CPU puede hacer otras acciones o simplemente esperar a que finalice la ejecución.
4- **Copiar el resultado:** Una vez que se ha ejecutado el kernel y tenemos el resultado en la memoria de la GPU, hemos de realizar el paso inverso al paso 1, es decir, mediante la instrucción clEnqueueReadBuffer, copiar los datos de la GPU a la CPU para poder seguir trabajando con ellos.

Un ejemplo sencillo comparable con la figura (Código 2) de CUDA de como lanzar este kernel en OpenCL es el siguiente:

```
int main(void)
{
  //Tenemos el código OpenCL del kernel en una variable tipo texto se puede
  //poner como código,o leer de un fichero .cl al igual que se lee cualquier fichero
  Char *codigo="_global square_array(...)\n {...}\n";
  //TAMAÑO
  Const int N=1024;
  //Punteros a la memoria del host y el device
  cl_mem a_d;
  size_t size = N * sizeof(float);
  //Reservar espacio en host e inicializar el host
  float *a_h = (float *)malloc(size);
  for (int i=0; i<N; i++) a_h[i] = (float)i;
  //Crear contexto OpenCL, Device...
  cl_context context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);
  cl_device_id device_id;
  clGetDeviceIDs( NULL, CL_DEVICE_TYPE_DEFAULT, 1, &device_id, NULL );
  cl_command_queue queue= clCreateCommandQueue(context, device_id, 0, NULL);
  //compilamos el código
  cl_program program = clCreateProgramWithSource(context, 1, &codigo, NULL, NULL);
  //Creamos el kernel con el código compilado
  cl_kernel kernel = clCreateKernel(program, "square_array", NULL);

  //PASO 1: Reservar y copiar memoria
  cl_buffer buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, size, NULL);
  //Copiar memoria
  clEnqueueWriteBuffer(command_queue, buffer, CL_TRUE, 0, size * N, a_h, 0, NULL, NULL);
  //Indicar el argumento del kernel
  clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&buffer);

  //PASO2: Lanzar kernel
  int global_work_size[0] = N;int local_work_size[0] = 64;
  clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size, local_work_size, 0, NULL, NUL

  //PASO3: Esperar que se ejecute (no vamos a esperar, nos fiamos de que va rapido
  //realmente habría que utilizar eventos)

  //PASO4: Copiar resultado a la CPU
  clEnqueueReadBuffer(command_queue, buffer, CL_TRUE, 0, sizeof(float) * N, a_h, 0, NULL, NULL);
  // Imprimir resultados
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
  //Limpiar
  clReleaseContext(context);
```

Código 4: Ejemplo de código C que lanza un kernel OpenCL

Como vemos, aunque los pasos y la filosofía explicados son similares a CUDA, el código es mucho más complejo y con dependencias entre unas instrucciones y otras, lo cual causará problemas y provocará cambios en el diseño al implementar OpenCL en Nanox.

2.3 OMPSS

OmpSS [9][10] es una implementación de parte del modelo de programación StarSSBibliografía [11] desarrollada por el BSC. En particular, **su objetivo es extender OpenMP proporcionando nuevas directivas para soportar paralelismo asíncrono y dispositivos heterogéneos** (CPU, GPU…). Aunque también contiene nuevas directivas que extienden otras APIs de programación como CUDA y OpenCL. OmpSS se implementa mediante un compilador (Mercurium) y un runtime (Nanox).

A continuación se definirán varios **conceptos clave relativos a OmpSS**, estos conceptos son totalmente necesarios para poder entender la implementación del proyecto:

1- **Runtime (Nanox):** Runtime que es capaz de ejecutar *tasks* OmpSS.
2- **Compilador (Mercurium):** Precompilador de código a código que transforma sencillas directivas de compilación (#pragma) en el complejo código necesario para que el runtime pueda lanzar una *task* OmpSS.
3- **Directiva (del preprocesador):** Líneas que se incluyen en el código de los programas, pero que no son código sino directivas para el preprocesador. Siempre están precedidas de una almohadilla (#). El preprocesador se ejecuta antes de que comience la compilación, por lo que ha de procesar todas estas directivas antes de compilarlo. Una directiva está compuesta por la propia directiva y cláusulas.
4- **Cláusula:** Parámetros que se añaden a una directiva para provocar diferentes comportamientos de la misma.
5- **Tarea*/task (durante el resto del documento utilizaremos nos referiremos a las mismas como task)*:** Una tarea o *task* es un bloque de código (para este proyecto en el cual se ejecutan kernels, este bloque será una función) que puede ejecutarse en paralelo respecto al código que está fuera de la propia tarea. En OmpSS las tareas tienen dependencias entre ellas con las que se controla el flujo de ejecución.
6- **Target:** Directiva que se introdujo para dar soporte a dispositivos heterogéneos, se utiliza para indicar que la declaración de una task, función o tipo de dato puede ser utilizada en los dispositivos especificados en la misma.
7- **Device:** Dispositivo al que pertenece una task, es decir, en qué tipo de dispositivo se ejecutará (CUDA, OpenCL, SMP…), este device indica que parte del compilador y del runtime han de procesar la función.
8- **Device Descriptor:** En OmpSS una tarea/task puede tener implementaciones para varios devices, es decir, distintas funciones que realizan la misma labor, pero programadas para diferentes dispositivos, nos referimos a device descriptor como el conjunto de datos que una *task* necesita para ejecutarse en ese dispositivo, será construido por el compilador de acuerdo a las directivas para que sean utilizados por el runtime.
9- **Device Provider:** Clase de Mercurium en la que se implementa el código necesario para dar soporte a un *Device* concreto.

10- **Cache:** Conjunto de clases de Nanox encargadas de realizar las copias de memoria en caso de ser necesarias y de proporcionar a las *task* la información necesaria para que la utilicen.

Una vez conocemos los conceptos clave podemos introducirnos más en detalle en el funcionamiento de OmpSS.

Como ya se ha comentado, OmpSS utiliza paralelismo asíncrono, que se realiza mediante tareas relacionadas entre sí a través de dependencias que controlan el flujo de ejecución, estas dependencias pueden ser de tres tipos:

1- **Input:** Dependencias de entrada, indica que la tarea necesita estos datos antes de iniciarse, es decir, no puede comenzar a ejecutarse hasta que todas las tareas declaradas previamente que tengan esta dependencia como output hayan terminado.
2- **Output:** Dependencias de salida, indican que la tarea no podrá ejecutarse hasta que todas las tareas declaradas previamente que tengan esta dependencia como input u output hayan terminado.
3- **Inout:** Dependencias de entrada y de salida, dependencias que se consideran a la vez como **Input** y **Output.**

Cada vez que una *task* se crea se evalúan sus dependencias y se construye un grafo de tareas en tiempo de ejecución. De esta forma una tarea sólo podrá ejecutarse cuando sus predecesores hayan terminado.

Además de poder generar tasks asíncronas, se puede utilizar la **directiva taskwait** (#pragma omp taskwait) para parar el thread principal, de forma que este espere a que terminen todas las tareas que se han creado previamente.

Para ayudar a entender el funcionamiento de lo explicado anteriormente nos apoyaremos en el siguiente ejemplo:

```
void foo ( int *a, int *b )
{
   for ( i  = 1; i < N; i++ ) {
      #pragma omp task input(a[i-1]) inout(a[i]) output(b[i])
         propagate(&a[i-1],&a[i],&b[i]);

      #pragma omp task input(b[i-1]) inout(b[i])
         correct(&b[i-1],&b[i]);
   }
}
```

Código 5: Bucle utilizando tareas OmpSS

Este bucle generará el siguiente grafo en el runtime:

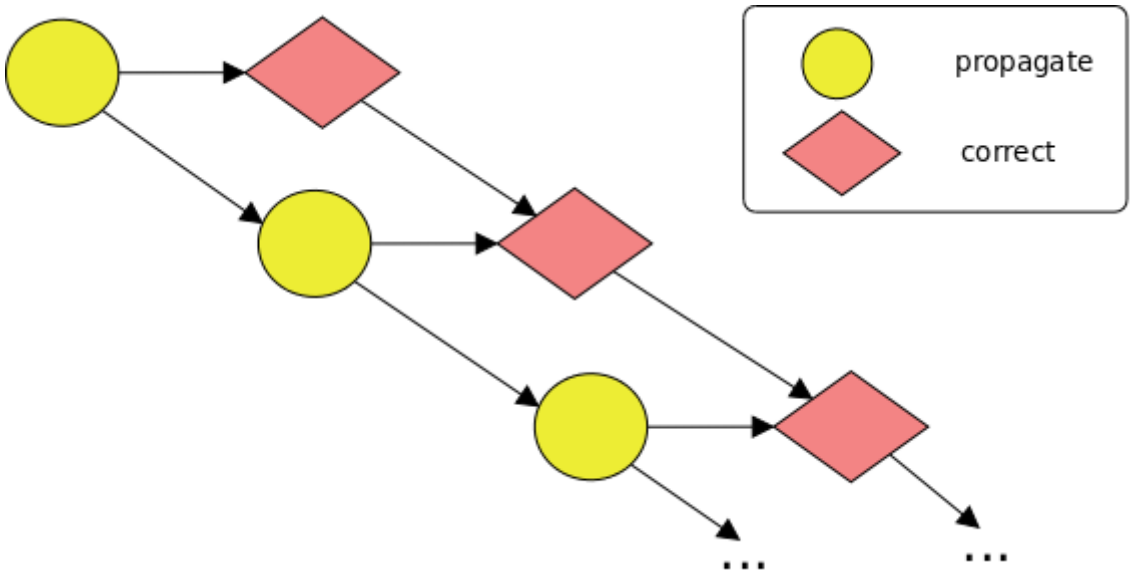


Figura 6: Grafo de dependencias OmpSS

Si se ejecuta el bucle, se ejecutarán todas las iteraciones generando todas las *task* (dos por cada iteración) y estas se ejecutarán en el orden establecido por sus dependencias.

En el grafo podemos ver como en la primera iteración se crean las *task* "propagate" y "correct". En cada iteración la *task* "correct" depende de la *task* "propagate" de su misma iteración por la dependencia inout b[i], y a su vez depende del correct de la iteración anterior, ya que tiene como input b[i-1], que depende del out b[i] anterior.

La *task* “propagate” depende de la *task* “propagate” de la iteración anterior mediante la dependencia a[i] como inout y el input a[i-1].

De esta forma, como se ve en el gráfico, conseguimos ejecutar con cierto paralelismo un bucle con dependencias entre iteraciones de una forma sencilla.

A continuación explicaremos en detalle de que dos partes se componen OmpSS.

MERCURIUM

Mercurium es un compilador de código a código orientado a prototipado rápido. Soporta los lenguajes de C y C++, y está siendo desarrollado para Fortran. Principalmente se utiliza en entornos OmpSS para implementar OpenMP para el runtime Nanox, pero dada su escalabilidad se ha utilizado para implementar otros modelos de programación, como Cell Superescalar y el proyecto ACOTES.

Extender Mercurium se consigue utilizando una arquitectura de plugins, donde estos plugins representan diferentes fases de compilación (OpenCL y CUDA se implementan como dos de estos plugins). Estos plugin se escriben en C++ y se cargan dinámicamente dependiendo de la configuración elegida. **Las transformaciones de código se implementan en forma de código fuente**, es decir, Mercurium abstrae al programador (hasta cierto nivel) de conocer la representación sintáctica del compilador.

El **compilador internamente está dividido en front-end**, que utiliza bison para transformar el código que procesa en estructuras de código que utiliza el back-end. Y en **back-end** en el que se implementan los plugins y las fases sobre las que trabajaremos tanto en CUDA como en OpenCL.

En la figura 7 se pueden ver los pasos que sigue Mercurium para compilar código.

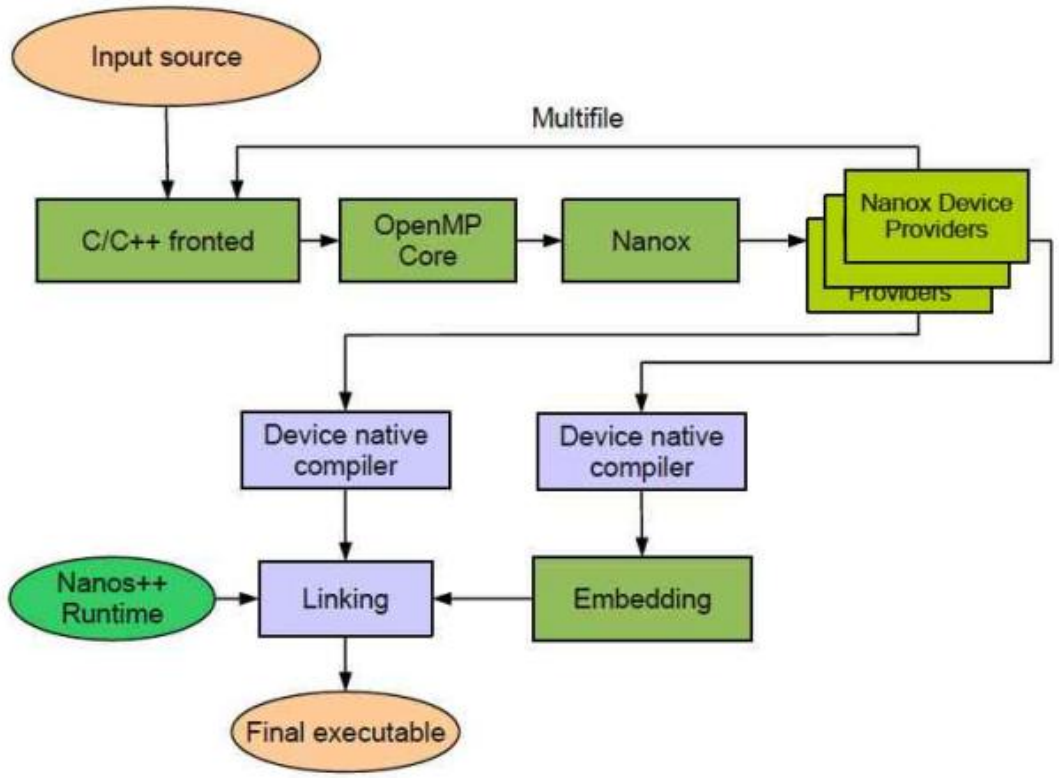


Figura 7: Esquema compilación Mercurium

Como entrada recibe el código fuente que ha de compilar. Este código es preprocesado por el front-end de Mercurium (bison). Tras preprocesarlo se realizan unas transformaciones intermedias al código (OpenMP Core y Nanox). Una vez realizadas estas transformaciones el código ha de ser tratado por cada *Device Provider* que realizará unas acciones concretas para cada *device* (CUDA, OpenCL, SMP…).

Una vez Mercurium procesa todos los ficheros de entrada, ha de compilar el código fuente de salida que ha generado en los pasos anteriores. Esto se realiza llamando al compilador necesario para compilar el código generado, P. Ej.: nvcc en CUDA, gcc/icc en SMP, gcc/icc con el parámetro "–lOpenCL" para OpenCL.

Tras compilar el código, se realiza el proceso de enlazado, entre las librerías y código que especifica el usuario, a lo que Mercurium añade las librerías del runtime Nanox. Tras terminar este proceso, Mercurium genera el ejecutable (fichero .out) desde el cual se puede lanzar el proceso.

Este ejecutable es totalmente portable, y únicamente ha de disponer del runtime Nanox para ser lanzado, es decir, se puede compartir entre diferentes máquinas si estas tienen instalado Nanox, sin necesidad de que tengan instalado Mercurium.

NANOX

Nanos++ o Nanox es un runtime diseñado para dar soporte a entornos paralelos (principalmente OmpSS).

Nanox proporciona servicios para soportar paralelismo a nivel de tareas utilizando sincronización basada en dependencias de datos. Estas tareas se implementan como hilos de ejecución cuando sea posible.

Además Nanox proporciona soporte para mantener la coherencia entre diferentes espacios de memoria (como GPUs).

El principal objetivo de Nanox es ser utilizado para desarrollar entornos paralelos, por ello es extensible mediante plugins. Actualmente se pueden añadir plugins que realicen diferentes funciones:

- Políticas de planificación de tareas.
- Barreras entre threads.
- Soporte a otros devices
- Instrumentación
- Asignación de recursos.

Algunos de estos plugins ya están desarrollados, como algunos dedicados a la de planificación de tareas o como el soporte para el device CUDA. También están en desarrollo plugins para OpenCL y clustering.

Además Nanox se puede instrumentar con la librería extrae, que produce trazas que se pueden analizar con la aplicación Paraver.

Sin embargo, Nanox no está diseñado para ser utilizado directamente, sino para ser utilizado a través de uno de los modelos de programación que facilite su uso. Los cuales están soportados mediante el compilador Mercurium.

CAPÍTULO 3. EXTENSIÓN DE MERCURIUM PARA CUDA

En este capítulo se procederá a describir con detalle los pasos que se han realizado para mejorar la implementación de CUDA ya existente en Mercurium. Cabe destacar que tanto los objetivos como la manera de alcanzarlos han estado abiertos a discusión durante la realización del proyecto.

3.1 ANÁLISIS PREVIO

Antes de comenzar hemos de aclarar que ya existía una versión anterior de OmpSS para ejecutar CUDA. Sin embargo, se quiere mejorar para que sea más transparente al programador, para ello hemos de revisar el estado ya existente de la misma y evaluar cual es el estado al que hemos de llegar a la finalización de esta parte del proyecto.

Revisando el código del *software* OmpSS que hemos recibido desde el BSC, comprobamos que la versión actual de CUDA es funcional. Sin embargo no abstrae totalmente al programador de la API de nVidia, ya que es el programador el que ha de generar el código que llama a los kernel/funciones CUDA.

```
#pragma omp target device(cuda)
__global__ void calculate_forces_kernel_naive(float time_interval, Particle* d_particles,
                                               int number_of_particles, Particle *output,
                                               int first_local, int last_local) {
    size_t id = (blockDim.x * blockIdx.x) + threadIdx.x + first_local;
    ....
}


#pragma omp target device(cuda) copy_deps
#pragma omp task output([size] output) \
                input(d_particles[number_of_particles])
void cuda_wrapper(int size, int bs, float time_interval, int number_of_particles,
                  Particle* d_particles, Particle *output, int first_local, int last_local){
        size_t num_blocks;
        size_t num_threads;
        num_threads = size < bs ? size : bs;
        num_blocks = size / bs + (size % bs == 0 ? 0 : 1);
        calculate_forces_kernel_naive<<<num_blocks,num_threads>>> (time_interval, number_of_particles,
}
```

Código 5: Cabecera de un kernel CUDA y task que invoca al kernel.

Como se puede ver en la figura (Código 5), tenemos dos funciones. La primera (calculate_forces_kernel_naive) con una directiva (#pragma) simple que indica que pertenece al *device* CUDA, nos referiremos a estas funciones como **kernel**.

La segunda función (cuda_wrapper) es una *task* OmpSS, en la cual el programador tiene que indicarle a OmpSS cuáles son las dependencias (input y output) y las copias que necesitará el *device* en su memoria (en este caso concreto, se pone copy_deps, ya que al ser las mismas, se le indica al runtime que tiene que copiar las dependencias). Además tiene que realizar la llamada al kernel manualmente al igual que lo hiciera en CUDA nativo.

El resultado de esto, es que cuando el programador llame a la función cuda_wrapper, se ejecutará el kernel CUDA deseado. En este caso el runtime será el encargado de asegurar que los parámetros que se utilizan en el kernel estén disponibles en los *devices* (GPU) donde se va a ejecutar el kernel, así como decidir en qué *device* se ejecuta el mismo.

El principal objetivo de la parte inicial del proyecto será el de ahorrar al programador el código necesario para llamar a la función CUDA, es decir, que el compilador sea capaz de generar el código automáticamente, además de obtener el número de bloques y threads necesarios para la ejecución. De esta forma conseguimos abstraerle del código necesario para llamar al kernel. Evitar que la llamada a CUDA requiera un código especial cobrará mayor importancia cuando en la segunda parte del proyecto se añada la funcionalidad OpenCL, ya que la forma de llamar a los kernel será similar en ambos casos.

3.2 DISEÑO

FASE 1

Una vez que conocemos el objetivo principal que es abstraer al programador de la API de CUDA, se ha de diseñar una forma de llamar a los kernel CUDA que sea cómoda para el programador. Siempre teniendo en cuenta que el compilador ya tiene una estructura predefinida para los ***devices*** que extienden su funcionalidad, los cuales **implementan una interfaz** a la que el compilador va llamando en diferentes fases.

Lo primero que se ha de hacer es evaluar qué puede generarse automáticamente y qué nos tiene que proporcionar el programador. A primera vista tenemos dos cosas claras, necesitamos el nombre del kernel que queremos llamar, y también necesitamos conocer el número de bloques y threads en los que queremos lanzar el kernel.

El pragma, compuesto por cláusulas, es la directiva que Mercurium preprocesa y a partir de la cual podemos obtener toda la información. Se coloca sobre una declaración o definición del kernel, por lo que a través del compilador podemos obtener el nombre del kernel. Sin embargo, el número de bloques y threads en los que el programador quiere lanzar el kernel no podemos deducirlo de ninguna manera a través del compilador, por lo cual el programador tendrá que proporcionarnos esta información a través de cláusulas en la directiva.

Además de lo anterior, este kernel implementa una función "puente", que es una función SMP a la que el programador podrá llamar y la que el compilador definirá para llamar al kernel.

```
#pragma omp target device(smp) copy_deps
#pragma omp task output([size] output) \
                    input([number_of_particles] d_particles)
void calculate_force_func(int size, int bs, float time_interval, int number_of_particles,
                          Particle* d_particles, Particle *output, int first_local, int last_local);

#pragma omp target device(cuda) implements(calculate_force_func) ndrange(1,size,bs)
__global__ void calculate_forces_kernel_naive(float time_interval,  int number_of_particles,
                                              Particle* d_particles, Particle *output,
                                              int first_local, int last_local) {
    size_t id = (blockDim.x * blockIdx.x) + threadIdx.x + first_local;
    if (id > last_local ) return;
```

Código 6: Prototipo inicial del kernel

Como se puede comprobar en el Código 6, tendremos una *task* "smp" (una función normal en C) que es la que el programador invoca, y un kernel CUDA que tiene el pragma que hemos de diseñar. A continuación se explica la sintaxis y semántica del "pragma".

1- *Device* (CUDA), para indicar que ese kernel/función es un kernel CUDA.
2- *Implements* (función), para indicar que ese kernel implementa la función/*task* padre que se indique (en la cláusula).
3- *Ndrange* (el nombre se ha elegido para mantenerlo similar a OpenCL). Cláusula a la cual se le ha de pasar, en este orden y separado por comas, los siguientes valores:
    a. Número de dimensiones: número de dimensiones en las que se lanzará el kernel, valores posibles (1 (x), 2 (x, y) o 3 (x, y, z)).
    b. Total size*: vector con el tamaño total de los datos a procesar, el número de bloques del kernel será total size/local size.
    c. Local size*: vector con el número de threads que se ejecutarán por cada bloque.
    d. noCheckDim (opcional): Mercurium automáticamente realizará unas funciones de redondeo/rounding a la hora de determinar el número de bloques y threads en función de los valores del ndrange, en caso de que el usuario prefiera no realizar ningún tipo de redondeo y que los valores del ndrange se usen sin comprobar/cambiar nada, se puede especificar este argumento.

*Tanto total size y local size tienen que tener una longitud igual a numero de dimensiones (cada dimensión tiene su total size y su local size).

Sin embargo este prototipo inicial presenta dos problemas:

1- Para saber qué argumentos se necesitan pasar de la función "puente" al kernel de CUDA, se utilizan los que tienen el mismo nombre y el mismo tipo, es decir, si ambos tienen un argumento "int local_size", se utilizará el parámetro local_size de la función puente para el kernel de CUDA. Sin embargo, el usuario puede querer cambiar este comportamiento, para ello se añadirá la **cláusula calls** al pragma (explicada posteriormente).
2- **Mercurium no soporta *task*s sin definir**, es decir, el prototipo anterior funciona bien, si además de los kernel CUDA, tenemos una función SMP que realiza la misma *task*, de forma que tendríamos la función definida para dos *device* diferentes. Pero se puede dar el caso (de hecho es lo más común) de que el programador solo quiera tener una implementación CUDA sin tener ninguna otra, para lo cual hay que encontrar la forma de modificar Mercurium para que soporte un ***device* "indefinido"** (cuyo significado es que esa *task* no tiene definición propia, solo implementaciones).

Para solventar estos dos problemas se propone un prototipo nuevo muy similar al inicial:

```
#pragma omp target device(undefined) copy_in([number_of_particles] d_particles) copy_out([size] output)
#pragma omp task output([size] output) \
                input(d_particles[number_of_particles])
void calculate_force_func(int size, int bs, float time_interval, int number_of_particles,
                          Particle* d_particles, Particle *output, int first_local, int last_local);
#pragma omp target device(cuda) implements(calculate_force_func) ndrange(1,size,size) \
calls (time_interval, number_of_particles, d_particles, output, first_local, last_local)
__global__ void calculate_forces_kernel_naive(float time_interval,  int number_of_particles,
                                              Particle* d_particles, Particle *output,
                                              int first_local, int last_local) {
    size_t id = (blockDim.x * blockIdx.x) + threadIdx.x + first_local;
```

Código 7: Prototipo fase 1 CUDA

En este prototipo añadimos al anterior la **posibilidad** (opcional) de indicar con qué parámetros queremos que se llame al kernel (cláusula calls) y vemos que la función calculate_force_func no necesita tener definición (*device* "undefined"). Este prototipo es el que se implementará durante la Fase 1 de la implementación.

FASE 2

Una vez realizada la implementación de la fase 1, tras una pequeña estancia en el BSC en la cual se presentaron los cambios realizados. Aunque el estado de la fase 1 estaba correcto y se implementaría en la rama principal de Mercurium, se propuso una posible mejora. Esta mejora consiste en que se podría intentar que el *device* CUDA sea totalmente transparente al usuario y se utilice como cualquier otro *device*, pudiendo **definir los kernel CUDA como *task*s directamente** (sin funciones puente).

El objetivo es ser capaces de ejecutar y compilar el siguiente prototipo, además de que las *task*s CUDA puedan usarse como cualquier otro *device* (ser una tarea en sí misma, que pueda implementadas por otros devices, implementar a otros…).

```
#pragma omp target device(cuda) ndrange(1,size,MAX_NUM_THREADS) copy_deps
#pragma omp task output([size] output) input([number_of_particles] d_particles)
__global__ void calculate_force_func(int size, float time_interval,  int number_of_particles,
                                     Particle* d_particles, Particle *output,
                                     int first_local, int last_local) {
    size_t id = (blockDim.x * blockIdx.x) + threadIdx.x + first_local;
```

Código 8: Prototipo fase 2 CUDA

Con todo esto**, hemos conseguido diseñar una directiva de precompilación que permita al compilador generar la mayor cantidad de código automáticamente**. La directiva estará compuesta por una serie de cláusulas, que proporcionarán la información que el compilador no puede extraer de forma automáticamente.

3.3 IMPLEMENTACIÓN[2]

Antes de comenzar a implementar, hemos de familiarizarnos con la API del runtime para CUDA, ya que es a la que deben llamar todas las *task*s que genera el compilador. **Las modificaciones necesarias únicamente hemos de realizarlas en el back-end del compilador.**

Además de todas las dependencias que controla Mercurium para cualquier tipo de *task* independientemente del *device* que tenga (copias, dependencias de entrada/salida…), cada *device* debe proporcionarle al runtime un descriptor (puntero a una función) en el cual el programador calcula los parámetros del ndrange y realiza la llamada al kernel CUDA (a posteriori será el runtime el encargado de ejecutarlo en el momento adecuado).

Por otra parte se debe conocer el *DeviceProvider* de Mercurium, es decir, la clase que tenemos que implementar. A priori su funcionamiento es simple: cuando Mercurium detecta un pragma, lo procesa, y en función del *device* (o *devices*) que vengan indicados en esa *task*, este *device* define unas funciones a las que llama Mercurium.

A continuación se describirán esas funciones, ya que sobre ellas se implementarán la mayoría de los cambios:

1- **Insert_function_definition/insert_declaration:** Funciones que se llaman cuando Mercurium lee cualquier tipo de "*#pragma omp target device"* (CUDA). Como parámetro se pasa un objeto que representa el pragma y a partir del cual se puede obtener información sobre la directiva, además de sobre la función o declaración a la que se refiere.
2- **Create_outline**: Esta es la función que más trabajo requiere por parte de nuestro *device*, ya que es la encargada de generar las funciones intermedias para que sean llamadas por la *task* de Mercurium. En otros *devices* esto puede ser trivial, ya que básicamente hace *renaming* de funciones, sin embargo en nuestro caso hay que realizar varias acciones:
    a. Se ha de hacer una funcionalidad similar a la implementada insert_function_definition (ya que la *task*, también es una funcion, y hemos de llevar su código al fichero CUDA).
    b. Se ha de crear la función SMP/puente que es la encargada de lanzar los kernel CUDA (y no solo crearla, sino generar su contenido).
3- **Get_*Device*_Descriptor:** En esta función únicamente se genera el descriptor del *device* (en este caso un simple puntero a la función que se genera en create_outline).
4- **Do_replacements**: Esta función se encarga de generar código necesario para llamar desde una función SMP que genera Mercurium y la *task* que implementa el programador. Para el desarrollo de este proyecto no será modificada.

[2] En los títulos correspondientes se encuentran referenciados los adjuntos implementados.

FASE 1

Como se ha visto en el diseño, en la implementación anterior de CUDA, el programador tiene que escribir una *task* que realiza la llamada al kernel CUDA. En nuestra implementación actual, es el compilador el encargado de generar esta *task*. Conociendo los detalles anteriormente descritos vemos que la primera función que a priori debemos modificar es la función **Create_outline (2).**

Revisando la implementación ya existente vemos que la función básicamente recibe de Mercurium una llamada a la función que el programador ha implementado (que llama al kernel CUDA). El *device* se encarga de buscar la función a la que se llama y junto con la llamada eliminarlas del fichero de fuentes original (.c) y guardarlas en el fichero de fuentes CUDA.

En la versión anterior se hace una llamada CUDA en 3 pasos:

1- La *task* de Mercurium llama a una función intermedia.
2- La función intermedia se encarga de llamar a otra función SMP*.
3- La función SMP (que es la que define el usuario) lanza los kernel CUDA.

Nuestro objetivo será el siguiente:

1- La *task* de Mercurium llama a una función intermedia.
2- La función intermedia lanzará el kernel CUDA*.

*El puntero a esta función es el que se envía al runtime para que lo ejecute cuando estime oportuno.

Para conseguir este objetivo, lo que se hará es sustituir la llamada a la función que Mercurium nos proporciona (punto 2), con el código necesario para lanzar los kernel. Código que tendremos que generar nosotros, ya que la función SMP (punto 3) no ha de ser definida por el usuario.

Comprobando el estado en que llega la *task* cuando Mercurium invoca a la función **Create_outline (2)** vemos que no tenemos información relativa al kernel (excepto el nombre), sino que tenemos información relativa a la *task* intermedia que genera Mercurium (que durante esta fase de la implementación suponemos que no conocemos los detalles de cómo son generadas). Y de la cual no se puede obtener la información requerida.

Sin embargo con el análisis realizado anteriormente y el diseño, en el cual se especificó que los kernel CUDA se declaran como target con las cláusulas implements y ndrange, que proporcionan información necesaria para generar la llamada al kernel, sabemos que todas los target son tratados mediante la función **Insert_function_declaration(1).** Por ello hemos de generar el código que llama al Kernel en esta función (aunque para facilitar el mantenimiento del código crearemos una función nueva (**generate_ wrapper_code**) que se invoca desde ella).

Este código será almacenado para que cuando la *task* deba crear la función (**create_outline (2)**) podamos generar el código para lanzar el kernel. Para ello lo almacenamos en una estructura de tipo Clave > Valor cuya clave está compuesta por el nombre del kernel y el nombre de la *task* a la que implementa.

IMPLEMENTACIÓN DE LA FUNCIÓN GENERATE_WRAPPER_CODE

Como ya se ha explicado anteriormente, esta función se encarga de generar el cuerpo de una función que llama a un kernel CUDA.

Se le pasan únicamente 4 argumentos. La declaración de la *task* (que no es más que una función de C), la declaración del kernel CUDA, y el contenido de las cláusulas ndrange y calls. El objetivo de esta función es generar un código similar al siguiente:

```
dim3 dimGrid;
dim3 dimBlock;

dimGrid.x = 4 ; dimGrid.y = 1; dimGrid.z = 1;
dimBlock.x = 64; dimBlock.y = 1; dimBlock.z = 1;

kernel<<<dimGrid,dimBlock>>>(A);
```

Código 9: Lanzamiento de un kernel en CUDA nativo de manera simple.

Como se puede ver en el Código 9 es necesario conocer las dimensiones (x, y, z) con las que se lanzará el kernel, las cuales se obtienen gracias a la cláusula ndrange (que proporcionará el programador), el nombre del kernel y los parámetros con los cuales se hará la llamada, los cuales podemos obtenerlos utilizando las dos declaraciones que se pasan a la función.

Comenzamos **generando el código ndrange** (dimensiones) que es lo más simple. Para ello recibiremos una cláusula con los siguientes parámetros (num_dimensiones, global_size[3], local_size[4]). Un ejemplo de esta para una sola dimensión lo tenemos en el **Código 8.** Para num_dimensiones 1, hemos de calcular la cantidad de bloques (dimGrid), para lo cual se divide el tamaño total (global_x) entre el numero de threads por bloque (dimBlock=local_x). Además de aplicar algunas funciones de redondeo que son necesarias a la hora de lanzar kernels, para corregir tamaños que no sean divisibles.

[3] - Vector de tamaño num_dimensiones.

[4] - Vector de tamaño num_dimensiones.

Siendo este el resultado (por simplicidad se utilizan valores inmediatos, 64 y 16):

```
dim3 dimGrid;
dim3 dimBlock;
dimBlock.x = ((64) < (16) ? (64) : (16));
dimGrid.x = ((64) < (16) ? 1 : (64) / (16) + ((64) % (16) == 0 ? 0 : 1));
dimBlock.y = 1;
dimGrid.y = 1;
dimBlock.z = 1;
dimGrid.z = 1;
```

Código 10: Código generado por Mercurium a partir de la cláusula ndrange(1,64,16).

Hay que tener en cuenta que normalmente en lugar de números directamente, se utilizarán parámetros de la propia función que será necesario renombrar de acuerdo al *renaming* que hace Mercurium con las *task* (y que aún no conocemos, por lo cual en esta fase de la implementación, las identificamos con un identificador propio) o constantes de las que tendremos que buscar su declaración y llevarlas al fichero CUDA. Estos dos problemas se solucionarán en la función **create_outline(2)** ya que en esta fase aún no disponemos de la información suficiente para solucionarlo.

Una vez generado el código del ndrange, hemos de **generar el código de la llamada al kernel** (calls) para lo cual hemos de obtener el nombre del kernel (el cual se puede obtener sin problemas de la declaración) y realizar un cruce entre las declaraciones del kernel y la *task*/función SMP para saber qué parámetros han de pasarse al kernel. El criterio a seguir será el siguiente:

1- Si está definida la cláusula calls, se utilizarán los valores que el programador indica en ella. Aún así, hemos de procesarlos y renombrarlos ya que cuando nos llegan por parte de Mercurium, no se llaman igual que lo que el programador indica en la directiva. La problemática es la misma que con el ndrange.
2- Si no está definida la cláusula calls, se comprobará que el argumento del kernel y el de la *task* se llaman igual y tienen el mismo tipo. Si esto se cumple, se utilizará el argumento de la *task* como argumento del kernel.

Tras esto se guardará el código generado en nuestra **lista de códigos de invocación Clave > Valor** que será utilizada más tarde, cuando las *task* de Mercurium pidan que se genere la función para llamar al kernel.

CAMBIOS REALIZADOS EN LA FUNCIÓN CREATE_OUTLINE (2)

Ahora que se ha solucionado el problema de no disponer de la información necesaria para lanzar el kernel, se puede modificar esta función para generar las llamadas a los kernel automáticamente. Se tiene la posibilidad de obtener el nombre de la función implementada y el nombre del kernel (target), lo cual se utiliza como clave en la **lista Clave > Valor con los códigos lanzadores.**

Dado que se tiene que mantener la compatibilidad con la versión anterior de CUDA, si se encuentra en la lista el código necesario para lanzar kernels, realizaremos el comportamiento "nuevo" y en caso de que no encontremos el código para lanzar el kernel, suponemos que estamos en el comportamiento de la versión anterior de OmpSS.

Como explicamos anteriormente, la diferencia de comportamiento es básicamente que en la versión anterior la función únicamente llamaba a otra función SMP (previamente implementada por el programador) que era la que llamaba al kernel.

En la versión actual, recogemos el código que lanza el kernel, y realizamos las conversiones anteriormente mencionadas que quedaron almacenadas como "identificadores especiales" ya que no disponíamos de su nombre real (nombre real que proviene del renombrado de Mercurium).

Ahora que ya tenemos el código final, en lugar de llamar a una función SMP (que además no existe, ya que el programador no la ha implementado), sustituimos esa llamada por el código generado automáticamente.

De esta forma, en lugar de pasarle a **NANOX** el puntero a una función que llama a otra que el programador ha implementado y lanza los kernel CUDA, le pasamos el puntero a esa misma función, pero ahora esta función llama directamente a los kernel, sin necesitar que el programador haya implementado nada más.

IMPLEMENTACIÓN *DEVICE* "UNDEFINED"

Todo lo explicado anteriormente, se ha basado en la suposición de que tenemos una *task* Mercurium ya creada, con cualquier otro *device* (smp en nuestro caso). Pero como hemos mencionado anteriormente, es muy posible que no tengamos ninguna otra implementación de la *task*, y no podemos dar una implementación vacía. Ya que Nanox para elegir entre un *device* u otro elige la más rápida, y si le damos una implementación vacía, en el momento que Nanox vea que una función tiene varias implementaciones para varios devices, intentará utilizar la más rápida, y si una de ellas es una función vacía, asumirá que esa es la más rápida. Por ello lo que necesitamos es añadir soporte a Mercurium para utilizar un *device* "sin definición", es decir, que nos permita declarar una tarea sin tener ningún device.

Para ello la forma más fácil de hacerlo es modificar el código que se encarga de procesar las *task*s (y llamar a los *Device*Provider), y en el momento que va a llamar a los *Device*Provider, conseguir que si el nombre del mismo es "undefined", no se genere el código necesario para ese *Device*Provider (de esta forma, la *task* queda incompleta, para completarse cuando se procesen los implements de CUDA). Además se han de añadir los controles pertinentes para asegurarse de que nadie defina una función con definición como "undefined" (o al menos lanzar un aviso), o que ninguna *task* se quede sin ninguna definición ni implementación.

Con esta modificación completamos la fase 1, ya que **hemos conseguido que se puedan lanzar kernels utilizando la cláusula implements sin que el programador tenga que realizar la llamada manualmente**.

FASE 2

Una vez probada la funcionalidad de la Fase 1, la cual ha sido evaluada positivamente, como se verá en el capítulo 5 (**Pruebas CUDA**), se aborda la Fase 2.

Aquí vemos una diferencia principal, las *task*s pueden estar anotadas como CUDA directamente, sin necesidad de utilizar como puente una *task* SMP/undefined a la que llamamos. Además de soportar implements, al igual que se hace con los demás *device* (smp, smp-numa y otros) de Mercurium.

Para ello vemos que nuestra aproximación inicial de generar el código en la función **insert_function_definition/declaration (1)** no es suficiente, ya que las *task*s no son procesadas por el *device* provider directamente, sino preprocesadas por Mercurium. El cual solicita al *device* provider que realice diversas acciones suponiendo (incorrectamente para nuestros *Device*, aunque es correcto para SMP por ejemplo) que tenemos la función ya disponible en nuestro código.

Para ello hay que realizar varias modificaciones en la función **create_outline (2)**, que vuelve a tener el mismo problema que antes. En el momento que nos llaman, no tenemos el código capaz de lanzar el kernel, a lo que hay que añadirle que el kernel aún no lo tenemos disponible en el código CUDA.

Previamente a modificar la función, hemos de conseguir almacenar en las *task*s la cláusula original del #pragma omp target (al estar definido como *task*, cuando Mercurium nos llama, ya ha modificado gran parte del pragma, por lo cual tenemos que almacenar la cláusula original). Esto lo hacemos **añadiendo** a las clases que almacenan **información de las *task*s**, y a los objetos que contienen esa información el código necesario para guardar también esta cláusula (al igual que se guardan otras cláusulas, como los copy_in,copy_out…)

CAMBIOS EN LA FUNCIÓN CREATE_OUTLINE (2)

Ahora que ya conseguimos guardar el ndrange (que es necesario para generar nuestro código que lanza el kernel) para acceder desde esta función, podemos modificarla. Ya que tenemos que mantener compatibilidad con la versión anterior de CUDA y Mercurium no es capaz de diferenciar entre una *task* CUDA que sea un kernel, y una *task* CUDA que sea un lanzador implementado por el programador (para el compilador son lo mismo, una directiva y una declaración/definición de función). Utilizaremos la existencia o no de la cláusula ndrange (requerida en el caso de que sea un kernel) en la *task* para distinguir si hemos de realizar el comportamiento de la versión anterior de OmpSS y nuestro comportamiento.

Una vez realizados los cambios anteriores, se ha tenido que refactorizar el código, ya que con la nueva implementación, se repiten comportamientos para tratar funciones que provienen de una directiva #pragma omp target y una directiva #pragma omp *task*.

En definitiva, tras finalizar la implementación de estas funciones **obtenemos un Device Provider que es capaz de almacenar el código CUDA del usuario, eliminarlo de los ficheros .c/.cpp(para que estos compilen correctamente con gcc), y de proveer a Mercurium un *device descriptor*** que apunta a una función auto-generada que es capaz de ejecutar un kernel CUDA utilizando todas las mejoras que aporta OmpSS (cache, dependencias entre tareas…). **Con ello conseguimos ejecutar el Código 8.** Es decir, hemos conseguido que se puedan anotar kernels CUDA como tasks Mercurium.

CAPÍTULO 4. EXTENSIÓN DE OMPSS PARA OPENCL

En este capítulo explicaremos con detalle cómo se realizó la implementación OpenCL para OmpSS (Mercurium+runtime). Al igual que en CUDA, los objetivos estaban totalmente abiertos a discusión.

4.1 ANÁLISIS PREVIO

A diferencia del caso anterior en el que ya existía una primera implementación para CUDA, OmpSS no tiene soporte para OpenCL, por lo tanto este *device* hemos de realizarlo sin tener una base clara. Aunque tiene ciertas similitudes con CUDA y comparten gran parte de los problemas.

Uno de los requisitos que tenemos es que **una llamada OpenCL tiene que parecerse lo máximo posible a CUDA**, de forma que el trabajo que necesita hacer un programador de aplicaciones que utiliza OmpSS, para portar un proceso de CUDA a OpenCL sea el mínimo necesario.

El objetivo que tenemos para implementar OpenCL es replicar el comportamiento de los *devices* CUDA, lo cual no entraña demasiada dificultad después de haber extendido el *device* CUDA modificándolo casi en su totalidad, sin embargo, hemos de tener en cuenta que tendremos que comunicarnos con la versión OpenCL de Nanox, lo cual producirá varias diferencias a la hora de realizar la implementación.

En esta ocasión no todo se reduce al compilador, ya que Nanox no soporta OpenCL, o al menos no en su versión pública, únicamente lo soporta en una versión en desarrollo que se está realizando en otro proyecto diferente.. Lo cual es bastante problemático, porque el desarrollador del otro proyecto no conoce OmpSS ni Mercurium, y además su trabajo no está orientado a ellos, si no a realizar un wrapper (Virtual-OpenCL) que simule la API OpenCL, y transforme las llamadas OpenCL para funcionar mediante Nanox (es decir, integrar Nanox con la API de OpenCL).

En resumen, **hemos de replicar** (con algunas diferencias) **el *device* de CUDA y adaptar el runtime ya existente de OpenCL para que sea compatible con OmpSS,** este Runtime estará basado en el existente en el proyecto VirtualCL**.**

4.2 DISEÑO

RUNTIME

En OpenCL comenzaremos por el diseño del Runtime, ya que el compilador ha de realizar las llamadas necesarias para que el Runtime ejecute el código OpenCL deseado.

El primer paso necesario es realizar una revisión del estado del Runtime actual. Para ello aparte de revisar el código se estableció contacto con el desarrollador mediante videoconferencia.

De la conferencia obtuvimos diversas conclusiones, la principal diferencia en las cache es que en su implementación el programador especifica las copias de los argumentos explícitamente, tal y como se hace en OpenCL nativo. Además, en los parámetros de la *task* en lugar de mandar el puntero a una función que lanza un kernel (como en el caso de CUDA y SMP), hemos de pasar diversos parámetros (código, nombre del kernel, información de ndrange, argumentos del kernel….), todos estos parámetros se utilizarán en el Runtime para lanzar el kernel.

Por ello no se puede seguir el esquema de CUDA para lanzar un kernel, ya que la el código necesario para realizar una llamada a un kernel OpenCL es más compleja que en CUDA y necesita de información que sólo se conoce en Runtime.

Esto causa otro problema adicional a Mercurium, y es que al enviar unos parámetros “no estándar”, en caso de que Mercurium quiera realizar una *task* con varias implementaciones (CUDA, SMP y OpenCL por ejemplo), los parámetros van a ser diferentes en una y en otra, y en la API de Nanox esto no está soportado.

La solución adoptada tras un profundo análisis es **modificar el front-end del Runtime, para que ofrezca un *device* OpenCL especial** que reciba los parámetros en una zona propia, diferente a los demás *devices*, utilizando parte del back-end del desarrollo ya existente.

En el back-end de Nanox, parece **que únicamente tendremos que modificar el sistema de cache**, ya que en la implementación ya existente, los datos se copian a petición del programador (al igual que ocurre en OpenCL). Y al realizar la implementación a través de Mercurium, los datos no los copia el usuario de manera explícita, sino que Mercurium debe manejar la memoria, apoyándose en las cláusulas copy_in y copy_out. Además hemos de modificar la parte del back-end que recoge las direcciones de cache en la GPU para utilizarlos en los kernel.

Debido a que la estructura de las partes OpenCL y CUDA en el compilador es similar, aunque realmente se hizo en dos fases, por abreviar se explicará directamente el prototipo final, ya que los cambios de la fase 2 (en la estructura) son similares a los de CUDA.

El diseño de la parte OpenCL ha sido bastante más sencillo que el de CUDA, ya que hemos de mantener una forma común para llamar a funciones tanto CUDA como OpenCL. Por lo cual únicamente tenemos que adaptar las llamadas de CUDA a las posibilidades que ofrece OpenCL.

La principal diferencia en la directiva está en la cláusula **ndrange** que en CUDA recibe los parámetros (num_dim, global_size, local_size). Sin embargo, en OpenCL puede recibir un parámetro más, el **offset,** obteniendo el siguiente prototipo ndrange(num_dim,offset,global_size,local_size). Dado que en muchos casos el valor más común del offset es 0, y puede ser que la llamada haya sido "portada" desde una *task* CUDA, si el offset no es indicado por el programador explícitamente, supondremos que es 0, es decir, el **offset es un parámetro opcional.**

Con ello obtenemos un prototipo muy similar al de CUDA (**Código 8, Pág. 38**), salvando las pequeñas diferencias en el kernel/función. En OpenCL también se soporta la cláusula calls en el caso de implementaciones (implements), al igual que en CUDA.

```
#pragma omp target device(opencl) ndrange(1,0,size,MAX_NUM_THREADS) copy_deps
#pragma omp task output([size] output) \
                input([number_of_particles] d_particles)
__kernel void calculate_forces_kernel_naive(float time_interval,  int number_of_particles,
                                            __global Particle* d_particles,__global Particle *output,
                                            int first_local, int last_local) {
    size_t id = get_global_id(0)+first_local;
    if (id > last_local ) return;

    __global Particle* this_particle = output + id - first_local;
```

Código 11: Prototipo final diseño OpenCL

El prototipo mostrado en el **Código 11** debe ser compilado correctamente por Mercurium generando las llamadas necesarias para que el Runtime sea capaz de ejecutarlo.

4.3 IMPLEMENTACIÓN[5]

RUNTIME

Comenzaremos con la implementación del Runtime, ya que necesitamos tener un runtime funcional para saber que código hemos de generar al compilar, aunque realmente no podremos probarlo hasta que esté completo el compilador, lo cual dificulta el desarrollo.

Hemos de estudiar la estructura del runtime para poder modificarlo, las cuatro partes principales que componen una arquitectura (OpenCL) son las siguientes clases:

1- **Ocldd.cpp/hpp**: Se describen todos los *device descriptor* de OpenCL (actualmente tiene varios, para leer y escribir en la memoria de la GPU, ejecutar un kernel, obtener información de los *devices*…, es decir, tiene un *device descriptor* diferente para mantener una relación directa con la API OpenCL). Es decir, una estructura que contiene la información independiente de la *task* que necesita una función de este *device* para ejecutarse. Como se ha comentado anteriormente, OpenCL necesita una información diferente a los demás *devices*, por lo que hemos de crear un nuevo *device descriptor* en el cual se le puedan pasar los parámetros que necesita, en lugar de realizarlo en los principales de la *task*.

2- **Oclthread.cpp/hpp:** En ella tenemos la función que Nanox llama cuando recibe un *device* correspondiente a la arquitectura. Se encarga de descubrir de que tipo es el *Device descriptor* que recibimos (de los descritos en **Ocldd.cpp/hpp**) y realizar las acciones que desencadena ese descriptor.

3- **Oclcache.cpp/hpp:** Contiene funciones que serán llamadas por la cache de Nanox indicándonos qué datos hemos de escribir/reservar/leer. En ella hemos de realizar las transferencias de datos entre la memoria principal y las GPU.

4- **Oclprocessor.cpp/hpp**: Contiene funciones que serán llamadas desde **Oclthread (2)** y que realizan las operaciones solicitadas utilizando la API de OpenCL.

[5] En los títulos correspondientes se encuentran referenciados los adjuntos implementados.

Para comenzar la implementación hemos de **añadir un nuevo *device descriptor* para OpenCL**, que sea capaz de lanzar un kernel OpenCL, pero además realice todas las operaciones necesarias que han de hacerse para lanzarlo (copia de datos a la GPU, ejecución, copia de datos a la CPU…). El descriptor será bastante similar al ya existente para lanzar kernels, pero tiene dos modificaciones:

-**Añadir parámetros de la función**: Debido a que no podemos reutilizar los parámetros que se le pasan a la *task* por defecto y necesitamos unos propios para el *device* OpenCL, hemos de incluirlos en el *device descriptor*, ya que es la única estructura de la *task* que es diferente para cada *device*. Para ello, en el momento que se pasan estos parámetros al declarar la tarea hemos de copiarlos (similar al comportamiento de **first_private** en OMP) ya que nadie nos garantiza que en el momento que se ejecuta la *task* realmente, aún estén disponibles esos datos y el mismo valor que en el momento donde se inicializa el *device descriptor* (cuando se declara la *task*).

-Habilitar un modo especial de funcionamiento para el back-end: Cuando detectamos que alguien declara una *task* OpenCL usando nuestro *device descriptor*, asumimos que está en modo OmpSS. Entonces habilitamos este modo para saber que hemos de realizar acciones diferentes a la implementación existente cuándo se realicen solicitudes al back-end (principalmente caches).

Una vez que tenemos el front-end OpenCL adaptado para ejecutar kernels OmpSS, hemos de modificar el back-end.

El principal problema del back-end actual con nuestra implementación es que no realiza copias de los datos que se necesitan para ejecutar el kernel (han de ser copiados a las GPU antes de poder trabajar desde ellas), ya que está preparado para funcionar con unos datos que Mercurium no maneja. Por lo cual hemos de controlar qué datos de la cache necesitan ser copiados y cuando utilizarlos.

Para ello hemos de guardar en estructuras la relación tamaño-dirección y objeto OpenCL que representa la copia en la GPU. De esta forma podremos decidir qué argumentos/buffers de GPU han de utilizarse en los kernels cuando son ejecutados.

Revisando la implementación actual comprobamos que se identifican los buffers mediante un identificador numérico (1,2,3,4…) que está asociado a un buffer OpenCL. Nosotros no podemos utilizar este sistema porque en OmpSS el programador no es consciente de los buffers OpenCL y no los crea explícitamente. Por lo cual hemos de identificar estos buffer con la información que tenemos (dirección de memoria del dato o tamaño).

CACHE OPENCL (OCLCACHE.CPP)

A continuación explicaremos los cambios en la clase **oclcache.cpp** que es la clase que se encarga de satisfacer las peticiones de copia del sistema de cache de Nanox, y en la que hemos de copiar los datos solicitados a la cache.

Antes de comenzar se ha de mencionar que en todas estas funciones compartiremos dos listas, las cuales han sido creadas para utilizarse cuando ejecutemos *task* OmPSS:

a) **Lista de tamaños:** Mapa en el que se guardarán parejas con Clave un entero indicando el tamaño del buffer, y el objeto representativo del buffer OpenCL (**cl_mem**). Por lo general esta lista representa buffers alocatados pero que aún no se han usado.

b) **Lista de direcciones:** Mapa en el que se guardarán parejas con Clave un número indicando la dirección del dato en el host y el objeto representativo del buffer OpenCL (**cl_mem**). Por lo general esta lista representa buffers que ya se han usado y están asociados a una dirección de host.

Funciones que se han modificado:

1- **Allocate (tamaño):** En esta función hemos de solicitar al *device* OpenCL que reserve un buffer de tamaño determinado y hemos de guardar el identificador al buffer en cuestión. Este buffer lo guardamos en la **lista de tamaños.**
2- **CopyIn (dirección host, tamaño):** En esta función se copia al *device* OpenCL los datos del host, para ello hemos de buscar en la **lista de tamaños** un buffer del mismo tamaño para el que se solicita la copia. Una vez lo encontramos, escribimos en ese buffer (writeBuffer), **eliminamos el buffer de la lista de tamaños y lo añadimos a la lista de direcciones.**
3- **CopyOut (dirección host, tamaño):** En esta función se copia al host los datos del *device* OpenCL. Para ello hemos de buscar en la **lista de direcciones** el dato solicitado y copiarlo a la memoria principal (readBuffer).
4- **Free (dirección host):** En esta función se busca el buffer en la **lista de direcciones** y se libera el buffer en el *device* OpenCL (freeBuffer). A continuación se borra el buffer de la **lista de direcciones.**
5- **Reallocate (dirección host, tamaño):** En esta función, por ahora, se invoca a las funciones **free (dirección host)** y **allocate (tamaño),** para reutilizar un buffer.

CAMBIOS OCLPROCESSOR.CPP

A pesar de que la mayor parte del código existente podemos reutilizarlo (compilar kernels, eventos…), hemos de modificar como se manejan los parámetros a la hora de ejecutar un kernel para que sea capaz de utilizar la implementación anterior de la cache, una vez realizados estos cambios, seremos capaces de ejecutar un kernel OpenCL correctamente.

Como se ha explicado anteriormente, en esta fase recibiremos de la llamanda a Nanox (que realiza el compilador), **el código fuente en forma de string, el nombre del kernel a ejecutar y los argumentos (dirección y tamaño)** que se pasarán al kernel.

Esta llamada podemos ejecutarla siguiendo los pasos necesarios para llamar a cualquier kernel OpenCL. A continuación compilamos el código fuente (clBuildProgram), indicamos a OpenCL los argumentos a utilizar (clSetKernelArg) y lanzamos el kernel (clEnruqueNdRangeKernel).

Sin embargo tenemos un único problema, y es que en el caso de que los argumentos del kernel sean buffers (punteros), en lugar de pasar el argumento directamente, hemos de pasar el **buffer** que proviene **de la cache** que implementamos previamente.

Para buscar estos buffer hemos de realizar varias acciones. En primer lugar del conjunto de argumentos que recibimos del compilador, **los inmediatos** (cualquier dato que no sea un puntero) **se pueden pasar directamente al kernel**, por lo cual no hemos de tratarlos.

En segundo lugar **los argumentos de tipo puntero** (buffers) **se tratan de una manera especial**, ya que el tamaño que proporciona el compilador es el del propio puntero (función *sizeof* de C), sin embargo, cualquier **buffer que en una *task* OmpSS se vaya a utilizar en un *device* que no sea SMP, ha de ser incluido en las clausulas copy_in o copy_out** en las que el programador indica el tamaño real del mismo.

Por lo tanto, utilizando la dirección del argumento y la información que se nos pasa en los copy_in o copy_out de la *task* Nanox, hemos de corregir el tamaño del argumento para que indique el tamaño real de los datos a los que apunta el puntero.

Ahora que ya conocemos el tamaño real de todos los punteros, hemos de buscar en la cache el objeto GPU que los representa, para ello seguiremos los siguientes criterios:

1- **Si existe un buffer asociado a la dirección del argumento, utilizaremos ese buffer,** este buffer ya está asociado a esa dirección, lo cual puede ocurrir por dos razones, o que se ha copiado directamente de la memoria compartida del host, o que se ha escrito previamente en otra *task* OpenCL.
2- **Si no existe un buffer asociado a la dirección del argumento**, significa que es un argumento de salida (del que Nanox ha solicitado reservar el tamaño, pero no se ha escrito ningún valor). **En este caso buscamos un buffer libre con el tamaño del argumento y lo asociamos a la dirección del mismo**.

Una vez tenemos automatizada tanto la cache como la ejecución de los kernel, el runtime de OpenCL ya es capaz de ejecutar *task*s OmpSS generadas mediante Mercurium.

COMPILADOR

Como ya hemos explicado anteriormente todos los *device* de Mercurium comparten la misma API. Además OpenCL y CUDA comparten la mayor parte de los problemas, por lo cual la implementación será bastante similar a la de CUDA.

FRONT-END

Dado que en OpenCL no tenemos ninguna base, a diferencia de CUDA, donde el front-end ya estaba completo , hemos de modificar el compilador front-end (que utiliza bison) para que reconozca los token de OpenCL, es decir, las palabras claves de OpenCL (_kernel, _global…). Sin embargo no es necesario que modifiquemos el front-end para que reconozca estructuras de código especiales, ya que un kernel OpenCL es similar a cualquier código C, exceptuando que ha de reconocer las palabras claves de OpenCL como tokens.

Para ello hemos de realizar **cambios en varios ficheros del front-end**.

Comenzamos declarando los tokens que necesitamos para OpenCL (**AST_OPENCL_GLOBAL, AST_OPENCL_KERNEL, AST_OPENCL_CONSTANT, AST_OPENCL_LOCAL**), añadiéndolos en el fichero **cxx-asstype-opencl.def**, que se importa en el asttype-all.def.

Estos tokens serán definidos en el fichero **cxx-opencl.y**, como ya hemos dicho son tokens simples, hojas del árbol sintáctico, es decir, que no pueden tener hijos por debajo suyo.

Hay que tener en cuenta que al ser un compilador código a código, además de dar la posibilidad de procesar código en tokens del compilador, hemos de ser capaces de volver a traducir estos tokens en código. En la clase **cxx-prettyprint.c** hemos de implementar esta transformación, indicando al compilador que un token de nombre por ejemplo AST_OPENCL_GLOBAL, se imprime como el texto “_global”.

Ahora ya somos capaces de preprocesar los tokens. OpenCL tiene otro tipo de operaciones propias del lenguaje. Por ejemplo se pueden utilizar determinadas funciones reservadas por OpenCL (P.E.: get_global_id()) que no existen para el preprocesador, ya que este no tiene constancia de que el código es OpenCL antes de preprocesarlo. Para solventar este problema añadiremos unas declaraciones “artificiales” de estas funciones, para que de esta forma Mercurium es capaz de preprocesar las llamadas. Con realizar las declaraciones es suficiente, ya que estas llamadas se eliminarán del código C y serán compiladas por el compilador de OpenCL.

Por ejemplo, para que Mercurium sea capaz de preprocesar una llamada OpenCL a la función get_global_id(0), que devuelve el identificador numérico del kernel. Realizamos la declaración “unsigned get_local_id(unsigned int arg0);”.

BACK-END

Al igual que en CUDA, hemos de implementar del *device* provider:

1- **Insert_function_definition/insert_declaration:** Funciones que se llaman cuando Mercurium lee cualquier tipo de #pragma omp target *device* (OpenCL). Como parámetros se pasa un objeto que representa el pragma y a partir del cual se puede obtener información sobre la directiva, además de sobre la función o declaración a la que se refiere.
2- **Create_outline**: Esta es la función que en el caso de CUDA se encargaba de generar una función que ejecutaba el Runtime. Ya que en el runtime de OpenCL esto no es necesario únicamente se ha de hacer una funcionalidad similar a la implementada insert_function_definition, para que en el caso de recibir una *task,* que también es una función, hemos de llevar su código al fichero OpenCL.
3- **Get_*Device*_Descriptor**: En esta función se genera el descriptor del *device*. En el caso de OpenCL es la función en la que se realiza un trabajo bastante similar al que realizaba la función create_outline en CUDA, pero en lugar de crear otra función, ha de crear y rellenar la estructura con los datos necesarios para que el runtime sea capaz de ejecutar el Kernel OpenCL.

La mayor diferencia con el *device* CUDA es que en CUDA **Mercurium genera un fichero .cu** en el que va copiando el código que necesita, para a continuación **compilarlo con NVCC** (compilador).

En OpenCL no existe un compilador independiente, sino que se compila el código para cada dispositivo en concreto, este código se pasa como una variable de tipo **char*.** Por lo tanto, en lugar de almacenar el código en un fichero como se hace en el *device* CUDA, **se almacenará el código en una variable String en memoria.** Este código se compilará en tiempo de ejecución, ya que es así como normalmente se hace en OpenCL.

IMPLEMENTACIÓN FUNCIÓN INSERT_FUNCTION_DEFINITION/INSERT_DECLARATION (1)

En esta función hemos de realizar dos acciones, al igual que en CUDA. En caso de que sea una declaración/definición de una función hay que generar el **código intermedio que necesita el *device descriptor*** llamando a la función **generate_wrapper_opencl**. Una vez terminado lo anterior, almacenamos el código que se ha definido/declarado dentro del código **OpenCL,** además de eliminarlo del fichero .c/.cpp original.

IMPLEMENTACIÓN DE LA FUNCIÓN GENERATE_WRAPPER_OPENCL

Como ya se ha explicado anteriormente, esta función se encarga de generar el **esqueleto** del código que se encarga de preparar los datos que necesita el runtime para ejecutar un kernel OpenCL.

Se le pasan únicamente 4 argumentos. La declaración de la *task* (que a fin de cuentas, es una función de C), la declaración del kernel OpenCL, y el contenido de las cláusulas ndrange y calls. Con el objetivo de generar un código similar al siguiente:

```
/* OpenCL device descriptor (and arguments preparation) */
/* This code is generated based on ndrange and calls clauses from: Impleme
nanos_ocl_ndrange_kernel_starss_args_t _opencl__ol_init33_30_0_ocl_args;
_opencl__ol_init33_30_0_ocl_args.work_dim = 1;
size_t offset_arr__opencl__ol_init33_30_0[1];
size_t local_size_arr__opencl__ol_init33_30_0[1];
size_t global_size_arr__opencl__ol_init33_30_0[1];
offset_arr__opencl__ol_init33_30_0[0] = (0);
local_size_arr__opencl__ol_init33_30_0[0] = ((__tmp_2) < (64) ? (__tmp_2)
global_size_arr__opencl__ol_init33_30_0[0] = ((__tmp_2) < (64) ? (64) : (_
_opencl__ol_init33_30_0_ocl_args.global_work_offset = offset_arr__opencl__
_opencl__ol_init33_30_0_ocl_args.local_work_size = local_size_arr__opencl_
_opencl__ol_init33_30_0_ocl_args.global_work_size = global_size_arr__openc
ocl_struct__opencl__ol_init33_30_0 ol_args__opencl__ol_init33_30_0;
/* This string has every .cl included in ImplementacinOCL.c */
/* and target device opencl functions/kernels */
ol_args__opencl__ol_init33_30_0.ocl_code = " __kernel void init22(__global
ol_args__opencl__ol_init33_30_0.ocl_kernelName = "init33";
ol_args__opencl__ol_init33_30_0.compiler_params = "";
ol_args__opencl__ol_init33_30_0.args_num = 3;
ol_args__opencl__ol_init33_30_0.__size_of_a = sizeof (__tmp_6);
ol_args__opencl__ol_init33_30_0.a = __tmp_6;
ol_args__opencl__ol_init33_30_0.__size_of_a |= 1;
ol_args__opencl__ol_init33_30_0.__size_of_b = sizeof (__tmp_7);
ol_args__opencl__ol_init33_30_0.b = __tmp_7;
ol_args__opencl__ol_init33_30_0.__size_of_b |= 1;
ol_args__opencl__ol_init33_30_0.__size_of_off_xxx = sizeof (__tmp_0);
ol_args__opencl__ol_init33_30_0.off_xxx = &__tmp_0;
_opencl__ol_init33_30_0_ocl_args.data = &ol_args__opencl__ol_init33_30_0;
nanos_device_t _ol_main_30_devices[] = {{
    nanos_ocl_ndrange_kernel_starss_factory,
    nanos_ocl_ndrange_kernel_starss_dd_size,
    &_opencl__ol_init33_30_0_ocl_args
}};
```

Código 12: Código necesario para que Nanox sea capaz de lanzar un kernel OpenCL

Aunque en este caso el código es mucho menos claro que en CUDA, realmente necesitamos la misma información**.** Es necesario conocer las dimensiones (x, y, z) con las que se lanzará el kernel, las cuales conocemos gracias a la cláusula ndrange (que proporcionará el

programador), el nombre del kernel y los parámetros con los cuales se realizará la llamada, los cuales se pueden conocer utilizando las dos declaraciones que se le pasan a la función.

Comenzamos **generando el código ndrange** (dimensiones) que es lo más simple, recibiremos una cláusula con los siguientes parámetros (num_dimensiones, offset[6], global_size[7], local_size[8]). Además hay que aplicar algunas funciones de redondeo que son necesarias a la hora de lanzar kernels, para corregir tamaños que no sean divisibles.

Hay que tener en cuenta que normalmente en lugar de números directamente, se utilizarán parámetros de la propia función que tendremos que renombrar de acuerdo al renombrado que hace Mercurium con las *task* o constantes de las que tendremos que buscar su declaración y llevarlas al fichero OpenCL.

Una vez generado el código del ndrange, hemos de **generar el código que se encarga de almacenar los punteros de los argumentos del kernel, además de su tamaño, en la estructura que le pasamos al runtime.** Para ello hemos de obtener el nombre del kernel (el cual se puede obtener sin problemas de la declaración) y realizar un cruce entre las declaraciones del kernel y la *task*/función SMP para saber que parámetros han de pasarse al kernel, el criterio a seguir será el siguiente:

1- Si está definida la cláusula calls, se utilizarán los valores que el programador indica en ella. Estos valores hay que procesarlos y renombrarlos ya que cuando nos llegan por parte de Mercurium, no se llaman igual que lo que el programador indica en la directiva. La problemática es la misma que con el ndrange.
2- Si no está definida la cláusula calls, se comprobará que el argumento del kernel y el de la *task* se llaman igual y tienen el mismo tipo. Si esto se cumple, se utilizará el argumento de la *task* como argumento del kernel.

Una vez conocemos los argumentos que se han de procesar, hemos de **generar el código que cree una estructura en la cual almacenamos los argumentos,** es decir, el código OpenCL, el nombre del kernel a ejecutar, el número de argumentos, los punteros de los argumentos y el tamaño de los mismos. **Esta estructura ha de ser contigua en memoria** (atributo "packed") para que el runtime pueda recorrerla conociendo únicamente el puntero al primer dato de la misma.

A su vez, al igual que generamos la declaración de la propia estructura, generamos el código necesario para rellenarla, aunque con ciertas incógnitas que serán resueltas en la función **get_device_descriptor (3)**.

[6] - Vectores de tamaño num_dimensiones.

[7] - Vectores de tamaño num_dimensiones.

[8] - Vectores de tamaño num_dimensiones.

Tras esto se guardará el código generado en nuestra **lista de códigos de invocación Clave > Valor** que será utilizada más tarde, cuando las *task* de Mercurium pidan que se genere la función para llamar al kernel.

IMPLEMENTACIÓN DE LA FUNCIÓN CREATE_OUTLINE (2)

Dado que en OpenCL no se puede pasar tanto por nuestro diseño como por limitaciones de la API de OpenCL ninguna función para que el runtime la ejecute, no es necesario generar ningún código en esta función.

Por lo tanto esta función realizará lo que en la parte de CUDA se añadió en la fase 2. Es decir, dado que cuando un kernel se declara como task directamente, únicamente se llama a esta función en lugar de a la función **insert_function_definition/declaration(1)**, hemos de buscar la declaración del kernel en el código del usuario y realizar las mismas acciones que se han explicado en esa función, es decir, **pasar ese código a OpenCL y generar el esqueleto del código que lanzará el kernel**.

IMPLEMENTACIÓN DE LA FUNCIÓN GET_DEVICE_DESCRIPTOR (3)

En esta función se generará el *device descriptor* que se le pasa al runtime, en OpenCL esto no es trivial ya que en lugar de pasar el puntero a la función generada en **create_outline (2)** el *device descriptor* contiene toda la información necesaria para lanzar el kernel, para ello **recogeremos el código que generamos previamente en la función generate_wrapper_opencl de nuestra lista clave->valor.**

En este código hemos de realizar varios reemplazos, ya que el nombre de las variables y estructuras. Ha de contener el identificador único que le da Mercurium a cada task para evitar pisarlos con otras instancias de la misma task que se puedan invocar en otra parte del código.

Una vez realizado ese paso, hemos de parsear la declaración de la estructura en la cual guardaremos los argumentos y añadirla al código del usuario.

Ahora que ya **disponemos del código completo y tenemos la estructura que almacenará los argumentos declarada,** únicamente hemos de parsear el código y devolverlo como el descriptor que Mercurium pasará al runtime.

En definitiva, tras finalizar la implementación de estas funciones **conseguimos un Device Provider que es capaz de almacenar el código OpenCL del usuario, eliminarlo de los ficheros .c/.cpp(para que estos compilen correctamente con gcc), y de proveer a Mercurium un *device descriptor*** que contiene toda la información necesaria para el runtime (Nanox) sea capaz de ejecutar una tarea OpenCL utilizando todas las mejoras que aporta OmpSS (cache, dependencias entre tareas…).

CAPÍTULO 5. EVALUACIÓN Y PRUEBAS

En el este capítulo explicaremos todas las pruebas que se han realizado en el proyecto. Principalmente son pruebas de funcionalidad, no de rendimiento, aunque se ha realizado alguna prueba de rendimiento.

Todas las pruebas se pueden encontrar en la **carpeta adjunta Pruebas** en el directorio correspondiente.

Los títulos de cada prueba referencian a la correspondiente carpeta, es decir, si se pulsa en el título se abrirá la carpeta correspondiente.

5.1 PRUEBAS CUDA

Como se ha visto en el desarrollo CUDA, se ha modificado el compilador para que haga automáticamente parte del trabajo del que anteriormente se encargaba el usuario, por lo que nuestras primeras pruebas consistirán en demostrar que el comportamiento CUDA es similar al funcionamiento previo de OmpSS. Además se va a demostrar que se pueden utilizar varias implementaciones (SMP y CUDA) de una sola función.

Plataforma de pruebas:

-**Sistema Operativo:** Debian GNU/Linux 5.0

-**GPU:** 2x nVidia Geforce 285 GTX

-**CPU:** 2x Intel Xeon X5472 3.0 GHz

-**RAM:** 4GB.

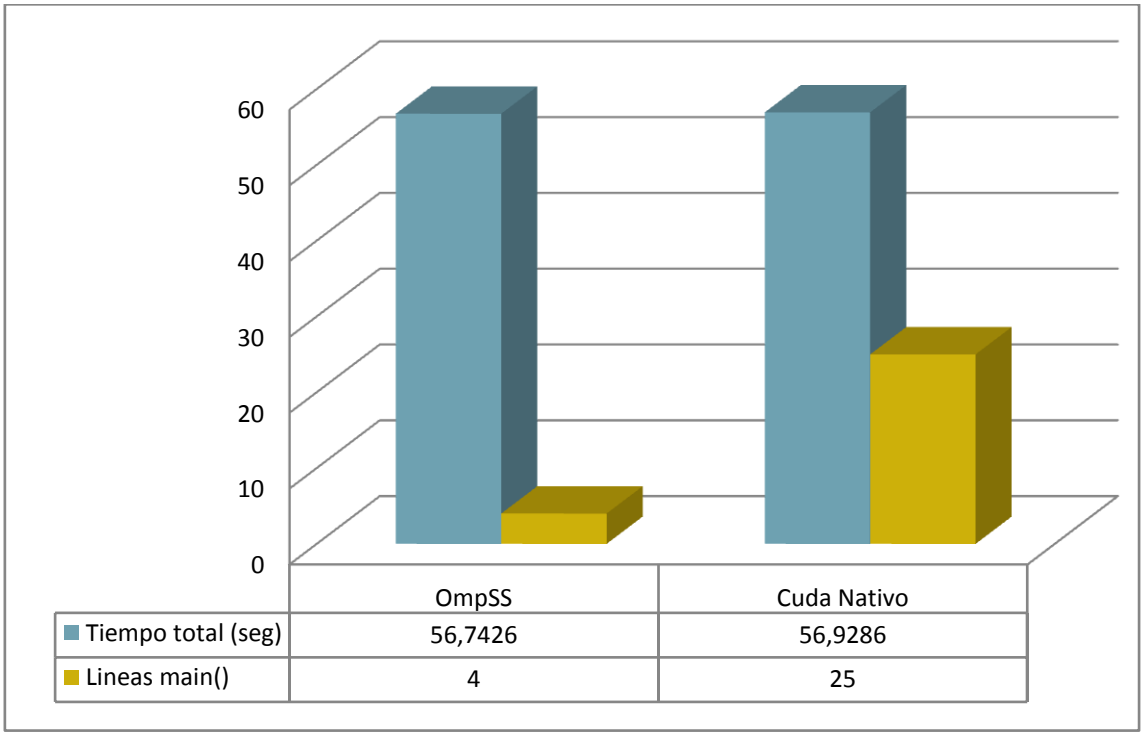
PRUEBA 1 (MATMUL)

La primera prueba será simple, se comprobará el rendimiento de las pruebas, tanto en el tiempo total que tarda en ejecutarse el proceso como de líneas de código necesarias para lanzar los kernel (***./Pruebas/CUDA/MatMul***), matmul, es decir, **multiplicación de matrices,** la comparación se realizará **entre CUDA nativo y la versión nueva de OmPSS.**

Multiplicaremos cuatro matrices dos a dos (a*b=c y e*d=f), cada matriz de tamaño 8192* 8192, todos sus valores serán 1 para facilitar la comprobación del resultado, en una de las ejecuciones lo haremos utilizando CUDA nativo y en la otra OmpSS.

Los resultados son los siguientes:

Ilustración 1: Gráfica pruebas CUDA multiplicación de matrices.



El primer resultado, el más importante para nuestra parte del proyecto, es la sencillez del código, aún teniendo en cuenta que es un programa muy simple de ejecutar en CUDA nativo, y muy simple para nosotros.

- En CUDA nativo se necesitan alrededor de 25 líneas de código en las que el programador ha de controlar los accesos y la copia de datos entre el host y el *device.*
- En OmpSS se necesitan 4 líneas de código, dos para especificar el kernel CUDA como tarea y sus input/output. Y otras dos para llamar a la función.

Los tiempos reales de ejecución son totalmente similares (Nativo: 56,9286 y OmpSS: 56,7426) con lo cual comprobamos que el *overhead* provocado por Nanox es totalmente despreciable (incluso el tiempo obtenido ha sido mejor). Si se comprueban los resultados en los ficheros adjuntos, vemos que el tiempo de usuario en OmpSS es el doble que en nativo. La causa de esto es que Nanox crea dos hilos para la ejecución, aunque uno únicamente se dedica a lanzar kernels y esperar, con un gasto inapreciable de recursos. En CUDA nativo ya que la ejecución es no bloqueante, no creamos dos threads, sino que esperamos con uno a ambos kernels.

PRUEBA 2 (NBODY OMPSS ANTERIOR CONTRA NBODY OMPSS NUEVA)

Nuestra segunda prueba **(*./Pruebas/CUDA/Nbody*)** para CUDA será un proceso mucho más complejo que el anterior, siendo uno de los **test que utilizan en el BSC para probar Mercurium**. Del que conseguir su ejecución es lo que confirmaría que se ha alcanzado el objetivo del proyecto. El problema **nbody**[9], en el cual se puede aprovechar muy bien el potencial de las GPU y OmpSS, en este problema se intenta predecir los movimientos que un grupo de partículas sufren por culpa de la interacción entre ellas en un momento de tiempo determinado. La prueba será tanto funcional como de rendimiento.

En este caso la prueba no será frente a CUDA nativo, sino que **compararemos la versión de OmpSS CUDA que ya existía antes de la realización del proyecto y la nuestra**, ambas usan Nanox, por lo que el objetivo no es que una sea más rápida que la otra, sino demostrar que tanto la nueva como la anterior tienen el mismo comportamiento. Para comprobarlo utilizaremos la aplicación **Paraver**[10].

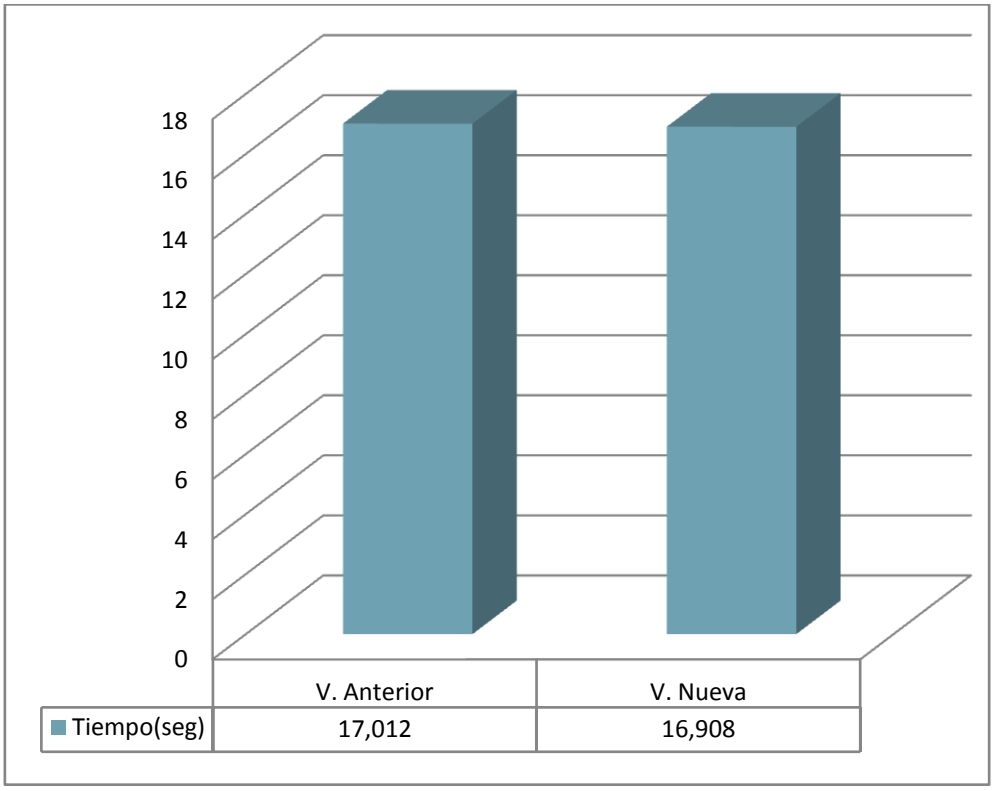
Además comprobaremos que se pueden **utilizar varias implementaciones de una función** para distintos *devices* (SMP y CUDA), aunque esto no tendrá un rendimiento ideal ya que es una rama de Nanox en desarrollo que por ahora tiene un *scheduler* muy simple.

[9] http://en.wikipedia.org/wiki/N-body_problem

[10] http://www.bsc.es/computer-sciences/performance-tools/paraver/

Comenzaremos con la prueba de rendimiento:

Ilustración 2: Gráfica pruebas CUDA multiplicación de matrices.



Como vemos ambos casos dan un **rendimiento similar**, siendo ligeramente superior la versión nueva (16,908s contra 17,012s) probablemente por cuestiones externas a Nanox (aplicaciones en background…), ya que realmente debería ejecutar el mismo código.

En la siguiente figura analizaremos el **comportamiento de ambas versiones**, que como ya hemos comentado anteriormente **ha de ser similar**, con la diferencia de que en la versión nueva el código lo ha creado Mercurium a partir del kernel definido como *task* en lugar de tener que utilizar el programador una función puente que lance el kernel.

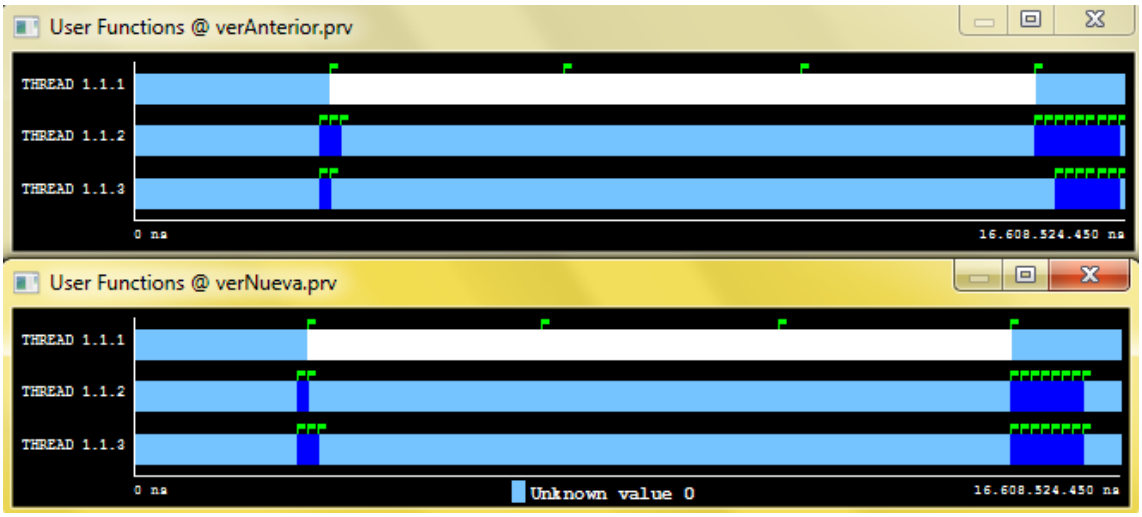


Figura 8: Funciones lanzadas en las pruebas (versión anterior arriba, versión nueva abajo).

Para entender el gráfico anterior hemos de explicar que existen tres threads:

1.1.1 – Thread SMP (CPU).
1.1.2 – Thread GPU0 (GPU).
1.1.3 –Thread GPU1 (GPU).

Como se puede ver en el fichero “kernel.c” de las pruebas, nuestra función de cálculo tiene dos implementaciones, un kernel CUDA (GPU) y una función SMP. También debemos explicar la versión de prueba de Nanox que hemos utilizado está aún en desarrollo y tiene un *scheduler* muy simple, que ejecuta 3 veces cada implementación y elige la más rápida.

Con estos datos podemos interpretar las llamadas a la función nbody. Como vemos gracias a las “banderas” que representan cada llamada, se realizan unas 20 llamadas a la función que calcula el nbody, 10 *timesteps* que se calculan en dos partes cada una, para poder lanzarlo en las dos GPU de las que disponemos.

Vemos que en ambos casos comienzan eligiendo la mejor implementación, para ello lanzan tres *task* de GPU, y tres *task* SMP (color blanco), que como se puede ver son mucho más lentas. Por lo tanto conociendo estos resultados, en un proceso real lo aconsejable sería eliminarlas, aunque entonces el proceso no funcionaría si no disponemos de una GPU CUDA.

Una vez se han lanzado tres *task* de cada tipo el *scheduler* decide que es mucho más óptima la implementación GPU y a continuación lanza las 14 *task* restantes en ambos casos.

Con esta prueba vemos que **ambas versiones realizan una ejecución similar**, siendo la única diferencia la provocada por el *scheduler* que elige distintas GPU para ejecutar las *task*. Con lo cual podemos dar la prueba como válida.

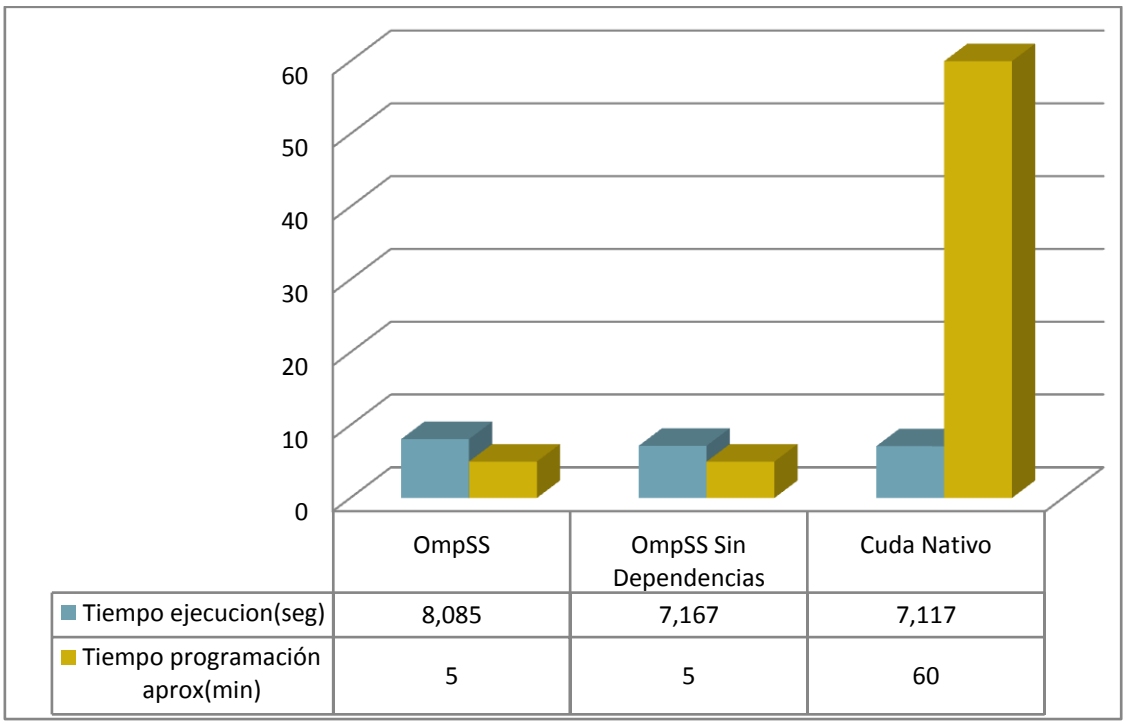
PRUEBA 3 (NBODY CUDA NATIVO CONTRA OMPSS)

Nuestra tercera prueba será similar a la anterior, utilizaremos como referencia la versión nueva, pero eliminaremos la implementación SMP, en esta prueba compararemos N-Body OmpSS con N-Body CUDA Nativo. El objetivo principal de la prueba es demostrar la dificultad de desarrollar en CUDA Nativo un proceso complejo en comparación con el desarrollo OmpSS.

La primera diferencia lo vemos en el desarrollo, ya que partiendo de la misma base, es decir, teniendo el kernel programado y únicamente necesitando copiar la memoria entre dos GPUs y lanzar el kernel, el tiempo de desarrollo del proceso en **CUDA Nativo** fue de aproximadamente **60 minutos** y el tiempo de desarrollo del proceso en **OmpSS** fue de **5 minutos**.

Aunque el tiempo para programarlo en CUDA nativo podría mejorarse más que el de OmpSS adquiriendo mayor soltura en CUDA, hay que tener en cuenta que no se implementaron las dependencias entre kernels en CUDA nativo, ya que aumentaría mucho la complejidad del código. Por tanto el resultado del proceso no es perfecto, pero esto no es relevante para estas pruebas ya que no necesitamos comprobar funcionalidad. Por ello vamos a eliminar las dependencias de Mercurium, tendremos el mismo error que en CUDA nativo, pero podremos hacer una comparativa más justa. Siendo este el resultado de las mismas:

Ilustración 3: Gráfica pruebas Nbody CUDA nativo contra OmpSS



	OmpSS	OmpSS Sin Dependencias	Cuda Nativo
Tiempo ejecucion(seg)	8,085	7,167	7,117
Tiempo programación aprox(min)	5	5	60

Como vemos en los resultados, **la versión de CUDA Nativo es casi igual de rápida que la de OmpSS sin dependencias**, como ya se ha comentado anteriormente ambas son un poco diferentes a la implementación con dependencias, que es un poco más lenta por las esperas que provocan las dependencias entre tareas.

5.2 PRUEBAS OPENCL

Como se ha visto en el desarrollo OpenCL, se ha modificado OmpSS para que sea capaz de ejecutar kernels OpenCL de una forma muy similar a como ejecuta los kernel en CUDA, una vez realizadas las modificaciones hemos de probar su correcto funcionamiento, en este caso la cantidad de las pruebas será menor, ya que al no tener una versión anterior de OpenCL, comprobaremos el funcionamiento correcto del test **nbody** y realizaremos una comparación con **OpenCL Nativo** multiplicando matrices.

Debido a problemas con elementos del sistema, posiblemente drivers, de la anterior plataforma de pruebas a la hora de ejecutar **nbody** OpenCL, para el desarrollo OpenCL se ha utilizado una nueva plataforma de pruebas:

-Sistema Operativo: Ubuntu 12.0

-**GPU:** 1x AMD Radeon 6870

-**CPU:** 1x AMD Phenom II X4 955 3.2 GHz

-**RAM:** 4GB DDR2 800Mhz CL5

PRUEBA 4 (OPENCL NBODY)

Esta prueba que realizaremos será diferente a las demás, ya que no se va a comparar con un nbody en OpenCL nativo, y no podemos comparar con la ejecución en CUDA, debido a que la GPU de la que disponemos no es capaz de realizar cálculos en doble precisión, por lo cual no podemos utilizarla para ejecutar el test nbody. **Para esta prueba utilizaremos la CPU como device OpenCL**, aunque su rendimiento sea mucho más lento que la GPU.

Como ya se ha comentado anteriormente, nbody utiliza muchas dependencias entre datos y tareas, por lo cual **si somos capaces de ejecutar la prueba correctamente y validar los resultados, podremos considerar que el desarrollo OpenCL funciona correctamente**.

Si se analiza el código de la prueba **nbody CUDA** y **nbody OpenCL** (actual) podemos ver como para ejecutar en OpenCL. En la definición de la *task* únicamente se ha de cambiar el *device* CUDA por OpenCL, aunque hay que traducir el kernel de CUDA y OpenCL. Dado que ambos son muy similares, es una traducción directa de las funciones que ofrece cada lenguaje y que realizan la misma labor.

El tiempo de ejecución es de **20 segundos** (muy superior a los 7 segundos que tarda en ejecutarse el mismo proceso en GPU).

La salida que obtenemos es la misma que se obtienen en la ejecución de CUDA y los mismos que tenemos como referencia. Con lo que consideramos que la implementación de OmpSS del runtime y del compilador es correcta.

PRUEBA 5 (OPENCL MATMUL)

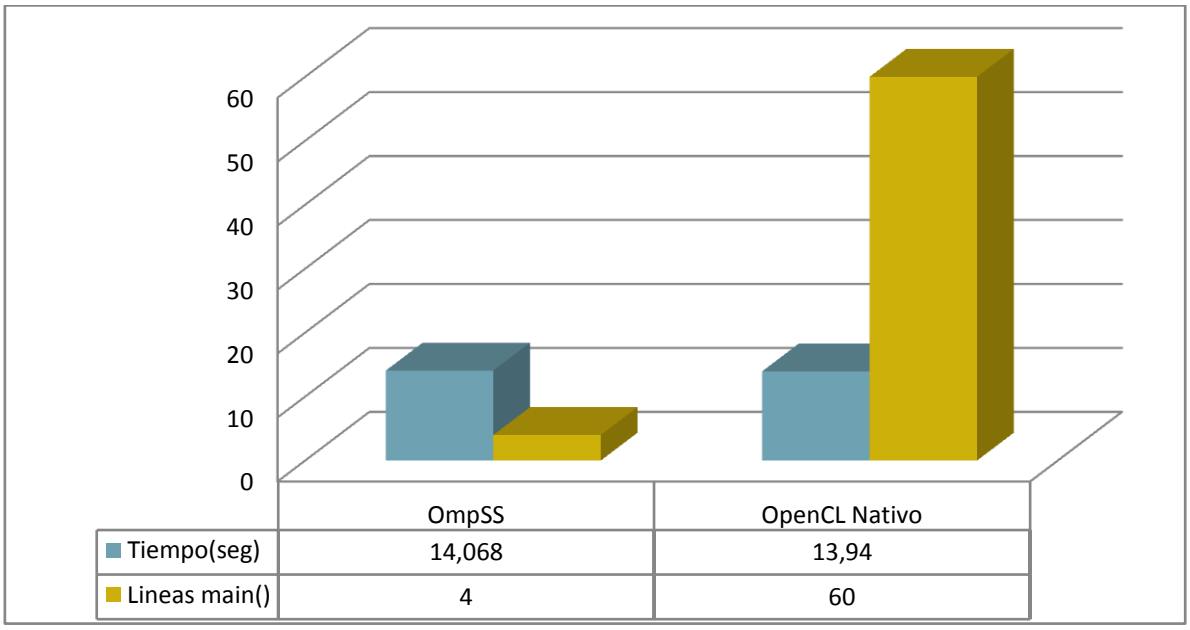
En esta última prueba compararemos la multiplicación de matrices implementado a través de OpenCL nativo, y OmpSS OpenCL. Además de medir el rendimiento, mediremos la cantidad de código (en líneas) que el programador ha de escribir para poder lanzar el kernel.

Al igual que en CUDA, multiplicaremos cuatro matrices dos a dos (a*b=c y e*d=f), cada matriz de tamaño 2048 * 2048, todos sus valores serán 1 para facilitar la comprobación del resultado, en una de las ejecuciones lo haremos utilizando OpenCL nativo y en la otra OpenCL OmpSS.

Si se comprueba el código nativo OpenCL vemos que exceptuando el kernel, es totalmente diferente al de CUDA, ya que en lugar de utilizar la API de CUDA, utiliza la de OpenCL, sin embargo, OpenCL OmpSS es similar al de CUDA OmpSS.

A continuación se describen los resultados de la prueba:

Ilustración 4: Gráfica resultados prueba MatMul OpenCL.



Como vemos el código en OpenCL es mucho más complejo que en OmpSS, y comparándolo con las pruebas anteriormente realizadas, también es más complejo que CUDA. Aunque realmente se realizan las mismas acciones, la API de OpenCL es bastante más compleja. La ventaja que ofrece es que bien utilizada también ofrece más posibilidades y compatibilidad con diferentes tipos de dispositivos y fabricantes.

Con esta prueba damos por finalizado el desarrollo y pruebas del proyecto, comprobando que OmpSS es capaz de ejecutar código CUDA y OpenCL utilizando esos dispositivos como un dispositivo más de forma transparente al usuario, y con un *overhead* despreciable en las pruebas realizadas.

CAPÍTULO 6. CONCLUSIONES Y TRABAJOS FUTUROS

6.1 CONCLUSIONES PERSONALES

A nivel personal este proyecto me ha aportado una valiosa experiencia a la hora de trabajar de forma remota con un equipo grande y con personas con las que no había mantenido contacto previamente. A lo que hay que añadir la colaboración con personas de otros países con los que la comunicación ha sido un poco más difícil debido al idioma, principalmente por la parte que me corresponde, ya que es la primera vez que he utilizado el inglés en un ambiente profesional.

También me ha ayudado a incrementar mi experiencia realizando proyectos en grupo en los cuales en mayor o menor medida se depende de otras personas y en los que las decisiones han de ser consensuadas con el resto del grupo.

Además de participar en un proyecto grande, con una gran cantidad de líneas de código, en el que todo está interrelacionado y aunque está diseñado con un sistema de plugins para extenderlo, a la hora de realizar algunos cambios hay que conocer en mayor medida todo el sistema e intentar evaluar que partes del mismo pueden verse afectadas por los cambios.

Hay que añadir que por parte del **BSC** han quedado satisfechos con el trabajo realizado y a nivel personal es una motivación extra para emprender futuros proyectos de esta o mayor complejidad.

6.2 CONCLUSIONES TÉCNICAS

A continuación explicaremos los objetivos (1.2) conseguidos durante el proyecto. Tal y como se ha explicado al comienzo del mismo, el objetivo era extender el compilador OmpSS para conseguir un device que fuera capaz de ejecutar OpenCL. Como paso intermedio para aprender el funcionamiento del compilador se comenzó modificando el device ya existente CUDA, con el objetivo de abstraer al usuario un poco más de lo que ya se abstraía anteriormente.

Respecto a los objetivos alcanzados, **se han alcanzados más objetivos de los propuestos inicialmente**.

Respecto al **objetivo 1**, se ha conseguido una implementación funcional del runtime de Nanox-OpenCL para OmpSS, se ha realizado el plugin para el compilador Mercurium gracias al cual se puede compilar código OpenCL y lanzar la tarea de Nanox, además **se ha conseguido poder anotar los kernel OpenCL directamente como *tasks*.** Algo que no estaba definido en los objetivos.

Respecto al **objetivo 2**, no sólo se ha conseguido generar el código que lanza los ***kernel* CUDA** automáticamente, sino que además, al igual que OpenCL, se ha conseguido anotar los **kernel CUDA** como *tasks*.

La consecución de estos objetivos de la forma en que se ha conseguido, provoca que alcancemos una **relación lógica "perfecta" entre kernel y *task* que facilita mucho la programación**. Esta relación proporciona un código más limpio y en el cual las tareas CUDA/OpenCL no necesitan de tareas secundarias cuya única función sea la de lanzar una sub-tarea CUDA/OpenCL, permite que el usuario no necesite aprender ningún estilo de programación especial para utilizar estos Device.

Para el **objetivo 3**, basándonos en lo ya conseguido para la realización de los objetivos anteriores, únicamente se modificaron algunas partes del compilador para permitir mayor versatilidad a la hora de implementar una *task* con otras funciones o *tasks*. Por lo demás, la implementación ya existente de mercurium, había sido diseñada para soportar esta funcionalidad, aunque aún no estaba soportada realmente.

En cuanto al trabajo realizado, a pesar de que la extensión de CUDA se extendió más de lo esperado, facilitó mucho el desarrollo de OpenCL, ya que empezando en un dispositivo ya funcional, fue más fácil familiarizarse con el compilador.

En la parte de OpenCL, en el compilador el trabajo fue más sencillo del esperado inicialmente, debido a que habiendo modificado casi totalmente la parte de CUDA, no había novedades en el compilador. Sin embargo, la rama Nanox-OpenCL del runtime, que en principio iba a ser similar a la de CUDA y no iba a ser necesario modificar, como se ha explicado anteriormente no era compatible con OmpSS ya que ha sido desarrollada para otro fin diferente.

Por ello **ha sido necesario adaptar el runtime al modo de funcionamiento OmpSS**, lo cual no solo implicó realizar el desarrollo, sino aprender el funcionamiento del runtime, del cual únicamente conocíamos la API que se utiliza desde Mercurium. Lo cual no estaba entre los objetivos del proyecto, aunque era necesario para poder ejecutar el device OpenCL.

Tras finalizar la implementación de los objetivos anteriores, se realizaron las pruebas explicadas en el capítulo 5 para satisfacer el **objetivo 4.**

6.3 TRABAJOS FUTUROS

Como trabajos y mejoras futuras, además de todo el desarrollo que se continuará realizando en el BSC para mejorar el runtime y el compilador, la dirección que han de tomar trabajos futuros es la siguiente:

-Mejora de la compatibilidad para realizar includes y linkado OpenCL y CUDA.

-Añadir trazas al runtime de OpenCL, para poder instrumentar la ejecución de procesos OmpSS con extrae.

-Optimizar y mejorar el runtime de OpenCL, permitir la utilización de varios *devices* OpenCL simultáneamente, implementar las copias de memoria entre estos *device*, optimizar la cache…

-Añadir soporte para clusters en la parte del runtime de CUDA.

-Añadir soporte para otros dispositivos en OmpSS.

-Mejorar la cache y scheduling del runtime mediante plugins.

BIBLIOGRAFÍA

[1] B. KIRK, David; W. HWU, Wen-Mei. *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series). Morgan Kaufmann, 2010, 280.*

CUDA:

[2] SANDERS, Jason; KANDROT, Edward. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley Professional, 2010, 312.

[3] FARBER, Rob, CUDA Application Design and Development. Morgan Kaufmann, 2011, 336.

[4] RUETSCH, Greg; OSTER, Brent. *Getting Started with CUDA*, nVidia Corporation, 2008. Disponible en internet <http://www.nvidia.com/content/CUDAzone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf> (consultado por última vez Julio 2012).

OpenCL:

[5] TSUCHIYAMA, Ryoji, et al. *The OpenCL Programming Book*. Fixstars Corporation, 2010, 260.

[6] MUNSHI, Aaftab, et al. *OpenCL Programming Guide.* Addison-Wesley Professional, 2011, 648.

[7] GASTER, Benedict, et al. *Heterogeneous Programming with OpenCL.* Morgan Kaufmann, 2011, 296.

[8] MUNSHI, Aaftab, et al. *OpenCL Parallel Computing on the GPU and the CPU.* 2008. Disponible en internet <http:/s08.idav.ucdavis.edu/munshi-opencl.pdf> (consultado por última vez Agosto 2012)

OmpSS:

[9] BUENO, Javier; MARTORELL, Xavier, et al. "Productive Cluster Programming with OmpSS" En: *Euro-Par 2011 Parallel Processing: 17th International Euro-ParConference, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part I*, Springer,2011, 631.

[10] LABARTA, Jesus. *OmpSS: Programming Clusters of GPUs Made Easy,* Game Developers Conference 2011. Disponible en internet <http://www.nvidia.com/content/PDF/GDC2011/Jesus_Labarta_BSC.pdf> (consultado por última vez Septiembre 2012).

[11] LABARTA, Jesus. *StarSS: A programming model for the multicore era*. Disponible en internet <http://www.prace-ri.eu/IMG/pdf/08_starss_jl.pdf> (consultado por última vez Septiembre 2012).