Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE Máster

Máster en Ingeniería Informática

Celerity: el futuro de la programación paralela en memoria distribuida

Autor:
D. Sergio Miguel López

Tutor:
Dr. D. Arturo González Escribano

Resumen

El gran desarrollo y despliegue actual de los clústeres de aceleración (*accelerator clusters*), debido a la necesidad de aprovechar al máximo la capacidad y eficiencia de la computación en paralelo, hace que se estén investigando y desarrollando una gran variedad de herramientas y modelos de programación para gestionar estos sistemas. Los clústeres de aceleración están formados por multitud de GPUs u otros aceleradores de cómputo localizados en distintos nodos, por lo que entra en juego la gestión de memoria distribuida en dispositivos heterogéneos. La antigua solución de combinar MPI para la memoria distribuida y CUDA, ROCm o una herramienta propia del fabricante para gestionar el paralelismo en los aceleradores es compleja de utilizar y se está quedando desfasada. Por tanto, se necesita un software que combine estos dos modelos de forma eficaz. En este punto aparece Celerity, que se presenta como una API de alto nivel para la programación en memoria distribuida en clústeres de aceleración, cuya característica principal es que consigue reducir enormemente la complejidad de la programación en memoria distribuida en dispositivos heterogéneos sin perder eficiencia. En este trabajo se tratará de analizar en detalle la herramienta Celerity y será comparada con Controller+Hitmap, un software similar desarrollado por el grupo de investigación Trasgo de la Universidad de Valladolid. Con los resultados podremos determinar si Celerity podría llegar a convertirse en el futuro de la programación paralela en memoria distribuida para los clústeres de aceleración.

Palabras clave

Celerity, SYCL, hipSYCL, MPI, CUDA, programación paralela, memoria distribuida, computación heterogénea, clúster de aceleración.

Abstract

The increasing development and current deployment of accelerator clusters, due to the need to take full advantage of the capacity and efficiency of parallel computing, leads to the research and development of a great variety of programming tools and models to manage these systems. But, accelerator clusters are made up of a multitude of GPUs or other computing accelerators located on different nodes. Thus, distributed memory management on heterogeneous devices comes into play. The old solution of combining MPI for distributed memory and CUDA, ROCm or a similar vendor-provided tool to manage the parallelism in the accelerators is complex to use and it is getting out of date. For that reason software that combines these two models in an efficient way is needed. At this point, Celerity appears, presented as a high-level API for distributed memory programming in accelerator clusters. Its main characteristic is that it greatly reduces the complexity of distributed memory programming in heterogeneous devices without losing efficiency. In this work, we will try to analyze the Celerity tool in detail and it will be compared with Controller+Hitmap, similar software developed by the Trasgo research group at the University of Valladolid. With the results we will determine if Celerity could become the future of parallel programming in distributed memory for acceleration clusters.

Keywords

Celerity, SYCL, hipSYCL, MPI, CUDA, parallel programming, distributed memory, heterogeneous computing, accelerator cluster.

Agradecimientos

Agradecer a mi tutor, Arturo por su inestimable ayuda y dedicación, por estar siempre ahí y por brindarme la oportunidad de realizar este trabajo de investigación tan guay.

También quiero agradecer a Fran por echarme una mano con la instalación y puesta en marcha de hipSYCL, a Manuel por proporcionarme los resultados de las pruebas realizadas con la herramienta Controller+Hitmap y a Yuri por sus valiosos consejos para la elaboración de esta memoria.

La investigación desarrollada en este Trabajo Fin de Máster ha sido financiada por el Ministerio de Economía, Industria y Competitividad con el programa FEDER de la Unión Europea, proyecto PCAS (TIN2017-88614-R) y por la Junta de Castilla y León - Becas FEDER, proyectos PROPHET y PROPHET-2 (VA082P17, VA226P20). Las opiniones son de exclusiva responsabilidad del autor que las emite.

Tabla de contenidos

Lista de figuras

1

Introducción

1.1 Contexto

Hoy en día existe un importante auge en el despliegue de clústeres de aceleración o clústeres de GPUs para mejorar el rendimiento de las aplicaciones o programas en paralelo. Esto se debe a varios factores. El primero de ellos, que una gran cantidad de problemas son adecuados para ser ejecutados en paralelo. Además, los clústeres de aceleración que utilizan memoria distribuida proporcionan algunos de los mejores ratios entre coste y eficiencia energética en computación de alto rendimiento (HPC). También, en una entrevista en el año 2015 [**1**], Gordon Moore, el creador de la ley de Moore, que dice que el número de transistores por unidad de superficie en circuitos integrado se duplicará cada dos años, indicaba que esta ley moriría en apenas 10 años. Por estas razones se buscan métodos más eficientes para aumentar el rendimiento de las máquinas con paralelismo. Actualmente, 7 de los 10 supercomputadores que se encuentran en el top 10 del TOP500 (Junio del 2021) [**2**], que es la lista de los 500 supercomputadores con mejor rendimiento del mundo, son clústeres de aceleración. También lo son 8 de los 10 primeros en la lista GREEN500 [**3**], la misma lista pero ordenados en función de la eficiencia energética.

Sin embargo, uno de los principales problemas que tienen los clústeres de aceleración es que son difíciles de configurar y de programar en ellos, ya que combinan dos complejos modelos: la programación de paralelismo masivo de grano fino (millones de hilos con poco trabajo coordinados por un hardware acelerador especializado) y la programación en memoria distribuida de grano grueso (grandes procesos con mucho trabajo comunicándose a través de mecanismos de red). Normalmente, estos clústeres están compuestos por un conjunto de nodos interconectados entre ellos y cada uno con una CPU multinúcleo y una o varias GPUs (*Graphics Proccessing Unit*), aunque estas últimas pueden ser otros aceleradores hardware especializados en tareas concretas como simulaciones, redes neuronales o inteligencia artificial. Primero hay que determinar cómo distribuir los datos entre los nodos, utilizando por ejemplo MPI [**4**]. Después necesitamos un modelo de programación para las GPUs, como puede ser CUDA [**5**]. Y además necesitamos de alguna forma utilizar todos los núcleos de la CPU, con un modelo de programación compartida como OpenMP [**6**]. Un programador tendría que conocer a la

perfección los distintos modelos de programación y tener en cuenta las posibles interferencias que se pueden dar al combinar los modelos, considerando además, que utilizan distintas formas de acceder a la memoria, lo que es una fuente de problemas.

1.2 Motivación

En los últimos años han aparecido dos corrientes programación en los clústeres de aceleración: la primera de ellas es MPI+X, siendo X normalmente CUDA u OpenCL, que mantiene el problema de que los programadores tengan que lidiar con distintos modelos de programación, y la segunda corriente son las librerías o los *frameworks* que proveen una capa de programación más abstracta, ocultando los problemas de despliegue, coordinación y ejecución de las tareas, pero que tienen el problema de que son muy específicos y son difíciles de generalizar. Entre estas dos corrientes encontramos a Celerity [**7**], la posible solución para la programación en los clústeres aceleradores. Celerity es una API de alto nivel para la programación en memoria distribuida en clústeres de aceleración, con distribución automática de trabajo y datos, y que abstrae las estructuras de datos y las funciones de procesamiento de forma más eficiente, haciendo que sea menos complejo que la programación general en memoria distribuida [**8**]. Además, Celerity está basado en el estándar de programación SYCL [**9**], una capa de abstracción multiplataforma basada en C++ moderno para programación paralela. SYCL se puede ejecutar en la mayoría del hardware que soporta OpenCL y también se puede ejecutar en CUDA y AMD ROCm mediante, por ejemplo, la implementación hipSYCL.

Celerity afirma que la facilidad de uso de la herramienta para la programación en clústeres de aceleración es uno de sus objetivos principales, pero que solo se puede llegar hasta cierto punto sin sacrificar un rendimiento considerable. Lo que nos lleva a pensar, ¿Podemos coger un programa cualquiera y obtener un rendimiento bueno? ¿Ese programa tendrá mejor o peor rendimiento que el mismo desarrollado con otras herramientas más convencionales? ¿Hasta qué punto es mejor eficiencia que simplicidad?

1.3 Objetivo

El objetivo de este TFM es realizar un estudio y evaluación del modelo de programación heterogénea distribuida Celerity. Siendo este el objetivo principal, tenemos algunos objetivos secundarios que derivan del principal, como son mostrar una evolución de los modelos de programación paralela hasta llegar a Celerity, explicar cómo funciona Celerity por debajo y mostrar una guía de instalación de la herramienta.

1.4 Metodología

Para conseguir nuestro objetivo, seguiremos el método científico y estableceremos la siguiente hipótesis: un programa ejecutado en un clúster de aceleración es más eficiente si se programa y ejecuta con Celerity comparado con programarlo utilizando las librerías desarrolladas por el grupo Trasgo, Controller+Hitmap [**10**] [**11**], que han demostrado conseguir una eficiencia similar a la que se puede obtener programando directamente con la herramientas de bajo nivel (MPI, CUDA, OpenCL, etc.). Seleccionaremos un tipo de programa concreto que es utilizado habitualmente como caso de estudio de abstracciones de programación paralela, y obtendremos su rendimiento con distintas cargas de trabajo y distinto número de nodos de computación, está será la fase de experimentación. Los resultados obtenidos con las dos herramientas serán comparados estadísticamente para determinar cuál de los dos es más eficiente, y así validar o rechazar la hipótesis inicial.

Durante la última revisión de la memoria de este TFM, los desarrolladores de Celerity han publicado una nueva versión de la herramienta. Las descripciones de Celerity, sus funcionalidades, ejemplos y el trabajo experimental se refieren en todo momento a la versión disponible durante el desarrollo del TFM (v0.2.1). La nueva versión de Celerity (v0.3.0), lanzada el 16 de Noviembre de 2021, no modifica fundamentalmente la sintaxis o forma de programar. Simplemente añade nuevas funcionalidades y mejoras para adaptarse al último estándar SYCL 2020.

Respecto a las medidas de rendimiento obtenidas durante el trabajo experimental, sería necesario repetir la experimentación completa de nuevo, con la nueva versión, para verificar si se han introducido diferencias que afecten al rendimiento y puedan modificar las conclusiones obtenidas.

1.5 Estado del arte

La programación paralela está conquistando todos los ámbitos de la programación, por lo que una de nuestras primeras tareas ha sido realizar una investigación de otras herramientas para automatizar la programación paralela, en clústeres aceleradores y en hardware heterogéneo, y que además sean lenguajes de programación o bibliotecas que estén relacionadas con las funcionalidades y objetivos de Celerity.

- STARPU: STARPU es una librería de alto nivel para generar tareas en paralelo en hardware heterogéneo y para desarrollar algoritmos de programación de forma sencilla [**12**]. A diferencia de Celerity, la distribución del trabajo se tiene que hacer de forma manual.
- SkePU 2: Es la segunda versión de SkePU, que es un *framework* de programación *skeleton* para sistemas paralelos heterogéneos. Utiliza un compilador de código fuente-a-fuente para transformar los programas a otros en el mismo lenguaje pero

que se ejecutan de forma eficiente y paralela [**13**]. No tiene soporte para sistemas de memoria distribuida.

- Kokkos: Kokkos es un modelo de programación y librería C++ que permite obtener una gran portabilidad del rendimiento en arquitecturas *manycore* mediante la unión de abstracciones para el paralelismo de datos de grano fino y para los patrones de acceso a la memoria. Sobre todo, los distintos patrones de acceso a memoria que tienen los dispositivos es lo que más afecta al rendimiento [**14**]. De nuevo, a diferencia de Celerity, el paralelismo en memoria distribuida se tiene que hacer de forma manual.
- PHAST: PHAST es una librería C++ para GPUs de NVIDIA y *multicores*. Se pueden establecer parámetros de programación independientemente del código de la aplicación. Especialmente diseñado para aplicaciones embebidas [**15**]. No tiene soporte para sistemas de memoria distribuida.
- Legion: Legion es muy útil cuando se trata de procesadores heterogéneos con distintas y complejas jerarquías de memoria. Legion proporciona una gran flexibilidad al describir los datos a los que acceden las tareas, hablando de regiones lógicas, cómo colocarlos en la memoria física y dónde ejecutar las tareas, utilizando mapeadores [**16**]. Legion se encarga de garantizar la coherencia de los datos entre los nodos en clústeres heterogéneos. Pero también el particionamiento y la asignación de tareas es tarea del programador, al contrario que en Celerity.
- OpenACC: OpenACC es un modelo de programación de alto nivel y basado en directivas para C, C++ y Fortran. Está diseñado para facilitar la programación en arquitecturas heterogéneas de hardware informático de alto rendimiento [**17**]. Está basado en OpenMP, por lo que también el paralelismo en memoria distribuida se tiene que hacer de forma manual.
- dOpenCL: dOpenCL extiende el estándar OpenCL de manera que varios dispositivos de un nodo en un sistema distribuido se combinan y se usan de manera que parezca una sola aplicación. Como está basado en OpenCL, permite usar los códigos escritos en ese lenguaje, pero añade la característica de que se ejecuten en sistemas distribuidos y heterogéneos [**18**]. Mientras que con dOpenCL solo se pueden utilizar dispositivos con OpenCL, con Celerity podemos utilizar dispositivos NVIDIA y AMD si escogemos la implementación de SYCL hipSYCL o podemos utilizar dispositivos OpenCL si elegimos otra implementación diferente de SYCL, ComputeCpp.

También encontramos interesantes herramientas con un propósito más específico, pero su uso no es generalizable para otras aplicaciones:

- SYCL-DNN: SYCL-DNN es una librería que permite acelerar rutinas, como rutinas matemáticas a bajo nivel con SYCL, OpenCL y CUDA, para operaciones

en redes neuronales sin depender del hardware ni del vendedor. Está escrito en C++ y construido sobre SYCL. Es interesante ya que acelera cálculos relacionados con redes neuronales profundas (DNN) [**19**].

- ARENA: ARENA es una arquitectura en anillo aceleradora reconfigurable y asíncrona, siendo una de las potenciales arquitecturas futuras para los HPC y los centros de datos. Además de los CGRA (*Coarse-Grained Reconfigurable Array*), ARENA pretende crear un nuevo modelo de programación mediante tareas asíncronas en un clúster de nodos reconfigurables, llevando la computación a los datos y no al contrario [**20**].
- CUDA-JMI: Es un sistema que permite acelerar la tarea de extracción de características en sistemas heterogéneos, tarea que es conocida por ser muy lenta en grandes conjuntos de datos [**21**].

1.6 Estructura de la Memoria del TFM

En el Capítulo 2 se describen todas las herramientas, software y programas utilizados durante todo el transcurso de realización del TFM, además de la planificación y seguimiento del proyecto y una descripción del entorno de trabajo.

En el Capítulo 3 se muestra una evolución de la programación en paralelo, con ejemplos que nos ayudaran a entender más fácilmente como son las aplicaciones en Celerity.

En el Capítulo 4 se da una guía de instalación de Celerity y se explica en detalle algunos conceptos fundamentales de este lenguaje.

En el Capítulo 5 se relata cómo funciona la aplicación de Celerity que hemos desarrollado y compararemos la eficiencia de la aplicación con otras desarrolladas con Controller+Hitmap.

La memoria finaliza en el Capítulo 6 con las conclusiones y las líneas de investigación futuras y el Capítulo 7 para las referencias utilizadas.

2

De herramientas y software

2.1 Introducción

En este capítulo se van a nombrar y a describir todas las herramientas y todo el software utilizado para la realización de este trabajo. Estas herramientas serán necesarias para poder ejecutar y desarrollar nuestros programas escritos en Celerity. Son las siguientes:

- El lenguaje de programación C++ como base de todas las otras herramientas.
- Los compiladores GCC, CMake y LLVM.
- La librería Boost.
- MPI para la programación paralela en memoria distribuida.
- CUDA para la programación en GPUs de NVIDIA.
- SYCL, una capa de abstracción multiplataforma para programación paralela.
- hipSYCL, la implementación de SYCL para CUDA.
- Celerity, una API (*Application Programming Interface*) para programación en memoria distribuida en clústeres de GPUs y distribución automática de trabajo y datos, como herramienta principal.
- Las librerías Controller y Hitmap.

Para las ejecuciones de los experimentos se van a utilizar máquinas del clúster de aceleración heterogéneo del grupo Trasgo, en concreto las máquinas llamadas Manticore, Medusa e Hydra. Además, también vamos a disponer de una máquina en el MareNostrum del BSC (*Barcelona Supercomputing Center*), llamada CTE-POWER. En la **Figura 1** se muestra un esquema con todos los recursos: máquinas, herramientas y programas que se van a usar en este TFM.

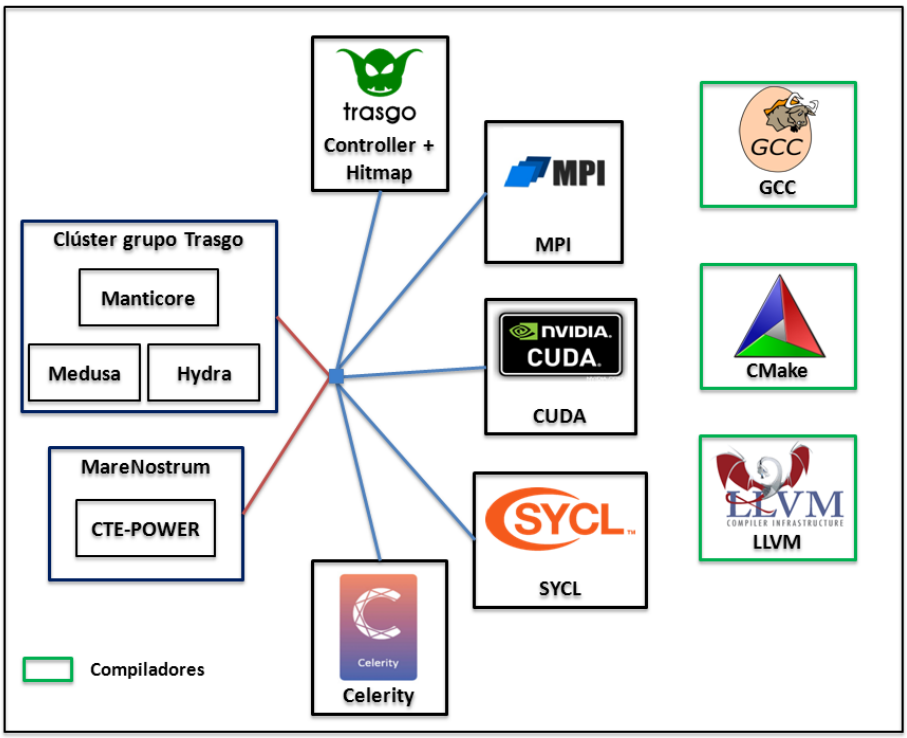

Figura 1: Diagrama con todas las herramientas que se desplegarán en este TFM

2.2 Introducción a CMake y LLVM

CMake es un sistema extensible de código abierto para gestionar los procesos de compilación de manera independiente del compilador en un sistema operativo [**22**]. El diseño de CMake hace que se pueda usar conjuntamente con el entorno de compilación nativo. Mediante unos archivos de configuración sencillos, los *CMakeLists.txt*, se generan archivos de compilación estándar, como son los ficheros *Make* en Unix o los proyectos en Windows MSVC. Implementado inicialmente a mediados del año 2000 y desarrollado en el 2001, Bill Hoffman, Brad King, Ken Martin y otros desarrolladores crearon CMake como respuesta a la necesidad de un entorno potente de construcción multiplataforma para Insight Segmentation and Registration Toolkit (ITK) como parte del Visible Human Project. Con CMake actualmente podemos generar un entorno de compilación para compilar código fuente, donde él creará jerarquías de directorios, enlazará las bibliotecas necesarias, tanto estáticas como dinámicas y se podrá vincular a una aplicación final, todo esto a partir de la serie de comandos que estén en el archivo *CMakeLists.txt*.

Además de CMake, nosotros vamos a utilizar también LLVM. El Proyecto LLVM es una colección de tecnologías *toolchain* y de compilación reutilizables y modulares. Lo que empezó siendo un proyecto de investigación en la Universidad de Illinois, cuya finalidad era la de proporcionar una estrategia de compilación moderna, tanto estática como dinámica, de lenguajes de programación arbitrarios, se ha desarrollado hasta un proyecto general formado por un conjunto de subproyectos, muchos de los cuales están

siendo utilizados en proyectos comerciales, en proyectos de código abierto y en la investigación académica [**23**]. De estos subproyectos, nosotros vamos a utilizar dos:

- El núcleo de LLVM: las librerías del núcleo de LLVM proporcionan un optimizador moderno, independiente de la fuente y el destino, además de soporte de generación de código para un amplio catálogo de procesadores.
- Clang: es un compilador de los lenguajes C, C++ y Objetive-C, desarrollado sobre LLVM, que realiza compilaciones muy rápidas, genera mensajes de advertencia y de error extremadamente útiles y proporciona una plataforma para construir herramientas a nivel fuente.

En nuestro proyecto vamos a usar tanto CMake como LLVM y Clang para compilar todos nuestros programas, así como para compilar SYCL y Celerity.

2.3 Introducción al lenguaje de programación C++

Una vez que tenemos las herramientas de compilación, necesitamos el lenguaje de programación. Escogemos C++, ya que es el lenguaje de programación estándar que se usa en este tipo de proyectos y porque SYCL y Celerity están diseñados como librerías que se incorporan a un programa escrito en C++.

El lenguaje de programación C de alto nivel de propósito general, tiene la ventaja principal frente a otros lenguajes de que el código compilado de un programa escrito en C puede estar muy optimizado en tamaño y estructura, lo que hace que se ejecute más rápido. A su vez, es una desventaja, ya que cada plataforma debe proporcionar su compilador adecuado para poder obtener el código compilado, siempre teniendo en cuenta que el código fuente es independiente de la plataforma. C se creó en los laboratorios Bell de AT&T, en el año 1972, y siempre ha estado unido al sistema operativo de UNIX. Algunas características de C son la riqueza de operadores, su optimización del código, su facilidad de aprendizaje, su transportabilidad y su programación estructurada. C++, inicialmente llamado C con Clases, surgió en 1980 al añadir características como las clases, *inlining* y los argumentos por defecto. Ya en 1984 C con Clases se rediseño a lo que hoy conocemos como C++, al añadir también funciones virtuales, funciones sobrecargadas, operadores sobrecargados, herencia múltiple, tipos genéricos de datos y manipulación de excepciones [**24**]. Para compilar programas C/C++ tenemos GCC (*GNU Compiler Collection*), originalmente escrito para ser el compilador del sistema operativo GNU, ahora sirve también como *frontend* y librerías para C, C++, Objetive-C, Fortran, Ada, Go y D [**25**]. También vamos a utilizar Boost, un conjunto de bibliotecas que añaden más funcionalidades al lenguaje C++ [**26**] y MPC, una biblioteca compleja de punto flotante con redondeo exacto [**27**].

2.4 Introducción a MPI

MPI (*Message-Passing Interface*), en español "Interfaz de Paso de Mensajes", es una especificación de interfaz de librería del modelo de programación paralela de *paso de mensajes*, donde los datos se mueven desde el espacio de direcciones de un proceso al de otro proceso a través de operaciones explícitas generalmente cooperativas. Hay que tener en cuenta que MPI es una especificación, no una implementación. MPI no es un lenguaje de programación, todas las operaciones son funciones, subrutinas o métodos escritos en C y Fortran. MPI ha ido evolucionando desde su creación en 1994 (MPI-1.0) por un conjunto de 40 organizaciones de Europa y Estados Unidos, hasta el estándar actual, MPI-4.0.

MPI es un modelo de paso de mensajes, por lo que tenemos un modelo de programación con memoria distribuida, donde cada proceso tiene su memoria local y si necesita información o datos de otros procesos, es necesaria una paralelización explícita, con mensajes y operaciones explícitas [**4**]. Con MPI podemos usar una red de procesadores independientes para ejecutar nuestras aplicaciones y así podremos aumentar nuestra capacidad de cómputo y de memoria. Pero es responsabilidad nuestra decidir cómo vamos a repartir los datos entre los procesadores, en qué momento tendremos que enviar datos entre procesadores para mantener la consistencia de los datos y qué operaciones vamos a utilizar para ello. MPI es el estándar de facto para programación distribuida, es una tecnología muy consolidada, se usa en todos los supercomputadores y también se puede usar en memoria compartida. Algunas implementaciones de MPI son MPICH [**28**] y Open MPI [**29**]. Nosotros vamos a utilizar MPICH, que es una implementación muy popular, de código abierto y es la base de otras implementaciones de MPI, incluidas algunas de las más usuales en supercomputadores, como MVAPICH [**30**].

2.5 Introducción a CUDA

CUDA (*Compute Unified Device Architecture*) es una plataforma de computación paralela y un modelo de programación desarrollado por NVIDIA para computación general en GPUs (*Graphics Proccessing Unit*). La primera GPU fue creada en 1999 por NVIDIA y a base de ver su gran rendimiento en las operaciones vectoriales, en 2006 lanzaron CUDA, la primera solución para programación de propósito general en GPUs. Mediante CUDA podemos acelerar los programas utilizando la potencia de las GPUs. La parte del programa que es secuencial se ejecuta en la CPU con un solo proceso, mientras que la parte paralela se ejecuta en miles de hilos de la GPU en paralelo. El kit de herramientas de CUDA desarrollado por NVIDIA (CUDA *toolkit*) contiene las bibliotecas, el compilador, las herramientas de desarrollo y el entorno de CUDA. Podemos programar en CUDA con lenguajes como C, C++, Fortran o Python. Las aplicaciones de CUDA van desde *Machine Learning* y *Data Science* hasta química

computacional, fluidos dinámicos, bioinformática o desarrollo de videojuegos, entre otros [**5**].

Las GPUs están especializadas en cálculos paralelos, ocultan las latencias de acceso a la memoria con la computación y están diseñadas de forma que se dediquen más transistores al procesamiento de datos en lugar de al almacenamiento en caché de datos y al control de flujo [**31**]. Una GPU está formada por varios multiprocesadores (SM), cada uno con su unidad de control, su memoria de registros y su memoria compartida (*shared*). Cada SM está formado a su vez por muchos procesadores o *cores* (SP), con sus registros y unidades de cálculo aritmético, como podemos ver en la **Figura 2 a**. Cada uno de los hilos de la GPU se procesa en un SP diferente y cada hilo tiene su identificador global y sus registros propios. Los hilos se agrupan en bloques, **Figura 2 b**, cuyo tamaño es determinado por el programador y cuya elección es muy importante, ya que es crucial para el rendimiento. Todo el bloque se ejecuta en el mismo SM y cada bloque tiene un identificador diferente. Los bloques pueden ser de 1, 2 o 3 dimensiones, al igual que los *grids*. Un *grid* es un conjunto de bloques, cuyas dimensiones también son determinadas por el programador. El hardware se encarga de distribuir los bloques entre los SM. Un *warp* es un conjunto de 32 hilos con índices consecutivos a nivel de bloque. En cada SM, los hilos se planifican y ejecutan en *warps*, con un modelo SIMD (*Single Instruction Multiple Data*) más general que las unidades típicas de las CPUs.

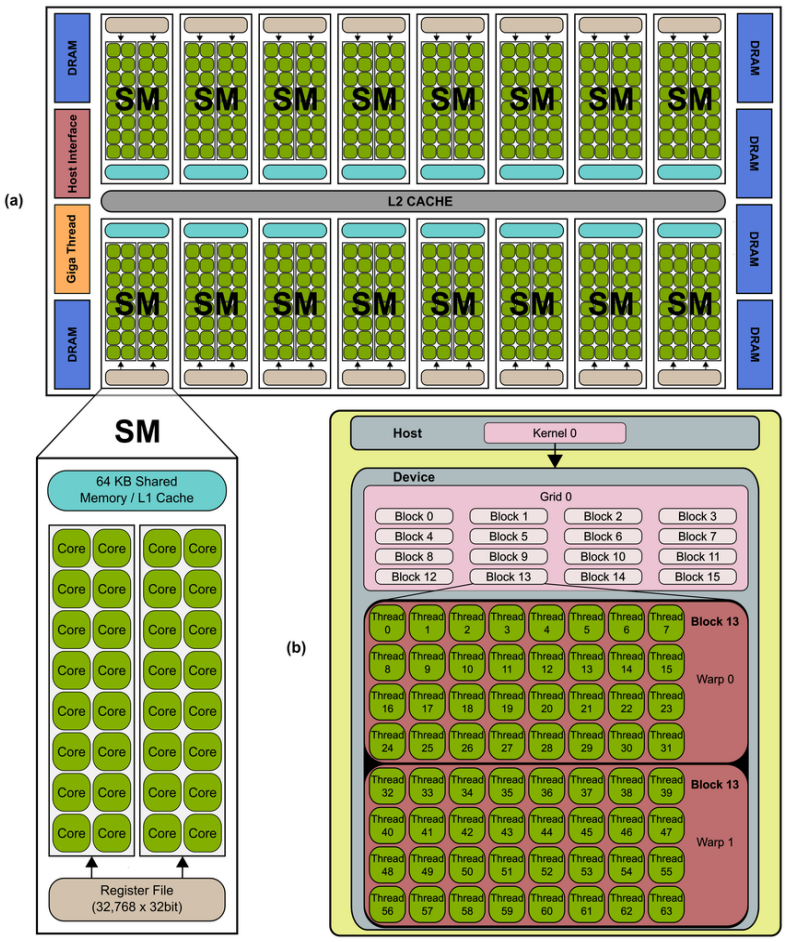

Figura 2: Arquitectura de una GPU. Figura extraída de **[**32**]**

En la **Figura 3** tenemos un ejemplo más claro. Hemos creado un *grid* de 2 dimensiones, con 2x2, lo que hace un total de 4 bloques. A su vez cada bloque tiene unas dimensiones de 4x4, lo que hacen un total de 16 hilos por bloque y por tanto, 64 hilos en total en el *grid*. CUDA nos proporciona una serie de variables para obtener las dimensiones del bloque y *grid* y sus identificadores: *gridDim.x* y *gridDim.y* nos dan las dimensiones del *grid* en el eje X y en el eje Y respectivamente, al igual que *blockDim.x* y *blockDim.y* con las dimensiones del bloque. Los identificadores del hilo en el bloque, en cada dimensión, los obtenemos con las variables *threadIdx.x* y *threadIdx.y* y los identificadores del bloque con *blockIdx.x* y *blockIdx.y*. Si tuviéramos una tercera dimensión, acabaría en *.z*. Con estos datos, el programador puede detallar paramétricamente el trabajo de un hilo, describiendo de forma sintética y abstracta el trabajo de todo un *grid* sobre grandes estructuras de datos. Por último, una fórmula muy útil es la que nos devuelve el identificador global del hilo en el *grid*, en 2 dimensiones:

int globalid = (threadIdx.x + blockIdx.x * blockDim.x) + (threadIdx.y + blockIdx.y * blockDim.y) * blockDim.x * gridDim.x;

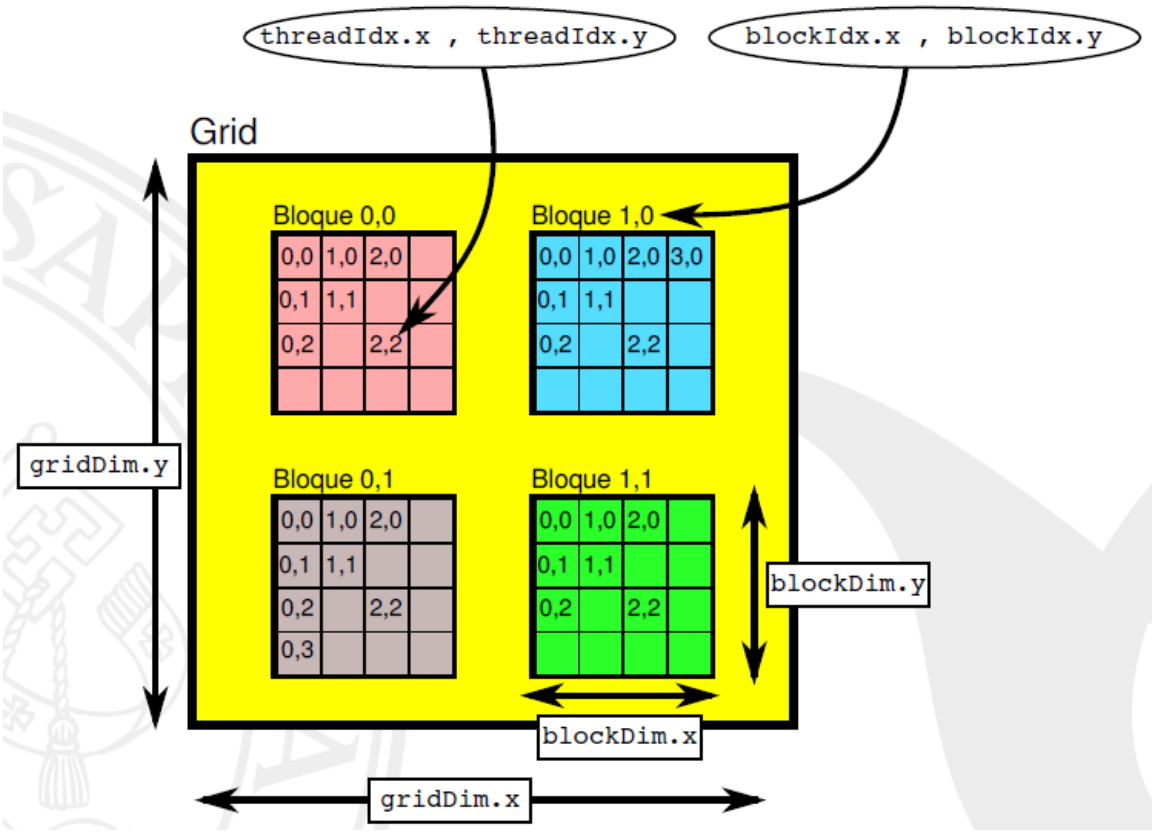

Figura 3: Ejemplo de un *grid* de 2x2 bloques y bloques de 4x4 hilos. Figura extraída de [33]

Nosotros vamos a utilizar CUDA indirectamente, no vamos a programar en CUDA, pero Celerity y SYCL utilizarán funciones de CUDA para reservar memoria en la GPU y para hacer transferencias de datos entre la CPU y las GPUs y para ejecutar los cómputos.

2.6 Introducción a SYCL y hipSYCL

SYCL es una capa de abstracción multiplataforma *Royalty-free* que permite escribir código para procesadores heterogéneos utilizando la ISO estándar de C ++ con el código del *host* o CPU y del *kernel* o GPU para una aplicación contenida en el mismo archivo fuente. SYCL utiliza programación genérica con plantillas (*templates*) y funciones lambda para permitir que el software de aplicación que está por encima se codifique correctamente de forma abstracta, optimizando el código del *kernel* en una amplia gama de *APIs* de aceleración, como OpenCL, CUDA y ROCm. Los programadores escriben a un nivel más alto que la API de aceleración nativa, pero siempre tienen acceso al código de nivel inferior a través de una integración con esta API de aceleración nativa mediante el modo de interoperabilidad, bibliotecas C / C ++ y *frameworks* como OpenCV u OpenMP [**34**].

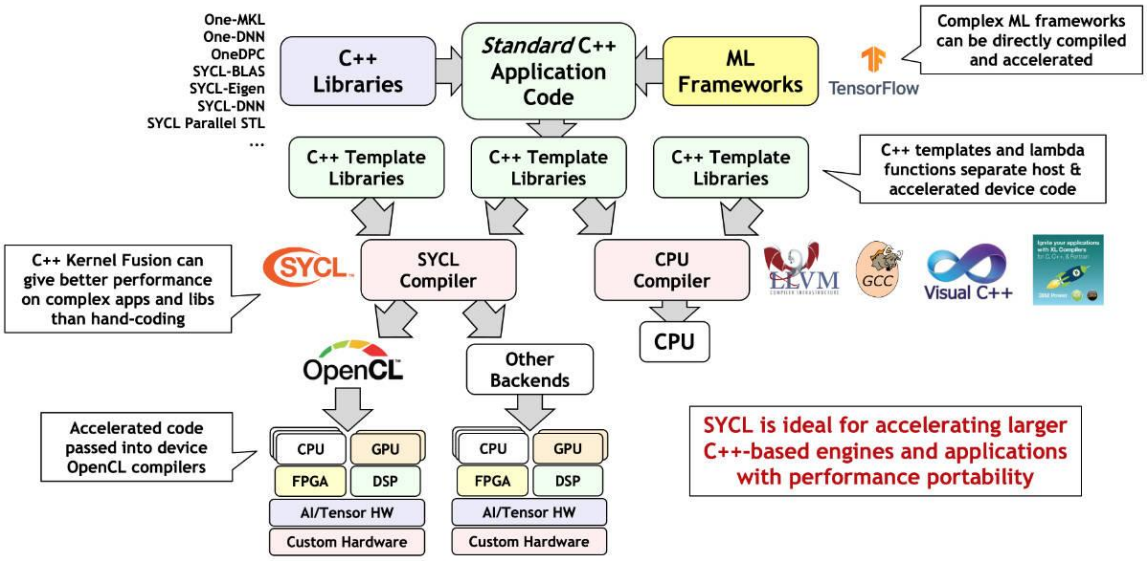

Figura 4: Arquitectura de SYCL. Figura extraída de **[**34**]**

SYCL amplía los conceptos de estándares como el modelo OpenCL de varias formas más allá del uso general de las características de C++, como la ejecución de *kernels* en paralelo en dispositivos heterogéneos de forma flexible, el uso de USM (*Unified Shared Memory*) para movimiento de datos y asignación de memoria, la separación entre el acceso a los datos y el almacenamiento de los datos cuando se usan búferes y *accessors* SYCL (abstracciones de SYCL para acceder a los datos compartidos entre el *host* y el acelerador) y la sintaxis de paralelismo jerárquico [**9**]. La última versión de SYCL, SYCL 2020, ha sido lanzada en Febrero del 2021 y añade reducciones en paralelo, simplificaciones de los *accessors*, interoperabilidad expandida con otros *backends* y mejoras en las operaciones atómicas para hacerlas más parecidas a C++.

Existen varias implementaciones en desarrollo de SYCL, como podemos observar en la **Figura 5**. DPC++ (*oneAPI Data Parallel C++*) está más relacionado con las APIs de programación de oneAPI, ComputeCpp está asociado a dispositivos OpenCL, triSYCL

con OpenMP y TBB (*oneAPI Threading Building Blocks*) para programación paralela en la CPU y neoSYCL es la implementación de SYCL para el SX-Aurora TSUBASA, un sistema informático vectorial en los que se basa la supercomputadora *Earth Simulator* (ES) para ejecutar modelos de predicción del cambio climático.

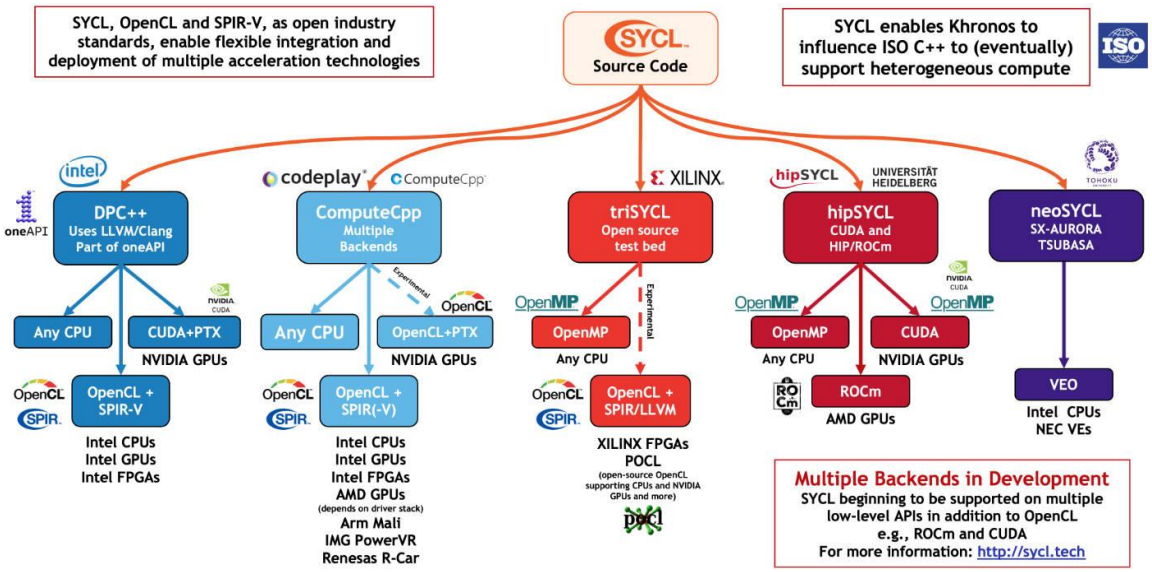

Figura 5: Implementaciones en desarrollo de SYCL. Figura extraída de **[**34**]**

La última de las implementaciones, hipSYCL, es la que nosotros vamos a utilizar en este proyecto, ya que es la implementación de SYCL recomendada por los desarrolladores de Celerity. hipSYCL, inicialmente creado como hobby y ahora desarrollado por la Universidad de Heidelberg, Alemania, está dirigida tanto a CPUs como a GPUs, tratando de aprovechar los *toolchains* existentes, como CUDA o HIP. En concreto, hipSYCL se dirige a cualquier CPU a través de OpenMP, GPUs de NVIDIA a través de CUDA, GPUs de AMD a través de HIP/ROCm y GPUs de Intel a través de Intel oneAPI Level Zero y SPIR-V (muy experimental). Podemos compilar archivos fuente en un solo binario capaz de ejecutarse en todas estas plataformas, cuando se compila con las distribuciones de Clang correctas [**35**] [**36**].

2.7 Introducción a Celerity

Celerity es una API para programación en memoria distribuida en clústeres de nodos aceleradores o GPUs. Está construido sobre SYCL, usando el lenguaje C++, lo que hace que sea fácil migrar programas escritos en SYCL a Celerity. Con Celerity podemos escribir programas paralelos para ejecutar en un clúster de GPUs sin tener que ser un experto en este tipo de programación, y esto es porque Celerity oculta los detalles de cómo un programa se está ejecutando en varios nodos a la vez. En Celerity no tenemos la noción de *ranks* de MPI (identificadores de procesos) y toda la parte del particionado de datos que hacíamos en MPI, se hace de manera transparente por debajo. Esto permite que

estemos más centrados en lo que hace nuestro programa, sin preocuparnos por las complejidades de la programación en paralelo en memoria distribuida. Tiene como objetivo la facilidad de uso sin sacrificar rendimiento. Celerity es un proyecto de investigación que sigue en desarrollo, liderado por Peter Thoman y Philip Salzmann, de la Universidad de Innsbruck, Austria y Biagio Cosenza, de la Universidad de Salerno, Italia [**37**] [**7**]. La **Figura 6** muestra un esquema de cómo los programas son ejecutados por Celerity.

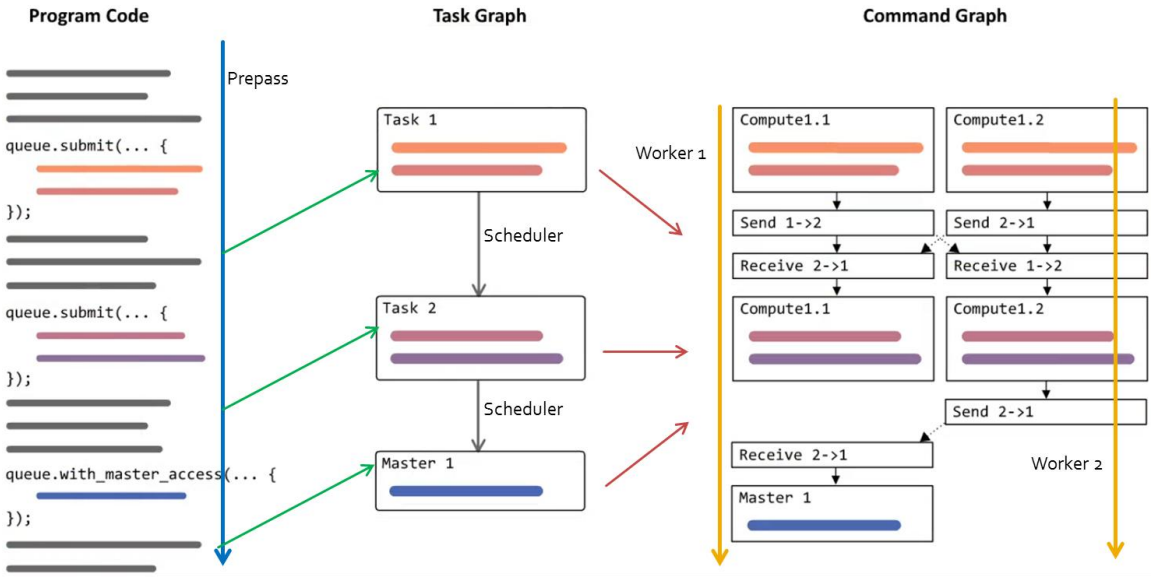

Figura 6: Diagrama de flujo de una aplicación en Celerity. Figura extraída de **[**38**]**

Cuando ejecutamos un programa con Celerity se realiza una primera pasada (*prepass*) similar a la ejecución normal del programa. Cuando Celerity detecta una petición de encolado de una operación (*queue submit*), no pasa directamente a ejecutar esa función o *kernel*, sino que el código de la función junto con información sobre dependencias y metainformación de la planificación se graba en el grafo de tareas (*Task Graph*) de Celerity. Mientras se está ejecutando la primera pasada, un hilo de planificación (*Scheduler*) construye un grafo de comandos (*Command Graph*) más detallado a partir de los nodos del grafo de tareas. El grafo de comandos contiene las instrucciones individuales para cada nodo del sistema y codifica todas las trasferencias de datos (envíos y recepciones) para mantener una vista consistente de los datos, como si se estuviera ejecutando en un solo nodo. En cada proceso, nodo o *worker*, el grafo de comandos es consumido y ejecutado de forma asíncrona, de manera que se ejecutan en algún momento las funciones o *kernels* que se han solicitado en el código, distribuidas en todos los aceleradores del clúster. Las funciones paralelas, al ser asíncronas, pueden producir dependencias de lectura y escritura de los datos. El grafo de tareas se encarga de ordenar las tareas teniendo esto en cuenta y evitando inconsistencias. La última función de la **Figura 6** es un *with_master_access*, que quiere decir que solo la ejecuta el nodo maestro, habitualmente el nodo 0. Estas operaciones se suelen utilizar para imprimir el resultado por pantalla, realizar entrada y salida de datos, etc. Al estar los datos

distribuidos, es necesario que todos los nodos envíen su parte al nodo maestro y sea este el que muestre el resultado o realice la operación correspondiente.

2.8 Introducción a Controller y Hitmap

La librería Hitmap está diseñada para realizar particionado automático de datos y para mapear eficientemente vectores, matrices o estructuras complejas [**10**]. Hitmap utiliza abstracciones para adaptarse al tipo de particionamiento. Para la parte de las comunicaciones y gestión de los aceleradores tenemos la librería Controller, que permite al programador gestionar las comunicaciones y los detalles de lanzamiento de los *kernels* de forma transparente y fácil en clústeres heterogéneos [**11**] [**39**]. Ambas librerías han sido desarrolladas por el grupo de investigación Trasgo, de la Universidad de Valladolid [**40**]. Combinando las dos herramientas podemos obtener un modelo de programación paralela en memoria distribuida, menos abstracto que Celerity, pero con un mayor control de las operaciones de particionado y comunicación.

2.9 Entorno de trabajo

Por último vamos a describir nuestro entorno de trabajo. El grupo Trasgo, con el que desarrollamos este proyecto, nos permite usar su clúster. Este heterogéneo clúster está formado por 5 máquinas conectadas entre sí mediante una red de 1Gb. De las 5 máquinas, nosotros vamos a utilizar solo las llamadas Manticore, Medusa e Hydra. La característica que más nos interesa es cuántas y qué GPUs tienen las máquinas. Manticore tiene 2 GPUs NVIDIA Tesla V100-PCIE-32GB y 2 GPUs AMD Radeon PRO WX 9100. Medusa tiene 3 GPUs NVIDIA GeForce GTX TITAN Black y 1 GPU NVIDIA GeForce GTX TITAN X. Por último, Hydra tiene 2 GPUs NVIDIA GeForce GTX TITAN Black y 2 GPUs NVIDIA Tesla K40c.

Para la gestión del clúster se utiliza la herramienta Slurm, que se encarga de asignar los recursos a los usuarios que solicitan ejecutar tareas en los nodos del clúster y de determinar si un recurso puede ser utilizado o en ese momento es de uso exclusivo de otro usuario hasta que acabe la tarea. Algunos comandos útiles en Slurm son: *srun*, para ejecutar un programa o tarea, *sinfo* para ver información sobre los nodos del clúster, *squeue* para ver si hay alguna tarea ejecutándose en el clúster y que recursos está utilizado y *scancel* para cancelar una tarea. Respecto a las herramientas instaladas, tenemos las siguientes versiones:

- GCC 10.0.1
- CMake 3.17.0
- Clang 13.0.0
- Boost 1.72.0
- MPC 1.0.3
- CUDA 11.3

- ROCm 4.1.0
- Python 3.8.2
- MPICH 3.4.1
- hipSYCL 0.9.1
- Celerity 0.2.1

La **Figura 7** nos muestra un diagrama del clúster con las principales características de cada máquina: su procesador, el número de *cores*, la memoria RAM, el almacenamiento y las GPUs que tiene conectadas.

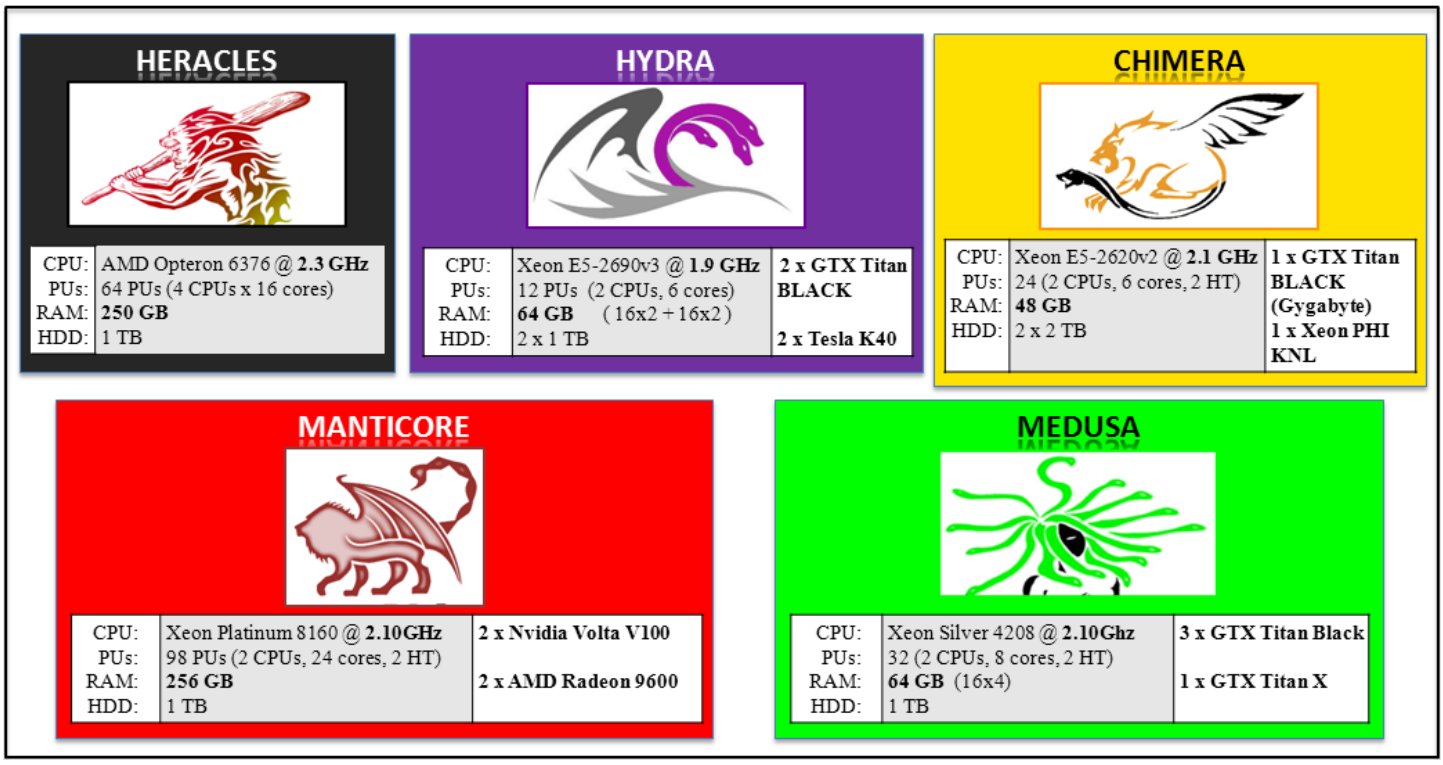

Figura 7: Diagrama del clúster

2.10 Planificación del proyecto

El proyecto se va a dividir en 5 fases: investigación, análisis, programación, pruebas y documentación. El proyecto comenzará el día 7 de Junio de 2021 y terminará el día 3 de Septiembre del mismo año.

La fase de investigación tendrá una duración de 14 días. Esta fase está dividida en dos tareas, una búsqueda bibliográfica de publicaciones relacionadas con el trabajo que vamos a hacer y una investigación de las tecnologías que vamos a utilizar en este proyecto. La duración de la primera tarea será de 8 días, quedando 6 para la segunda tarea. El resultado de esa fase queda documentado en este capítulo. La fase de análisis tendrá una duración de 21 días, siendo 10 de ellos para introducirnos en el estándar SYCL y 11 para la instalación y puesta en marcha de Celerity, y para desarrollar algunos ejemplos para aprender a usar la herramienta. La fase de programación tendrá una

duración de 21 días y la seguirá una fase de pruebas de 35 días, siendo 7 de ellos para el diseño de las pruebas, 21 días para la ejecución de las pruebas y 7 días para la obtención y discusión de los resultados de las pruebas. Ya que las pruebas se van a realizar no solo en el clúster mencionado anteriormente, sino también en el supercomputador del BSC (*Barcelona Supercomputing Center*), vamos a necesitar una gran cantidad de días para esta fase. Por último, la fase de documentación, en la que redactaremos esta memoria, tendrá una duración de 28 días. En la **Figura 8** podemos ver un diagrama de Gantt con la planificación del proyecto.

2.11 Seguimiento del proyecto

A lo largo de la realización del proyecto algunas tareas han sufrido retrasos. La instalación y puesta en marcha de Celerity se ha visto retrasada en 4 días ya que hubo que hacer algunas modificaciones en los ficheros de instalación para que pudiera ser instalada en el clúster. También se debió a que se tuvieron problemas con la instalación de hipSYCL.

En la tarea de programación se necesitó una semana más de lo previsto ya que se encontraron algunos ajustes que tuvimos que implementar en el programa para que fuese lo más eficiente posible.

Por último, la tarea de la ejecución de las pruebas experimentales se prolongó una semana más, puesto que las pruebas realizadas en el clúster del BSC tardaron en ejecutarse más de lo esperado debido a las características de encolado de trabajos de este clúster (hay una gran cantidad de usuarios lanzando sus trabajos para que se ejecuten y si tú lanzas uno, tienes que esperar a tener la prioridad para que tu trabajo sea ejecutado).

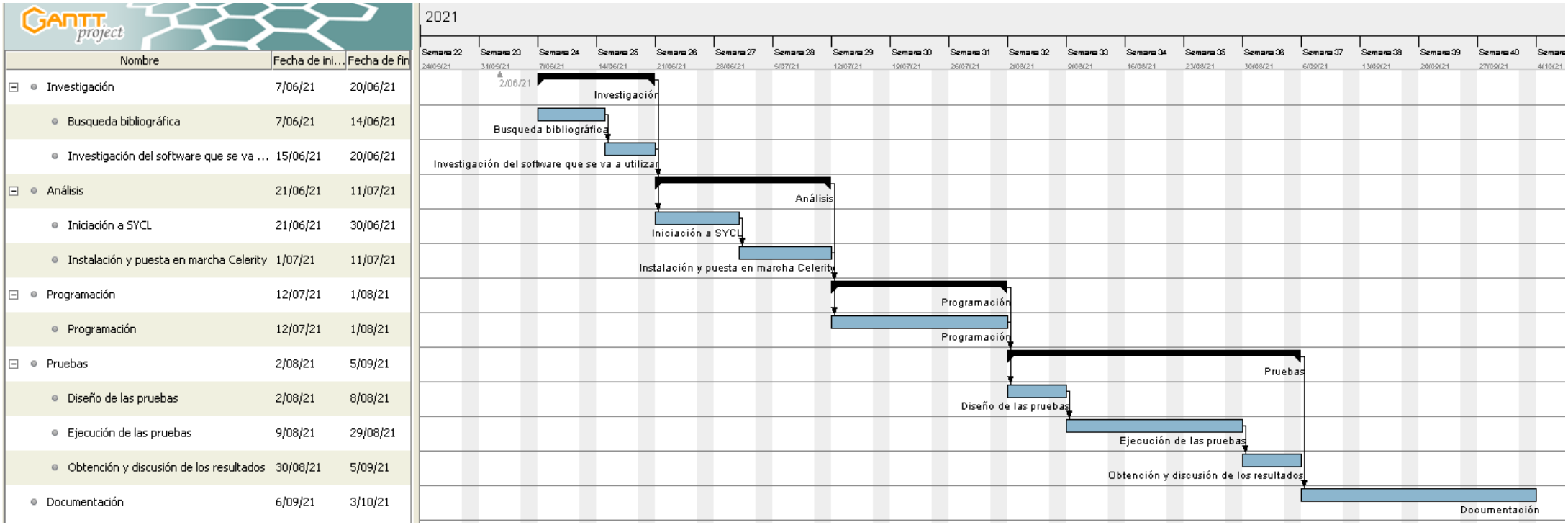

Figura 8: Diagrama de Gantt de la planificación del proyecto

3

Evolución de la programación en paralelo: De MPI a Celerity

3.1 Introducción

Celerity es una herramienta que puede ser un reto si no has trabajado antes con lenguajes como SYCL o similares. Además, su complejidad aumenta con respecto a SYCL teniendo en cuenta que es un modelo de programación paralela en memoria distribuida. Para abordar de la manera más sencilla un programa en Celerity, primero vamos a mostrar y explicar un programa de ejemplo en MPI, que nos valdrá para entender conceptos básicos de los modelos de paso de mensajes y de la memoria distribuida. Continuaremos con CUDA y veremos como programar para las GPUs de NVIDIA. Pasaremos a SYCL, que nos introduce en las abstracciones para la ejecución asíncrona de tareas y en los conceptos de colas, búferes y *accessors*. Finalmente con los conocimientos básicos de estas tres herramientas, explicaremos un ejemplo de programa escrito en Celerity.

3.2 Programación en MPI

Una de las principales ventajas de MPI y la que es de nuestro interés, es la posibilidad de ejecutar grandes cargas de trabajo para un programa en un clúster distribuido, que de otra manera no sería posible al tener que ejecutarse en un único nodo. Pongamos por ejemplo que tenemos un programa que a partir de un array de números enteros, nos devuelve la suma del array. Nuestro programa funciona bien siempre que el tamaño del array sea razonable, pero queremos que se pueda ejecutar con tamaños muy grandes. Con un solo nodo nuestro programa es capaz de procesar un array de 1 millón de elementos. Pero resulta que disponemos de dos máquinas interconectadas, lo que nos permite utilizar dos nodos y el doble de memoria, por lo que podemos valernos de MPI para poder ejecutar nuestro programa, con los cambios necesarios, con un array de hasta 2 millones de elementos. Esto se consigue gracias a que con MPI podemos repartir la carga. Nuestro programa empezaría de la siguiente forma, como vemos en la **Figura 9**:

```c
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>

int main(int argc, char *argv[]) {

        MPI_Init(&argc, &argv);

        int num_procs, rank;
        char maquina[MPI_MAX_PROCESSOR_NAME];
        int maquina_len;

        MPI_Get_processor_name(maquina, &maquina_len);
        MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        printf("Proceso: %d de %d, en maquina: %s\n", rank, num_procs, maquina);
```

Figura 9: Programa ejemplo MPI - Inicialización

Aparte de los ficheros de cabecera habituales *stdio.h* y *stdlib.h*, incluimos *mpi.h*, que contiene la definición de constantes, tipos y prototipos de funciones propias de MPI. La función *MPI_Init()* recibe los punteros a los parámetros *argc* y *argv* de la función *main* y todos los procesos deben llamar a esta función de inicialización. Después tenemos tres funciones útiles: *MPI_Get_processor_name()* guarda en una variable el nombre de la máquina donde está el proceso ejecutándose, *MPI_Comm_rank()*, que guarda en una variable el identificador o *rank* del proceso dentro del grupo asociado al comunicador y *MPI_Comm_size()*, que guarda en una variable el tamaño, en número de procesos, del grupo asociado al comunicador. Un comunicador está compuesto por un contexto y un grupo, donde un contexto define universos separados de mensajes en MPI y un grupo es una colección ordenada de procesos, cada proceso con su identificador o *rank*. El comunicador predefinido MPI_COMM_WORLD contiene todos los procesos. Si llamamos a nuestro programa con dos procesos en la máquina Manticore, el *printf* nos imprimiría: *Proceso 0 de 2 en maquina: Manticore* y *Proceso 1 de 2 en maquina: Manticore*. Los dos procesos ejecutan el mismo código (un programa en MPI ya comienza ejecutándose en paralelo) y terminan imprimiendo la información sobre qué proceso son y dónde se está ejecutando, pero debemos de tener en cuenta que ese *printf* no siempre saldrá en ese orden, ya que habrá veces que el procesador con *rank* 1 sea más rápido que el del *rank* 0 y viceversa. Esto es porque no hay puntos de sincronización si no se escriben específicamente. A continuación, vamos a pasar a la parte de la creación de los arrays y a la parte de cómputo, **Figura 10**:

```
int totalSize=2000000;
int size=totalSize/num_procs;
int begin=size*rank;
long *enteros=(long *)malloc(sizeof(long) * size);
int i;
long suma=0;

for(i=0;i<size;i++) enteros[i]=i + begin;
for(i=0;i<size;i++) suma=suma + enteros[i];
```

Figura 10: Programa de ejemplo MPI - Creación de arrays y cómputo

Hemos dicho que nuestro array es de 2 millones de posiciones, pero que en este hipotético ejemplo no sería posible almacenarlo en memoria, por lo que dividimos el tamaño total entre el número de procesos, 2 en este caso, lo que hace que cada proceso tenga un array de 1 millón de posiciones. Obtenemos también el comienzo de cada proceso en el supuesto array total, el proceso 0 tendría la parte [0,..,999999] y el proceso 1 la parte [1000000,..,1999999]. Creamos el array de enteros de tipo *long*, ya que vamos a utilizar números muy grandes que no entrarían en un *int*. El primer bucle inicializa nuestro array y el segundo bucle realiza el sumatorio. Al finalizar este último bucle, cada proceso tendría la suma de todas las posiciones del array, pero no sabe la suma que ha obtenido el otro proceso. El último paso son los envíos y recepciones, **Figura 11**:

```
if(rank==1) MPI_Send(&suma,1,MPI_LONG,rank-1,288,MPI_COMM_WORLD);

long sumaRecv;
MPI_Status status;
if(rank==0) MPI_Recv(&sumaRecv,1,MPI_LONG,rank+1,288,MPI_COMM_WORLD,&status);

if(rank==0){
        suma=suma + sumaRecv;
        printf("La suma es: %ld\n", suma);
}
MPI_Finalize();
return 0;
```

Figura 11: Programa de ejemplo MPI - Envíos e impresión del resultado

El proceso 1 tiene que enviar su suma al proceso 0, que obtendrá la suma total. Para ello, el proceso 1 llama a la función *MPI_Send()*, cuyos argumentos son: los datos a enviar, la cantidad de elementos que se van a enviar, el tipo de los elementos a enviar (nosotros vamos a enviar 1 *long*), el destino (como es el *rank* 1, el *rank-1* es el 0), la etiqueta del envío (ayuda al proceso que recibe el mensaje a identificar el mensaje) y el comunicador. A su vez el proceso 0 llama a la función de recepción *MPI_Recv()*, cuyos argumentos son: búfer donde se va a recibir los datos, cantidad máxima de elementos a recibir (se pueden recibir menos, pero no más), tipo de los elementos a recibir, *rank* del proceso que se espera que envíe, la etiqueta, el comunicador y una variable de estado (contiene datos útiles sobre el mensaje recibido). Las llamadas a *MPI_Send()* y *MPI_Recv()* son bloqueantes, lo que quiere decir que el proceso 1 se quedará esperando desde que envía a que el proceso 0 reciba el mensaje y el proceso 0 esperará hasta que

reciba el mensaje. MPI también provee otros tipos de envíos e incluso tipos no bloqueantes que se sincronizan explícitamente con otras funciones de MPI. Cuando el proceso 0 reciba la suma que había calculado el proceso 1, lo sumará a lo que él había calculado e imprimirá el resultado. La función *MPI_Finalize()* finaliza el entorno MPI, todos los procesos tienen que llamar a esta función una vez. El código completo se muestra en el **Anexo I: Ejemplo MPI**.

Con este ejemplo hemos visto que ha sido necesario conocer el número de procesos para poder decidir cómo dividir nuestra estructura de datos. Hemos inicializado la estructura basándonos en la división anterior asumiendo por simplicidad que el tamaño del array tiene división entera entre el número de procesos. Tras los cálculos, hemos tenido que establecer el punto donde hacer los envíos de datos indicando explícitamente que datos tenemos que enviar y recibir. Como veremos más adelante, con Celerity no nos tendremos que preocupar de todo esto, ya que lo realizará en segundo plano, haciendo que nos centremos en partes más importantes del programa.

3.3 Programación en CUDA

El modelo de programación paralela de CUDA es diferente al de MPI. MPI se suele utilizar cuando los programas se van a ejecutar en un sistema distribuido, como es un clúster, mientras que CUDA es un modelo distinto, para programación paralela en GPUs de NVIDIA. Podemos ejecutar código secuencial en la CPU y código en paralelo en la GPU. Esta vez el programa de ejemplo es una simple suma de arrays, sumamos la posición *i* del array A con la posición *i* del array B y lo guardamos en el array C. Nuestro programa empezaría de la siguiente manera, mostrado en la **Figura 12**:

```
#include <stdio.h>
#include <cuda.h>
int main(){

        cudaSetDevice(0);
        cudaError_t error;
        int elements=10000;
        int tamBlockX=1024;
        int tamGridX;

        if(elements%tamBlockX==0){
                tamGridX=elements/tamBlockX;
        }else{
                tamGridX=elements/tamBlockX+1;
        }
        dim3 tamBlock(tamBlockX,1);
        dim3 tamGrid(tamGridX,1);
```

Figura 12: Programa de ejemplo CUDA - Tamaños de bloque y de *grid*

Incluimos el fichero de cabecera *cuda.h*, que contiene las funciones de CUDA. La función *cudaSetDevice()* es para escoger el dispositivo GPU de entre todos los instalados en la máquina. Cada dispositivo tiene un identificador y el orden de los dispositivos es

asignado por el *driver*. Por defecto se utiliza el 0, pero se puede obtener una lista de los dispositivos y elegir otro. Creamos una variable *error* del tipo *cudaError_t*, ya que todas las llamadas a funciones de la biblioteca de CUDA tienen que ser comprobadas (no es una obligación, sino una buena práctica). Nuestros arrays serán de 10000 elementos. Ahora tenemos que establecer el tamaño del bloque en todas las dimensiones que vayamos a usar. Como estamos trabajando con arrays de 1 dimensión, bloques de 1 dimensión es suficiente. Si trabajáramos con matrices, sería mejor utilizar 2 dimensiones para el tamaño de bloque y *grid*. Elegir el tamaño de bloque es una decisión crucial que debe tomar el programador para maximizar el rendimiento y una propiedad de los accesos a memoria denominada coalescencia, que permite a la GPU maximizar el rendimiento de los accesos a datos y de la ejecución en general. En este ejemplo no vamos a ser muy estrictos y hemos elegido el tamaño máximo de bloque para nuestro tipo de GPU, 1024, es decir, cada bloque tiene 1024 hilos, en la dimensión X, la única al estar trabajando con una sola dimensión. El cálculo del tamaño del *grid* es muy sencillo, si la división del número de elementos de nuestro array entre el tamaño de bloque es exacta, el resultado de la división es el tamaño del *grid* o lo que es lo mismo, el número de bloques. Si la división no es exacta, necesitaremos un bloque más. Es mejor que nos sobren hilos a que nos falten, aunque tenemos que minimizar el número de hilos sobrantes todo lo posible. En nuestro caso tendremos 10000 / 1024 = 9,76 -> 10 bloques en el *grid*. Finalmente, asignamos los tamaños de bloque y de *grid* mediante el tipo *dim3*. Ponemos un 1 como segundo parámetro ya que solo estamos trabajando con una dimensión, si tuviéramos dos dimensiones, pondríamos los tamaños en el eje de las Y. A continuación, pasamos a la inicialización de los arrays, como vemos en la **Figura 13**:

```
int *A=(int *) malloc(sizeof(int) * (size_t) elements);
int *B=(int *) malloc(sizeof(int) * (size_t) elements);
int *C=(int *) malloc(sizeof(int) * (size_t) elements);
int *Agpu, *Bgpu, *Cgpu;

error=cudaMalloc((void**) &Agpu, sizeof(int) * (size_t) elements);
if(error!=cudaSuccess)
  printf("Error malloc A: %s\n", cudaGetErrorString(error));
error=cudaMalloc((void**) &Bgpu, sizeof(int) * (size_t) elements);
if(error!=cudaSuccess)
  printf("Error malloc B: %s\n", cudaGetErrorString(error));
error=cudaMalloc((void**) &Cgpu, sizeof(int) * (size_t) elements);
if(error!=cudaSuccess)
  printf("Error malloc C: %s\n", cudaGetErrorString(error));

for(int i=0; i<elements; i++){
        A[i]=i;
        B[i]=i+1;
}
```

Figura 13: Programa de ejemplo CUDA - Inicialización de los arrays

Reservamos memoria para los 3 arrays en la CPU y mediante *cudaMalloc()*, para los 3 arrays en la GPU. La memoria de la CPU y la GPU es distinta. Cada llamada a

cudaMalloc() nos devuelve un valor que guardamos en la variable error. Si nos devuelve *cudaSuccess* es que no ha habido ningún error, en caso contrario imprimimos la información del error. Inicializamos los arrays A y B en la CPU, pero necesitamos que los datos estén en la GPU, para ello realizamos las operaciones de la **Figura 14**:

```
error=cudaMemcpy(Agpu, A, sizeof(int) * elements, cudaMemcpyHostToDevice);
if(error!=cudaSuccess)
  printf("Error cudaMemcpy A: %s\n", cudaGetErrorString(error));
error=cudaMemcpy(Bgpu, B, sizeof(int) * elements, cudaMemcpyHostToDevice);
if(error!=cudaSuccess)
  printf("Error cudaMemcpy B: %s\n", cudaGetErrorString(error));

vectorAdd<<<tamGrid,tamBlock>>>(Agpu, Bgpu, Cgpu, elements);
```

Figura 14: Programa de ejemplo CUDA - Copia de datos a la GPU y suma de los arrays

La función *cudaMemcpy()* nos copia los datos entre CPU y GPU, los argumentos de la función son los siguientes: el destino de los datos, el origen de los datos, el número de bytes de datos y el tipo de copia (*cudaMemcpyHostToDevice* copia de la CPU a la GPU, pero existen *cudaMemcpyDeviceToHost*, *cudaMemcpyHostToHost*, y *cudaMemcpyDeviceToDevice*). Esta inicialización se podría hacer también directamente en la GPU mediante un *kernel* o función paralela y así nos evitaríamos hacer las copias. Con los arrays a punto en la GPU, lanzamos nuestro *kernel* de suma. Los argumentos entre <<<>>> son el tamaño del *grid* y el tamaño del bloque, nuestras variables *dim3* que habíamos creado. Después tenemos los argumentos que queremos pasar a nuestra función, los tres punteros a los arrays de la GPU y el número de elementos de cada array. El *kernel* de suma es el mostrado en la **Figura 15**:

```
__global__ void vectorAdd(int *A, int *B, int *C, int elements){

        int tid=threadIdx.x + blockIdx.x * blockDim.x;

        if (tid<elements){
                C[tid]= A[tid]+B[tid];
        }
}
```

Figura 15: Programa de ejemplo CUDA - Función de suma en la GPU

El tipo*__global__* indica que es un *kernel* que puede ser llamado desde la CPU para ser ejecutado en la GPU. Todos los hilos del *grid* ejecutan el *kernel* a la vez (a la vez teóricamente, puede que unos hilos empiecen más tarde, tarden menos, etc.). Obtenemos el id del hilo utilizando el id del hilo en su bloque en el eje X, el id del bloque en el eje X y las dimensiones del bloque en el eje X. Cada hilo guardará en esa variable su identificador global en el *grid*. La fórmula utilizada sirve cuando estamos en 1 dimensión, varía según las dimensiones utilizadas. Hemos dicho cuando decidíamos el número de bloques o tamaño del *grid* que teníamos que tener mínimo un número de hilos mayor o igual al número de elementos que vamos a procesar, de manera que cada hilo haga la suma de su posición, pero nos sobran hilos. Los hilos con un id mayor al número

de elementos no actúan. Los hilos restantes, cogen el dato del array A de su posición, lo suman a la posición correspondiente de B y lo guardan en su posición en el array C. Cada hilo solo actúa leyendo y escribiendo en su posición, por lo que no se dan condiciones de carrera. Terminamos el programa imprimiendo el resultado, lo vemos en la **Figura 16**:

```
cudaMemcpy(C, Cgpu, sizeof(int) * elements, cudaMemcpyDeviceToHost);
if(error!=cudaSuccess)
  printf("Error cudaMemcpy C: %s\n", cudaGetErrorString(error));

for(int i=0; i<elements; i++){
        printf("%d ", C[i]);
}
printf("\n");

cudaFree(Agpu);
cudaFree(Bgpu);
cudaFree(Cgpu);
cudaDeviceReset();
```

Figura 16: Programa de ejemplo CUDA - Copia de datos al *host* e impresión del resultado

Tenemos nuestro resultado en el array C de la GPU, que no podemos imprimir directamente, por lo que tenemos que copiar los datos a la CPU. De nuevo con un *cudaMemcpy()*, pero ahora del tipo *cudaMemcpyDeviceToHost*, es decir, de la GPU a la CPU. Finalmente imprimimos el resultado. La función *cudaFree()* libera memoria de los arrays en la GPU y *cudaDeviceReset()* libera los recursos de la GPU. El código completo se muestra en el **Anexo II: Ejemplo CUDA**.

Con este ejemplo y el de MPI hemos visto como cada herramienta resuelve un tipo de problema diferente, pero MPI y CUDA no son excluyentes. Muchas aplicaciones se benefician de una combinación de ambas, como es Celerity. Con MPI + CUDA podemos mejorar un programa que tarda mucho en MPI utilizando GPUs en vez solamente la CPU y a su vez podemos mejorar un programa en CUDA que tarda mucho distribuyendo la carga de trabajo en procesos con MPI.

3.4 Programación en SYCL

Nuestro tercer ejemplo es de SYCL. Podríamos haber pasado directamente a Celerity, pero ya que Celerity se basa y usa una gran cantidad de funciones de SYCL, es más sencillo explicar los conceptos básicos de una aplicación SYCL primero. Mientras que con CUDA solo podemos programar para dispositivos de NVIDIA y con su lenguaje, SYCL usa solamente C++ estándar y su librería de objetos y funciones. Después podemos elegir la implementación de SYCL que se ajuste a nuestras necesidades entre un amplio rango de plataformas heterogéneas. Nosotros hemos elegido hipSYCL, ya que soporta GPUs de NVIDIA y es la implementación de SYCL recomendada por los desarrolladores de Celerity. El programa de ejemplo es el mismo que con CUDA, una suma de vectores. Nuestro programa principal es el mostrado en la **Figura 17**:

```
#include <CL/sycl.hpp>
int main(){
        cl::sycl::queue q;

        std::vector<float> A = {1.0, 2.0, 3.0, 4.0, 5.0};
        std::vector<float> B = {1.0, 2.0, 3.0, 4.0, 5.0};

        auto C = add(q, A, B);

        for(int i=0; i<C.size();i++){
                printf("%lf ", C[i]);
        }
}
```

Figura 17: Programa de ejemplo SYCL - Función principal del programa

Incluimos la librería de SYCL, *CL/sycl.hpp*, que contiene la cabecera de las funciones de SYCL. Lo primero nuevo que tenemos es el concepto de cola (*queue*), los *kernels* o funciones paralelas que se ejecutan en el dispositivo se tienen que ejecutar a través de colas. El programador crea al menos una cola, que referencia a un contexto, plataforma y dispositivo, elegidos automáticamente o manualmente. Una cola planifica el orden de ejecución de los *kernels* y detecta las posibles dependencias entre ellos incluso si se están ejecutando bajo otra cola, por ejemplo un *kernel* que accede en modo lectura y otro posterior en modo escritura al mismo array. Una vez creada la cola, inicializamos nuestros arrays, que para este ejemplo son vectores C++. Llamamos a la función de suma (esta todavía no es un *kernel*), con nuestra cola y los dos vectores. Esta función nos devuelve un tercer vector. El modificador *auto* sirve de inferencia automática, es decir, el compilador inferirá el tipo de puntero de C. Finalmente imprimimos el resultado. Nos centramos a continuación en la función de suma de la **Figura 18**:

```
std::vector<float> add(cl::sycl::queue& q, const std::vector<float>& A,
  const std::vector<float>& B){

   std::vector<float> C(A.size());
   cl::sycl::range<1> elements(A.size());

   {
   cl::sycl::buffer<float> buff_A(A.data(), A.size());
   cl::sycl::buffer<float> buff_B(B.data(), B.size());
   cl::sycl::buffer<float> buff_C(C.data(), C.size());

   q.submit([&](cl::sycl::handler& cgh){
      auto access_A = buff_A.get_access<cl::sycl::access::mode::read>(cgh);
      auto access_B = buff_B.get_access<cl::sycl::access::mode::read>(cgh);
      auto access_C = buff_C.get_access<cl::sycl::access::mode::write>(cgh);

      cgh.parallel_for<class vectorAdd>(elements, [=] (cl::sycl::id<1> tid) {
         access_C[tid] = access_A[tid] + access_B[tid];
      });
   });
   }
   return C;
}
```

Figura 18: Programa de ejemplo CUDA - Función de suma de vectores

Creamos el vector C para el resultado de la suma, con el mismo tamaño que los otros vectores. Lo siguiente que definimos es el rango de hilos que vamos a utilizar, *range<>*. El concepto de rango es similar al de bloque y *grid* de CUDA, pero más sencillo, es como si no tuviéramos *grid*, solo un bloque del tamaño que queramos. El número o constante entre los <> indica las dimensiones. Como tenemos un vector de una sola dimensión, ponemos un 1 y el rango o número de hilos en este caso sería el tamaño del vector. Si por ejemplo tuviéramos una matriz de 1024x512, podríamos poner el rango *cl::sycl::range<2> elements(1024,512)*. A continuación creamos tres búferes de SYCL con los datos y tamaños de los vectores. Tenemos que usar búferes de SYCL ya que son con los que trabajan los *kernels* de SYCL. Los búferes se pueden crear de 1, 2 o 3 dimensiones, en nuestro caso, de 1. Se suele meter toda la parte en que se trabaja con SYCL en un bloque de {} para asegurar que todas las tareas de SYCL se completan antes de salir del bloque. En nuestro ejemplo empieza antes de la creación de los búferes y acaba antes del retorno de la función.

La siguiente línea es muy importante, un *submit* envía un grupo de comandos (*command group*) a la cola de ejecución. Los grupos de comandos incluyen la especificación de los requisitos de acceso a los datos y el código del *kernel* que utiliza esos datos mediante el controlador del grupo de comandos (*handler*) *cgh*. Este controlador es pasado como argumento a la función lambda que constituye el grupo de comandos y se usa para solicitar acceso a los datos. En esta función lambda utilizamos el paso por referencia [&]. Para que el entorno de ejecución de SYCL determine las posibles dependencias entre *kernels* y pueda planificar las transferencias de datos, tenemos que definir los tipos de acceso a los datos. Utilizando el controlador del grupo de comandos *cgh* como argumento, solicitamos acceso a los tres búferes mediante la función *get_access<>()*. El tipo de acceso se especifica entre los <> y puede ser lectura, escritura, lectura y escritura, escritura descartante, lectura y escritura descartante y acceso atómico. Los modos descartantes asumen que los datos no están inicializados. Nosotros accedemos en modo lectura a los arrays A y B y en modo escritura al array C. Con esta operación conseguimos tres *accessors*, que son los que usaremos en el *kernel*.

Por último, las siguientes tres líneas son el código del *kernel*, la función paralela. Llamamos al *kernel* con un método con una plantilla (*template*) predefinida en SYCL, un *parallel_for<nombre del kernel>*. Como argumentos tenemos el número de hilos que establecimos con *range<>* y el nombre del id del hilo con *cl::sycl::id<>*. El id del hilo, en este caso llamado *tid*, ya es el id global. No necesitamos fórmulas como en CUDA para calcularlo. La operación de este *kernel* es la suma de A con B en C y usamos los *accessors* y el id del hilo. De nuevo, cada hilo hace la suma de su posición y lo guarda en la posición correspondiente de C. Retornamos el array C y volveríamos al programa principal para imprimir el resultado. El código completo se muestra en el **Anexo III: Ejemplo SYCL**.

Como hemos visto con este ejemplo, la programación en SYCL es parecida a CUDA en cuanto al concepto de *kernels*, pero con algunas diferencias, como que no existen ni los bloques ni los *grids*, sino que se ponen directamente el número de hilos a utilizar. Por otro lado, aunque la semántica de los grupos de comandos y de los *accessors* puede parecer compleja al principio, en poco tiempo te habitúas a ella y resulta más mecánica cuando tienes que hacer varios *kernels* para un mismo programa. La gran ventaja es que los *accessors* expresan cuándo y cómo se accede a los datos, de forma que el sistema de ejecución puede autorizar las transferencias entre *host* y dispositivos. En la siguiente sección vamos a ver cómo es posible convertir un programa SYCL en Celerity con apenas unos pocos cambios en el código.

3.5 Programación en Celerity

La explicación de los tres ejemplos que hemos dado tiene una razón, y es que Celerity se basa en estas tres herramientas para funcionar. Básicamente un programa Celerity es un programa SYCL distribuido. Celerity utiliza muchas de las funciones de SYCL y deja a este la ejecución de los *kernels*. Internamente usa CUDA para la gestión de las GPUs de NVIDIA realizando los cálculos de las funciones paralelas y coordinando todos los procesos a través de MPI, encargándose del reparto de datos, la planificación y la comunicación entre nodos para aprovechar todos los recursos del clúster. En el último ejemplo de este capítulo vamos a ver la como el mismo programa, con algunos cambios, puede ser ejecutado por Celerity. La **Figura 19** muestra el *main* del programa:

```
#include <CL/sycl.hpp>
#include <celerity/celerity.h>
int main(){
   celerity::distr_queue q;

   std::vector<float> A = {1.0, 2.0, 3.0, 4.0, 5.0};
   std::vector<float> B = {1.0, 2.0, 3.0, 4.0, 5.0};

   celerity::buffer<float,1> buff_A(A.data(), A.size());
   celerity::buffer<float,1> buff_B(B.data(), B.size());
   celerity::buffer<float,1> buff_C(A.size());

   add(q, buff_A, buff_B, buff_C);

   q.submit([=](celerity::handler& cgh){

      auto C = buff_C.get_access<cl::sycl::access::mode::read,
        cl::sycl::access::target::host_buffer>(cgh, celerity::access::all<1>());

      cgh.host_task(celerity::on_master_node, [=]() {
         for(int i=0; i<A.size(); i++)
            printf("%lf\n", C[i]);
         });
   });
}
```

Figura 19: Programa de ejemplo Celerity - Función principal del programa

Incluimos las cabeceras de la librería de SYCL y la de Celerity, *celerity/celerity.h*, las de CUDA y MPI ya las incluye Celerity en su código. Creamos una cola, pero ahora es

del tipo *distr_queue*. Solo podemos tener una cola por programa, mientras que en SYCL podíamos tener varias. Las funciones de la cola son las mismas que en SYCL. Creamos nuestros vectores y los búferes, ahora de Celerity. En Celerity aunque los búferes sean de 1 dimensión también hay que escribirlo en la plantilla, en SYCL si no se ponía dimensión, se asumía que era 1. Llamamos a la función de suma, que como veremos en la **Figura 20**, ahora es *void*. En Celerity no podemos hacer que una función que vaya a usar un grupo de comandos devuelva un valor. Esto es porque los *kernels* se ejecutan de manera asíncrona y el *main* se puede llegar a ejecutar antes mientras todavía los trabajos están en cola y por tanto no devolvería nada. En SYCL, aunque también es asíncrono, establecíamos bloques entre {} para asegurar que todas las tareas habían terminado antes de retornar.

Antes de pasar a la explicación de la función de suma, en el programa principal vemos que tras esta función, tenemos un grupo de comandos en los que se accede a C en modo lectura con la propiedad *host_buffer*, que indica que va a ser accedido desde la CPU. Si nos fijamos, en los parámetros del *accessor*, además de *cgh*, tenemos un *celerity::access:all<1>()*. Todos los accesos a búferes de Celerity tienen que indicar un mapeador (*range mapper*). El mapeador *all* indica que se quiere acceder a todo el vector y combinado con el *host_buffer*, que se quiere acceder a todo el vector desde la CPU. Esta operación se debe introducir porque con Celerity los búferes están repartidos entre los procesos, cada uno tiene una parte, pero para poder imprimir todo el vector tendríamos que recuperar cada parte, juntarlo e imprimirlo, como ocurría con MPI. Esta es la manera de hacerlo en Celerity. En este caso el *kernel* indica que es una operación que se va a hacer en la CPU o *host* (*host task*) y en concreto en el nodo maestro, típicamente en el proceso 0 (*master node*). La impresión del resultado es el bucle de siempre, ya que Celerity se ha encargado por debajo de planificar una tarea en la que va a leer el búfer C y va a enviar todos los trozos del búfer al nodo 0 y este será el único que imprima el resultado. Estamos seguros que esta tarea se va a ejecutar después de la suma ya que en la suma accedemos a C en modo escritura y en la impresión es en modo lectura, por lo que hay una dependencia que Celerity planifica correctamente. La función de suma se muestra en la **Figura 20**:

```
void add(celerity::distr_queue& q, celerity::buffer<float,1> A,
  celerity::buffer<float,1> B, celerity::buffer<float,1> C){

   cl::sycl::range<1> elements{A.get_range()};

   q.submit([=](celerity::handler& cgh){
      auto one_to_one = celerity::access::one_to_one<1>();
      celerity::accessor access_A{A, cgh, one_to_one, cl::sycl::read_only};
      celerity::accessor access_B{B, cgh, one_to_one, cl::sycl::read_only};
      celerity::accessor access_C{C, cgh, one_to_one, cl::sycl::write_only, cl::sycl::no_init};

      cgh.parallel_for<class vectorAdd>(elements, [=] (cl::sycl::id<1> tid) {
         access_C[tid] = access_A[tid] + access_B[tid];
      });
   });
}
```

Figura 20: Programa de ejemplo Celerity - Función de suma de vectores

Creamos de nuevo un *range*<> para el número de hilos y también creamos el grupo de comandos, pero ahora la función lambda es por valor [=], ya que en Celerity es posible que llame varias veces a los grupos de comandos durante la ejecución del programa. Definimos los *accessors* al modo Celerity, aunque creándolos con la sintaxis de SYCL también es compatible. Decimos el búfer al que accedemos, el *cgh*, el mapeador y el modo de acceso. En A y B accedemos en modo lectura y en C en modo escritura. La propiedad de *no_init* indica que los datos de ese búfer no están inicializados. El mapeador ahora es del tipo *one_to_one*, que indica que cada hilo solo va a acceder a su posición. Si hiciéramos que, con este mapeador, accediera a la posición *id+1*, por ejemplo, podría causar problemas, ya que tenemos que tener claro que el array está distribuido y que por tanto, la posición *id+1* la puede tener otro proceso. Existen otros mapeadores que resuelven este problema y que veremos en el próximo capítulo. Por último, el *kernel* de la suma es igual que en SYCL. El código completo se muestra en el **Anexo IV: Ejemplo Celerity**.

Hemos dicho que Celerity utiliza MPI para la distribución de la carga de trabajo entre los procesos, pero viendo el código del programa de ejemplo en ningún momento hemos visto indicadores de que estamos trabajando explícitamente en memoria distribuida, como el concepto de *rank*, la distribución de datos entre procesos o el número total de procesos que están ejecutándose. Es una de las principales ventajas de Celerity, y es que nos oculta todo esto, se encarga de todo distribuyendo la carga de trabajo de los *kernels* dinámicamente en tiempo de ejecución entre los procesos y nos da la impresión de que estamos trabajando sin memoria distribuida. En el próximo capítulo entraremos en detalle en Celerity y mostraremos el programa que vamos a utilizar para medir el rendimiento de Celerity.

4

Las claves de Celerity

4.1 Introducción

En este capítulo se explicará en primer lugar como se instala hipSYCL y Celerity. A continuación, veremos como compilar programas tanto en hipSYCL como en Celerity. Esto es esencial, a menudo en las guías de instalación se pasa por alto y en estos lenguajes en concreto es necesario configurar ciertos archivos para que se pueda compilar. Después entraremos en detalle en dos aspectos fundamentales de Celerity. El primero es cómo Celerity funciona por debajo, los grafos de tareas y los grafos de comandos. El segundo aspecto son los mapeadores, un mecanismo que permite a Celerity saber cómo se va a acceder a los búferes de datos y que le ayuda a tomar decisiones de planificación, reparto y comunicación entre nodos.

4.2 Instalación de hipSYCL

Para la instalación de hipSYCL nos vamos a guiar con las instrucciones de su repositorio oficial [**41**]. Necesitamos tener de antemano instalado los compiladores LLVM y Clang con la versión 8 al menos, CUDA con la versión 9.2 al menos, una versión reciente de ROCm (es como CUDA, pero para GPUs de la marca AMD), un compilador C++ con soporte para OpenMP, GCC nos sirve, Python 3, CMake y Boost. Todas estas herramientas las tenemos instaladas ya en nuestro clúster. Primero descargamos el repositorio utilizando la herramienta Git con el siguiente comando:

git clone https://github.com/illuhad/hipSYCL

La versión que nos descarga es la más reciente, en este momento la v0.9.1. El comando anterior nos habrá creado una carpeta llamada *hipSYCL*, y nos movemos a ella:

cd hipSYCL

Por las características de nuestro clúster, en nuestro caso tenemos que abrir el archivo *CMakeLists.txt* y pegar una serie de líneas para incluir las rutas de algunas herramientas para la compilación. Pegamos estas líneas tras la línea *project(hipSYCL VERSION ${HIPSYCL_VERSION_MAJOR}.${HIPSYCL_VERSION_MINOR}.${HIPSYCL_VERSION_PATCH}):*

set(CLANG_EXECUTABLE_PATH /frontend/sergmig/local/llvm12/bin/clang++)

set(LLVM_DIR /frontend/sergmig/local/llvm12/lib/cmake/llvm)

set(CLANG_INCLUDE_PATH
/frontend/sergmig/local/llvm12/lib/clang/13.0.0/include)

set(HIP_CLANG_INCLUDE_PATH
/frontend/sergmig/local/llvm12/lib/clang/13.0.0/include)

set(CUDA_TOOLKIT_ROOT_DIR /opt/cuda-11.3)

set(WITH_CPU_BACKEND ON)

set(WITH_ROCM_BACKEND ON)

set(WITH_CUDA_BACKEND ON)

set(ROCM_PATH /opt/rocm-4.1.0)

set(DISABLE_LLVM_VERSION_CHECK ON)

Guardamos los cambios. Sin movernos de carpeta, creamos un directorio para la compilación, lo llamaremos *build* y nos movemos a él:

mkdir build; cd build

Ejecutamos el siguiente comando, que termina la configuración del CMake y nos genera un archivo *Makefile*:

cmake .. -DCMAKE_INSTALL_PREFIX=/frontend/sergmig/local/hipSYCL2021

La ruta que hemos escrito es la ruta donde se nos va a instalar hipSYCL. El último paso termina la instalación:

make install

Antes de poder compilar un programa SYCL, vamos a tener que añadir la ruta del directorio de instalación al *PATH*. Veremos cómo hacerlo en la siguiente sección, tras instalar Celerity.

4.3 Instalación de Celerity

Para la instalación de Celerity vamos a seguir las instrucciones de su página web [**42**]. Necesitamos tener de antemano instalado un compilador C++ 17, una implementación de MPI versión 2 como mínimo (nosotros usaremos MPICH), la librería Boost y finalmente una implementación de SYCL, como hipSYCL. Primero descargamos el repositorio utilizando la herramienta Git con el siguiente comando:

git clone https://github.com/celerity/celerity-runtime.git

La versión que nos descarga es la más reciente, en este momento la v0.2.1. El comando anterior nos habrá creado una carpeta llamada *celerity-runtime*, y nos movemos a ella:

cd celerity-runtime

Creamos un directorio para la compilación, lo llamaremos *build* y nos movemos a él:

mkdir build; cd build

Ejecutamos el siguiente comando, que termina la configuración del CMake y nos genera un archivo *Makefile*:

cmake -G "Unix Makefiles" .. -DCMAKE_PREFIX_PATH= /frontend/sergmig/local/hipSYCL2021/lib -DHIPSYCL_PLATFORM=cuda -DHIPSYCL_GPU_ARCH=sm_60 -DCMAKE_INSTALL_PREFIX= /frontend/sergmig/local/celerity -DCMAKE_BUILD_TYPE=Release

En *DCMAKE_PREFIX_PATH* tenemos que escribir la ruta de donde tenemos instalado hipSYCL, en *DHIPSYCL_GPU_ARCH* la arquitectura de nuestras GPUs y en *DCMAKE_INSTALL_PREFIX* la ruta donde queremos que nos instale Celerity. Tenemos también que exportar la biblioteca Boost a *LIBRARY_PATH*:

export LIBRARY_PATH=$HOME/local/boost/lib/:$LIBRARY_PATH

El último paso termina la instalación:

make install

Para poder compilar y ejecutar programas SYCL y Celerity tenemos que añadir y exportar las rutas de los ejecutables y librerías de las herramientas que utilizan. Tenemos creado el archivo *senv-vars21.sh*, y con el siguiente comando establecemos nuestras variables de entorno:

source /$HOME/local/bin/senv-vars21.sh

La **Figura 21** muestra el contenido de este archivo:

```
PY3=/frontend/sergmig/local/py3
ROCM=/opt/rocm-4.1.0
HIPSYCL=/frontend/sergmig/local/hypSYCL2021
BOOST=/frontend/sergmig/local/boost
LLVM=/frontend/sergmig/local/llvm12
CELERITY=/frontend/sergmig/local/celerity

PATH=/frontend/sergmig/local/bin:$PY3/bin:$ROCM/bin:$LLVM/bin:$HIPSYCL/bin:$CELERITY:$PATH

export LD_LIBRARY_PATH=$HIPSYCL/lib:/frontend/sergmig/local/lib:$PY3/lib:
/frontend/sergmig/local/lib64:$BOOST/lib:$LLVM/lib:$LLVM/lib/clang/13.0.0/lib/linux:
$ROCM/lib:$ROCM/lib64:$CELERITY/lib:$LD_LIBRARY_PATH
export LD_RUN_PATH=/frontend/sergmig/local/lib:/frontend/sergmig/local/lib64:
$PY3/lib:$ROCM/lib:$ROCM/lib64:$LLVM/lib/clang/13.0.0/lib/linux:$LD_RUN_PATH
export CC=/frontend/sergmig/local/bin/gcc
export CXX=/frontend/sergmig/local/bin/g++
export CPP=/frontend/sergmig/local/bin/cpp
export LD=/frontend/sergmig/local/bin/ld
export CPATH=$BOOST/include:$CPATH
export LIBRARY_PATH=/frontend/sergmig/local/include:/frontend/sergmig/local/boost/lib:$LIBRARY_PATH
export CPPFLAGS="-L/frontend/sergmig/local/boost/lib/"
```

Figura 21: Contenido del archivo *senv-vars21.sh*

4.4 Como compilar programas SYCL y Celerity

Uno de los inconvenientes principales que se tiene cuando se usan herramientas nuevas y en desarrollo es que a veces no se dispone de suficiente información para hacer algunas operaciones con la herramienta, en este caso hablamos de compilar un programa. En SYCL, una manera sencilla de hacerlo es con un archivo *Makefile*. Como podemos observar en la **Figura 22**, tenemos definidas algunas banderas, por un lado, las de optimización (-O3), las de hipSYCL, las de NVIDIA y las de AMD. Después, las juntamos todas a la hora de compilar, aunque decidiendo entre NVIDIA o AMD. Esto es porque nosotros tenemos una máquina que tiene dos tipos de GPUs. No podemos compilar para las dos arquitecturas a la vez. En el caso de NVIDIA tenemos la bandera *--hipsycl-gpu-arch*, donde indicamos la arquitectura de la GPU de NVIDIA para la que queremos compilar. De nuevo, nosotros tenemos dos arquitecturas, *sm_35*, para cuando queramos usar cualquiera de las máquinas, y *sm_60*, para cuando solo queramos usar *Manticore*, ya que optimiza mejor el código. Por lo demás, el fichero es el típico *Makefile*: el comando *make all* nos compila todos los programas que hayamos añadido a OBJS y un *make "programa"* nos compila solo el programa concreto.

```
RELEASE_FLAGS=-O3
HIPSYCL_FLAGS=-lgomp -lstdc++ -L/frontend/sergmig/local/boost/lib/
NVIDIA_FLAGS=--hipsycl-gpu-arch=sm_35
AMD_FLAGS=--rocm-path=/opt/rocm-4.1.0 --hipsycl-platform=rocm --hipsycl-gpu-arch=gfx900 -lm
-L/frontend/sergmig/local/boost/lib/ -I/opt/rocm-4.1.0/hip/include/hip/ -L/opt/rocm-4.1.0/lib64
-L/opt/rocm-4.1.0/lib -I/frontend/sergmig/local/boost/include
-I/frontend/sergmig/local/hypSYCL2021/include

FLAGS=$(HIPSYCL_FLAGS) $(RELEASE_FLAGS) $(INCLUDE_FLAGS) $(NVIDIA_FLAGS)
OBJS=sumaVectores windsycl

all: $(OBJS)

sumaVectores: sumaVectores.cpp
    syclcc $(FLAGS) $< -o $@

windsycl: windsycl.cpp
    syclcc $(FLAGS) $< -o $@

clean:
    rm -rf $(OBJS)
```

Figura 22: *Makefile* de ejemplo para compilar programas SYCL

Para compilar un programa Celerity tenemos que crear un archivo *CMakelists.txt* como el de la **Figura 23**. En *HIPSYCL_TARGETS* indicamos la arquitectura para la que queremos compilar y en *target_compile_options()* podemos añadir banderas de compilación, como -O3. El programa que estamos compilando aquí es *sumaVectores.cpp*, por lo que tendríamos que crear un *CMakelists.txt* por programa, sustituyendo el nombre del programa por el que corresponda. Cada carpeta solo puede contener un *CMakelists.txt*.

```
cmake_minimum_required(VERSION 3.5.1)
project(celerity_sumaVectores)

set(Celerity_DIR /frontend/sergmig/local/celerity/lib/cmake)
set(HIPSYCL_TARGETS cuda:sm_35)

find_package(Celerity CONFIG REQUIRED)

add_executable(sumaVectores sumaVectores.cpp)
add_celerity_to_target(TARGET sumaVectores SOURCES sumaVectores.cpp)
target_compile_options(sumaVectores PRIVATE -O3)
```

Figura 23: *CMakelists.txt* de ejemplo para compilar programas Celerity

Para compilar el programa tendríamos que ejecutar los comandos *cmake* . y tras este, *make*. Cuando necesitemos recompilar porque hayamos cambiado nuestro programa, basta con ejecutar *make* de nuevo. Si cambiamos algún parámetro del *CMakelists.txt*, tendremos que ejecutar de nuevo los dos comandos.

4.5 Funcionamiento de Celerity

Celerity está construido sobre SYCL y utiliza MPI por debajo para la distribución del cómputo. Celerity utiliza un modelo de maestro/trabajador, donde el maestro es el que planifica como distribuir el trabajo y los trabajadores son los que reciben los comandos del maestro y ejecutan sus tareas en orden. Ya hemos explicado y mostrado que a pesar de que Celerity está utilizando MPI, no hay nada en el código que nos muestre que así sea, ya que en ningún momento decidimos cómo distribuir la carga o cómo y dónde realizar los envíos de datos. Pero, podemos ejecutar un programa Celerity con los comandos de ejecución de MPI (*mpirun*). Celerity está diseñado para que los programadores solo se esfuercen en pensar cómo crear los búferes y los *kernels*, y se olviden de todo lo relacionado con el paralelismo en memoria distribuida. Los *kernels*, también llamados grupos de comandos, pertenecen a una entidad mayor denominada tarea. Celerity utiliza las tareas para saber cómo gestionar la memoria distribuida.

Debido a este modelo de memoria distribuida, las tareas no pueden ser ejecutadas según son presentadas en el programa, ya que si no todos los nodos ejecutarían el mismo trabajo. En vez de esto, Celerity primero realiza una primera pasada (*prepass*) similar a la ejecución normal del programa. Cuando Celerity detecta una petición de encolado de una operación (*queue submit*), no pasa directamente a ejecutar esa función o *kernel*, sino que el código de la función junto con información sobre dependencias y metainformación de la planificación se graba en el grafo de tareas (*Task Graph*) de Celerity. Este grafo de tareas se construye sobre todo basándose en qué tipo de acceso se hace sobre los búferes y por tanto, que dependencias existen entre sus accesos. Con esta representación Celerity puede saber que tareas se tienen que ejecutar, en qué orden y cuales se pueden ejecutar en paralelo. La **Figura 24** muestra el grafo de tareas generado del programa de ejemplo de Celerity que explicamos anteriormente, el de la suma de vectores.

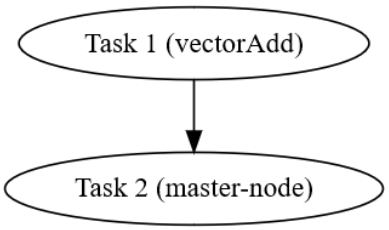

Figura 24: Grafo de tareas del programa de suma de vectores

En este caso tenemos dos tareas, la primera de ellas es la tarea correspondiente al *kernel* de suma de vectores y la segunda es la tarea que indica que hay una tarea que se hace en el nodo maestro, en el *host*, que es la impresión del vector resultado. La flecha negra que va desde la tarea 1 a la 2 indica que hay una dependencia entre las tareas y que por tanto, la tarea 2 no se puede ejecutar hasta que la tarea 1 haya terminado.

Continuando con el funcionamiento de Celerity, mientras se está ejecutando la primera pasada, un hilo de planificación (*Scheduler*) construye un grafo de comandos (*Command Graph*) más detallado a partir de los nodos del grafo de tareas. El grafo de comandos contiene las instrucciones individuales para cada nodo del sistema, codifica todas las trasferencias de datos (envíos y recepciones) para mantener una vista consistente de los datos, como si se estuviera ejecutando en un solo nodo. Cada nodo consume el grafo de comandos a medida que se van recibiendo las tareas ya en la ejecución real del programa (*livepass*).

Celerity solo soporta dos tipos de tareas, las tareas de cómputo, las más comunes, en las que se utiliza el controlador del grupo de comandos (*cgh*) con un *parallel_for* y las tareas del nodo maestro, en las que se solicita que esa tarea sea ejecutada solamente por el nodo maestro (por ejemplo, la impresión del resultado del cómputo), en las que se utiliza el controlador del grupo de comandos junto con un *host_task* y un *celerity::on_master_node*. Estas tareas tienen que ser ejecutadas por todos los nodos, por lo que no es posible tener un condicional que diga que si se cumple una condición se haga una tarea o se cree un búfer y si no se cumple la condición no se haga, a no ser que todos los nodos siempre bifurquen a la misma parte de la condición. Condiciones basadas por ejemplo en números aleatorios no son posibles. Actualmente Celerity tiene varias limitaciones, no soporta las reducciones de MPI y tampoco tiene un control del flujo del programa para tomar decisiones de bifurcación fuera del programa principal. Además, solo está implementada la operación *parallel_for* para expresar el paralelismo de datos.

El grafo de tareas se genera teniendo en cuenta que pueden darse tres tipos de dependencias [**43**]:

- Lectura después de escritura (*Read After Write*, RAW): también llamadas dependencias verdaderas, existen siempre que una instrucción depende del resultado de una instrucción anterior. Pueden surgir problemas por una instrucción que se ejecuta antes de que se completen sus dependencias.

- Escritura después de lectura (*Write After Read*, WAR): también llamadas anti-dependencias, se dan cuando una instrucción quiere escribir en una ubicación de memoria que primero debe ser leída por otra instrucción.
- Escritura después de escritura (*Write After Write*, WAW): también llamadas dependencias de salida, se dan cuando una instrucción quiere escribir en una ubicación de memoria en la que está escribiendo otra instrucción.

Celerity solo representa las dependencias RAW en sus grafos de tareas, mediante flechas negras. Además, Celerity trata las dependencias WAR y WAW de la misma forma, por lo que podemos definir anti-dependencia cómo la circunstancia que se da cuando una tarea quiere escribir en un búfer que, al mismo tiempo, puede estar todavía en uso (lectura o escritura) por otra tarea. Las anti-dependencias no se representan en el grafo de tareas, sino que solo se representan en el grafo de comandos.

El grafo de comandos se genera a partir del grafo de tareas, obteniendo las descripciones de las tareas y traduciéndolas a comandos que se envían y procesan por nodos diferentes. La **Figura 25** muestra el grafo de comandos del ejemplo de suma de vectores. Cada nodo del grafo de tareas está numerado con un número entre corchetes, el id de comando. En las tareas de cómputo, en el grafo de comandos se guarda el *chunk* que va a ser ejecutado en ese nodo. El *chunk* es la parte de cada estructura de datos con la que se computa que está asignada a esa tarea por la partición de datos entre procesos. Por el contrario, en las tareas del nodo maestro no se guarda el *chunk*, ya que en este tipo de tareas no existe la noción de *chunk*. Además, en cada nodo se guarda información de los tipos de acceso que se dan en esa tarea. Hay nodos especiales que no representan una tarea, sino que representan una trasferencia de datos, cómo un envío (*PUSH*) o una recepción (*AWAIT PUSH*). La flecha con una línea de puntos en el grafo de comandos indica el origen y el destino de la transferencia.

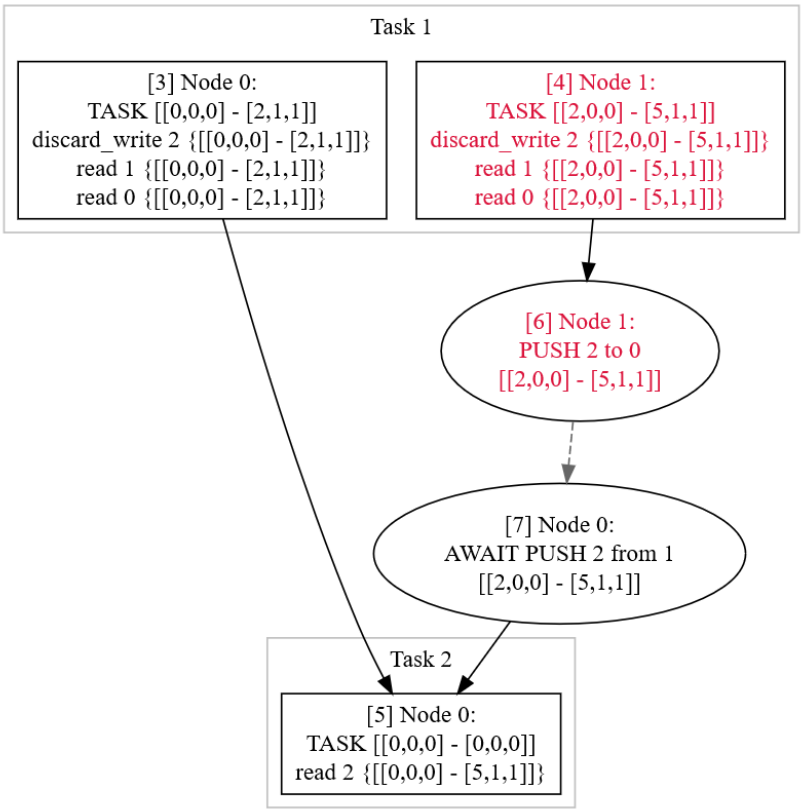

Figura 25: Grafo de comandos del programa de suma de vectores

La tarea 1 está formada por el comando *[3]*, que es la parte que se ejecuta en el nodo 0 y el comando *[4]*, para el nodo 1. Cada nodo tiene la palabra *TASK*, que indica que es una tarea, y la secuencia de números posterior es el *chunk* que maneja ese nodo. Vemos también los accesos que se están produciendo en esa tarea: *read 0* (lectura del vector A), *read 1* (lectura del vector B) y *discard_write 2* (escritura en C, búfer no inicializado), junto con los subrangos a los que accede. La segunda tarea, *[5]*, solo es ejecutada por el nodo maestro, pues es una tarea de este tipo, lo sabemos porque estas tareas no tienen la noción de *chunk* (esto es representado en la tarea del grafo de comandos como *TASK [[0,0,0] - [0,0,0]]*) y porque pone que solo la ejecuta el nodo 0. Esta tarea lee del vector C (*read 2*). Pero antes de poder leer de C, el nodo 1 tiene que pasar su parte del vector C al nodo 0. El nodo 1 realiza un envío (*PUSH*) de su parte del vector: *[6] PUSH 2* (vector C) *to 0* (nodo 0). A su vez el nodo 0 realiza una recepción (*AWAIT PUSH*): *[7] AWAIT PUSH 2* (vector C) *from 1* (nodo 1). De nuevo, los mismos tipos de dependencias que se nos daban con el grafo de tareas, se nos pueden dar en el grafo de comandos [**43**]:

- RAW: El rango válido del búfer existe en otro nodo, en cuyo caso se puede generar directamente una dependencia de un comando de ejecución en el *AWAIT_PUSH* correspondiente. Pero también puede ocurrir que el rango de búfer ya se ha recibido (a través de un *AWAIT_PUSH*) para otro comando, o que se produjo en el mismo nodo anteriormente. En ambos casos todavía tenemos que

asegurarnos de que existe una verdadera dependencia entre el comando fuente original y el comando de ejecución actual.

- WAR y WAW: las anti-dependencias se suelen dar en el mismo sitio que en el grafo de tareas, es decir, si un comando de ejecución escribe en un determinado subrango del búfer, primero tiene que esperar hasta que todos los comandos que leen de ese subrango en el mismo nodo estén terminados. Pero recibir el contenido del búfer de otro nodo (ejecutar un *AWAIT_PUSH*), también escribe en un búfer. Esto significa que en el grafo de comandos, pueden existir anti-dependencias incluso para tareas de solo lectura.

Las anti-dependencias se representan con una flecha verde en el grafo de comandos.

4.6 Mapeadores

Los mapeadores son los elementos que utiliza Celerity para determinar cómo se reparte el trabajo entre los procesos y cómo un *kernel* accede a un búfer distribuido. Tienen un papel vital en la generación del grafo de tareas y del grafo de comandos. En SYCL este elemento no existía, tan solo valía con establecer el modo de acceso (lectura, escritura,…), pero en Celerity no es suficiente, ya que este divide la ejecución del *kernel* entre los nodos, por lo que necesita saber cómo cada unidad de trabajo accede espacialmente a cada subbúfer. Un *chunk* es un trozo de la ejecución de un *kernel*. Los mapeadores mapean un *chunk* a un subrango del búfer al que está accediendo el *kernel*. La dimensionalidad de un *kernel* no tiene por qué coincidir con la del búfer al que accede. Una manera más fácil de entender los *chunks* es como un conjunto de hilos del *kernel*, denominado unidades de trabajo (*work ítems*). Cada unidad de trabajo se corresponde con un índice, que puede ser en 1, 2 o 3 dimensiones. Por tanto, si sabes cómo acceder a un búfer para cada unidad de trabajo, es fácil ampliarlo a un conjunto de unidades de trabajo. Celerity nos permite escribir nuestros propios mapeadores, siguiendo la siguiente función, que es la definición de mapeador:

(celerity::chunk<KernelDims>) -> celerity::subrange<BufferDims>

Implementar nuestro propio mapeador es una tarea complicada, tenemos que tener muy claro qué acceso queremos conseguir, y necesitamos saber, dado un *chunk*, formado por un desplazamiento (*offset*), un rango y un tamaño global, cuál es el desplazamiento y el rango del subrango de salida. Un ejemplo de mapeador creado a demanda le podemos encontrar en uno de los ejemplos de programas de Celerity. Este mapeador se puede utilizar para realizar una transposición a una matriz de 2 dimensiones [**44**]:

```
celerity::subrange<2> transposed(celerity::chunk<2> chnk) {
        assert(chnk.global_size[0] == chnk.global_size[1]);
        return
            {{chnk.offset[1], chnk.offset[0]},
            {chnk.range[1], chnk.range[0]},};}
```

En muchos de los casos no necesitaremos crear nuestro propio mapeador, por lo que Celerity nos proporciona 5 mapeadores diferentes, para las operaciones de distribución más habituales, los cuales vamos a ver a continuación.

4.6.1 One-to-one

El mapeador *one-to-one* es el mapeador más simple y común de los 5. Básicamente sería un *kernel* que accede a un búfer en el mismo índice que el índice de su unidad de trabajo. Hablando con *chunks*, mapea directamente el desplazamiento y el rango de un *chunk* a un subrango del búfer al que accede. La **Figura 26** muestra un ejemplo del comportamiento del mapeador *one-to-one*, a la izquierda aplicado a elementos únicos y la derecha aplicado a *chunks*. En todos los casos sería *one_to_one<2>()*. El borde negro es el *chunk* de entrada y el área coloreada el subrango de salida. Este mapeador solo se puede aplicar cuando el *kernel* y el búfer tienen las mismas dimensiones.

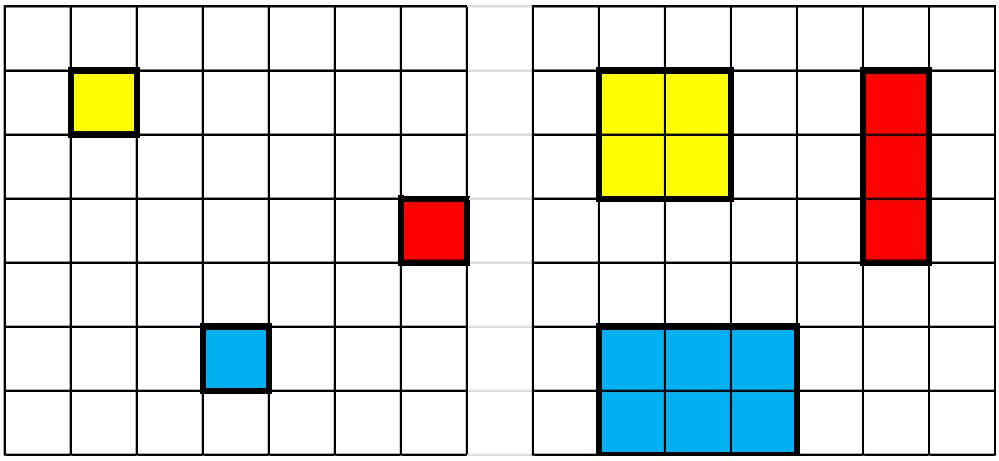

Figura 26: Comportamiento del mapeador *One-to-one*

4.6.2 Slice

El mapeador *slice* extiende las coordenadas de un *chunk* infinitamente en una dimensión dada, lo que resulta en un corte completo del búfer en esa dimensión. La **Figura 27** muestra varios ejemplos del comportamiento de este mapeador, a la izquierda aplicado a elementos únicos y a la derecha aplicado a *chunks*. El borde negro es el *chunk* de entrada y el área coloreada el subrango de salida. El ejemplo verde sería *slice<2>(1)* (extensión en el eje de las Y) y el azul *slice<2>(0)* (extensión en el eje de las X). Este mapeador solo se puede aplicar cuando el *kernel* y el búfer tienen las mismas dimensiones y las dimensiones de ambos son mayores a 1. Se suele usar en multiplicación de matrices.

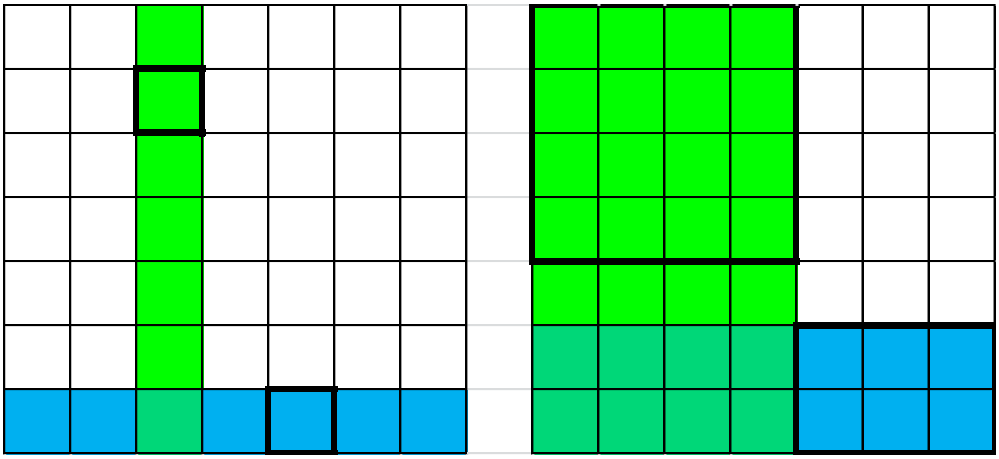

Figura 27: Comportamiento del mapeador *Slice*

4.6.3 Neighborhood

El mapeador *neighborhood* permite seleccionar un número de elementos alrededor del elemento concreto en cada dimensión. La **Figura 28** muestra varios ejemplos del comportamiento de este mapeador, a la izquierda aplicado a elementos únicos y a la derecha aplicado a *chunks*. El borde negro es el *chunk* de entrada y el área coloreada el subrango de salida. El ejemplo rojo sería *neighborhood<2>(1,1)*, el verde sería *neighborhood<2>(0,1)* y el azul *neighborhood<2>(2,0)*. Este mapeador solo se puede aplicar cuando el *kernel* y el búfer tienen las mismas dimensiones. Se suele usar en los códigos de tipo Stencil.

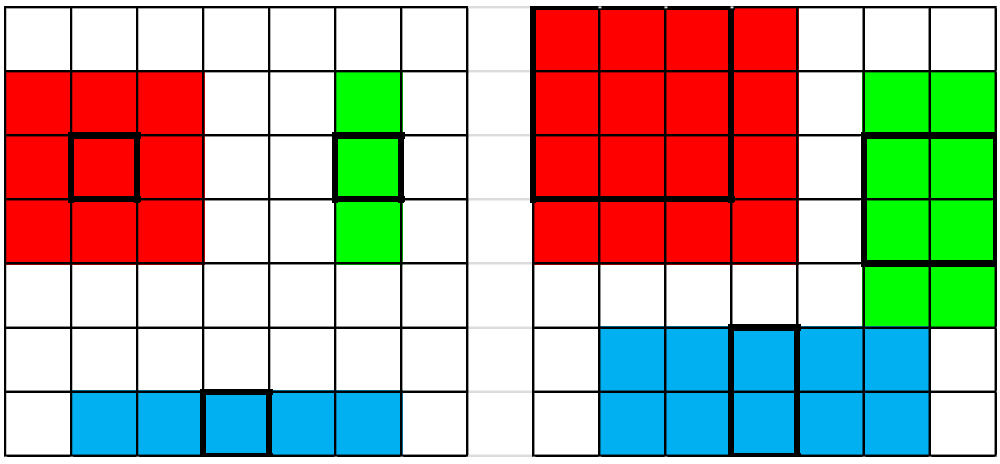

Figura 28: Comportamiento del mapeador *Neighborhood*

4.6.4 Fixed

El mapeador *fixed* permite seleccionar un subrango específico común para todos los *chunks*. La **Figura 29** muestra varios ejemplos del comportamiento de este mapeador, a la izquierda aplicado a elementos únicos y a la derecha aplicado a *chunks*. El borde negro es el *chunk* de entrada y el área coloreada el subrango de salida. El ejemplo rojo sería *fixed<2>(subrange({4,0},{3,3}))* y el amarillo *fixed<2>(subrange({1,4},{5,3}))*. Este

mapeador se suele usar en las convoluciones o en situaciones en las que todos los nodos tienen que acceder a la misma sección del búfer. En escritura, este mapeador puede traer problemas. La salida de los mapeadores no debe solaparse. Si dos procesos escriben sobre el mismo subrango de datos a la vez, no existe una regla que resuelva este problema de concurrencia, lo que llevaría a una condición de carrera.

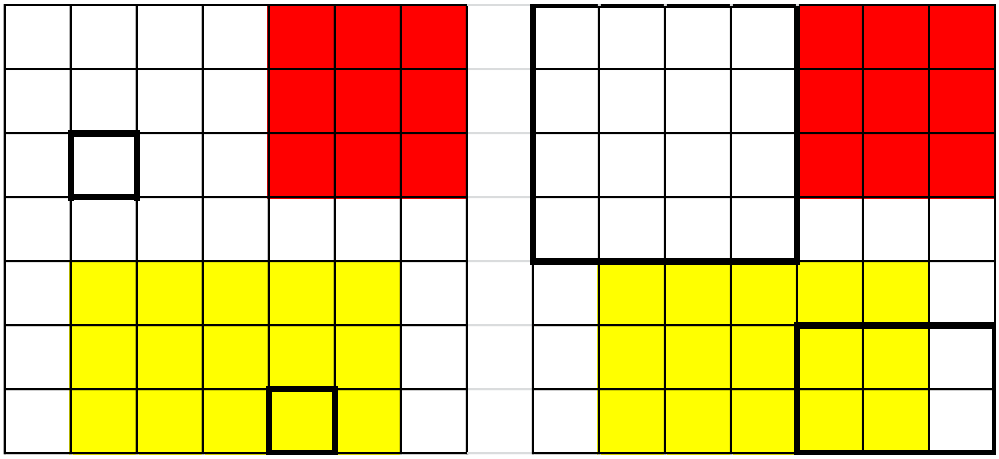

Figura 29: Comportamiento del mapeador *Fixed*

4.6.5 All

El último de los 5 mapeadores predefinidos por Celerity, el mapeador *all*, permite seleccionar el búfer entero, sin importar el *chunk* de entrada. Es un caso especial del mapeador *fixed*. El uso habitual de este mapeador es en las operaciones que se realizan en el nodo maestro, por ejemplo, al finalizar el programa e imprimir los resultados, todos los datos se suelen traer al nodo maestro para que sea él el encargado de imprimirlos. Para indicar que este nodo va a acceder a todo el búfer usamos este mapeador, *all<dim>()*.

5

Estudio experimental

5.1 Introducción

Una vez explicado cómo funciona Celerity por debajo y uno de sus principales conceptos, los mapeadores, estamos en disposición de realizar nuestro estudio experimental. Partimos de la hipótesis de que un programa ejecutado en un clúster de aceleración es tan eficiente si se programa y ejecuta con Celerity a si lo hace utilizando herramientas con el mismo propósito, con menor abstracción, pero más flexibles o adaptables. Vamos a compararlo con la herramienta Controller+Hitmap, que ofrece un punto medio de abstracción entre CUDA+MPI y Celerity, con unos rendimientos similares a la solución de más bajo nivel. Celerity tiene distintos tipos de programas en su banco de pruebas [**45**] y nosotros hemos seleccionado los programas del tipo Stencil. Son una clase de programas con múltiples aplicaciones reales y que presentan un tipo de comunicación donde es necesario resolver varios problemas de reparto y comunicación clásicos y generalizables a otros tipos de aplicaciones. Son bancos de pruebas (*benchmarks*) típicos para los desarrolladores de lenguajes de programación y *runtimes* paralelos. Se puede disponer fácilmente de muchas variantes con distintas cargas de trabajo y de balanceo de carga. Los programas Stencil son una clase de procesamiento de datos numéricos que actualizan los elementos de un vector o array de 1, 2 o 3 dimensiones en base a un determinado patrón. Este patrón es llamado Stencil. Los programas Stencil se usan en simulaciones de fluidos dinámicos, para resolver ecuaciones por el método de Jacobi o Gauss-Seidel, para el procesamiento de imágenes o para autómatas celulares. Un famoso ejemplo de autómata celular es el juego de la vida de Conway (*Conway's Game of Life*).

En la primera parte del capítulo vamos mostrar los tipos de Stencil que soporta nuestro programa, además del flujo del mismo. A continuación, se describirá como está programada nuestra solución utilizando Celerity y se explicarán sus funciones más importantes. Por último, se realizará un diseño de las pruebas que vamos a hacer con nuestro programa, de cara a obtener el tiempo de cómputo con diferentes cargas de trabajo y distinto número de nodos de ejecución. Con los resultados de las pruebas, haremos una comparativa con los tiempos obtenidos por programas similares pero utilizando Controller+Hitmap, que se proveen como ejemplos en su distribución actual.

5.2 Visión global del programa

Nuestro programa es muy sencillo conceptualmente. Comienza con la lectura de los argumentos introducidos por el usuario. A continuación, se comprueban que los argumentos sean válidos. Si no lo son, se muestra un mensaje de error y la ejecución del programa termina. Cuando los argumentos son correctos, comienza el entorno de Celerity, empezando por un *kernel* que inicializa los búferes con los datos. Al acabar esta inicialización, establecemos un punto de sincronización, una barrera, para poner en marcha un contador de tiempo. Después, pasamos a la parte de cómputo, compuesta por una iteración (pasos de tiempo de simulación) con un *kernel* en la que se computan resultados en un array a partir de valores de otro. En cada iteración se invierte el papel de los dos arrays con un intercambio de punteros entre los dos búferes. El *kernel* de cómputo varía dependiendo del tipo de Stencil. Cuando acaba el bucle, volvemos a tener otra barrera y el contador de tiempo termina, lo que nos proporciona el tiempo de la fase de cómputo. Por último, imprimimos el array resultante. Para nuestra experimentación, esta última fase de impresión solo está activa cuando queramos comprobar si el resultado es correcto, normalmente durante la fase de desarrollo del programa. En las ejecuciones experimentales está desactivada, ya que la impresión de un array de un tamaño muy grande hace que tarde mucho en ejecutar el programa, y lo que a nosotros nos interesa es el tiempo que transcurre en la fase de cómputo. El resultado ya hemos comprobado previamente que es correcto. La **Figura 30** muestra el diagrama de flujo del programa.

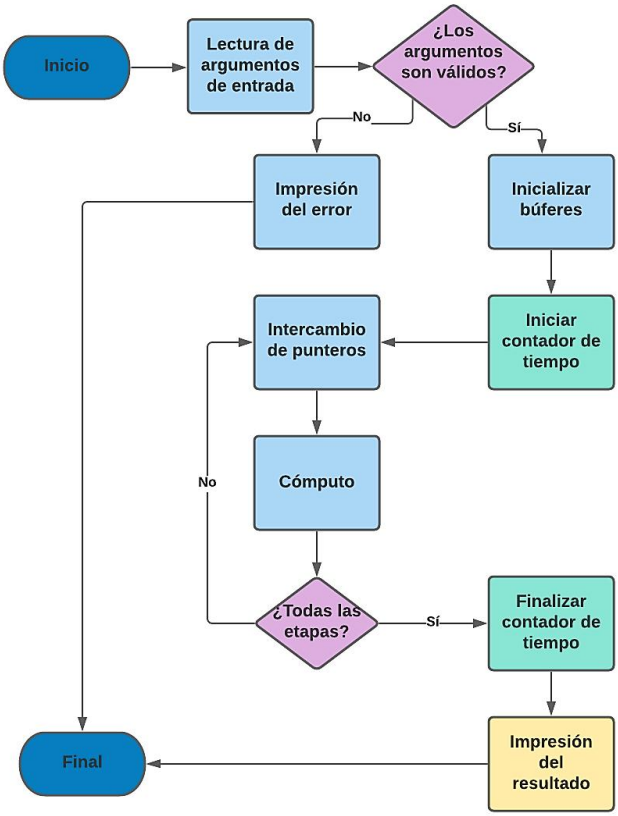

Figura 30: Diagrama de flujo del programa

El programa que hemos desarrollado soporta Stencil de 1, 2 y 3 dimensiones. El Stencil más famoso de los que hemos implementado es el llamado Jacobi (Stencil 2D de 4 puntos compacto), donde cada elemento de la matriz se actualiza con la media aritmética de las posiciones de arriba, abajo, izquierda y derecha del elemento. Esta operación se realiza con todos los elementos de la matriz, menos los bordes. La lista completa de Stencil soportados es la siguiente:

- 1D de 2 puntos compacto.
- 1D de 4 puntos compacto.
- 2D de 4 puntos compacto (Jacobi).
- 2D de 9 puntos compacto.
- 2D de 9 puntos no compacto.
- 2D de 5 puntos no compacto asimétrico.
- 3D de 27 puntos compacto.

En la **Figura 31** podemos ver una representación gráfica de los tipos de Stencil implementados en nuestro programa. Este conjunto cubre las diferentes situaciones o patrones de comunicación que puede generar este tipo de aplicación: Stencils compactos vs no compactos, en 2D vecinos con espacios vs sin espacios y simétricos vs asimétricos unidireccionales.

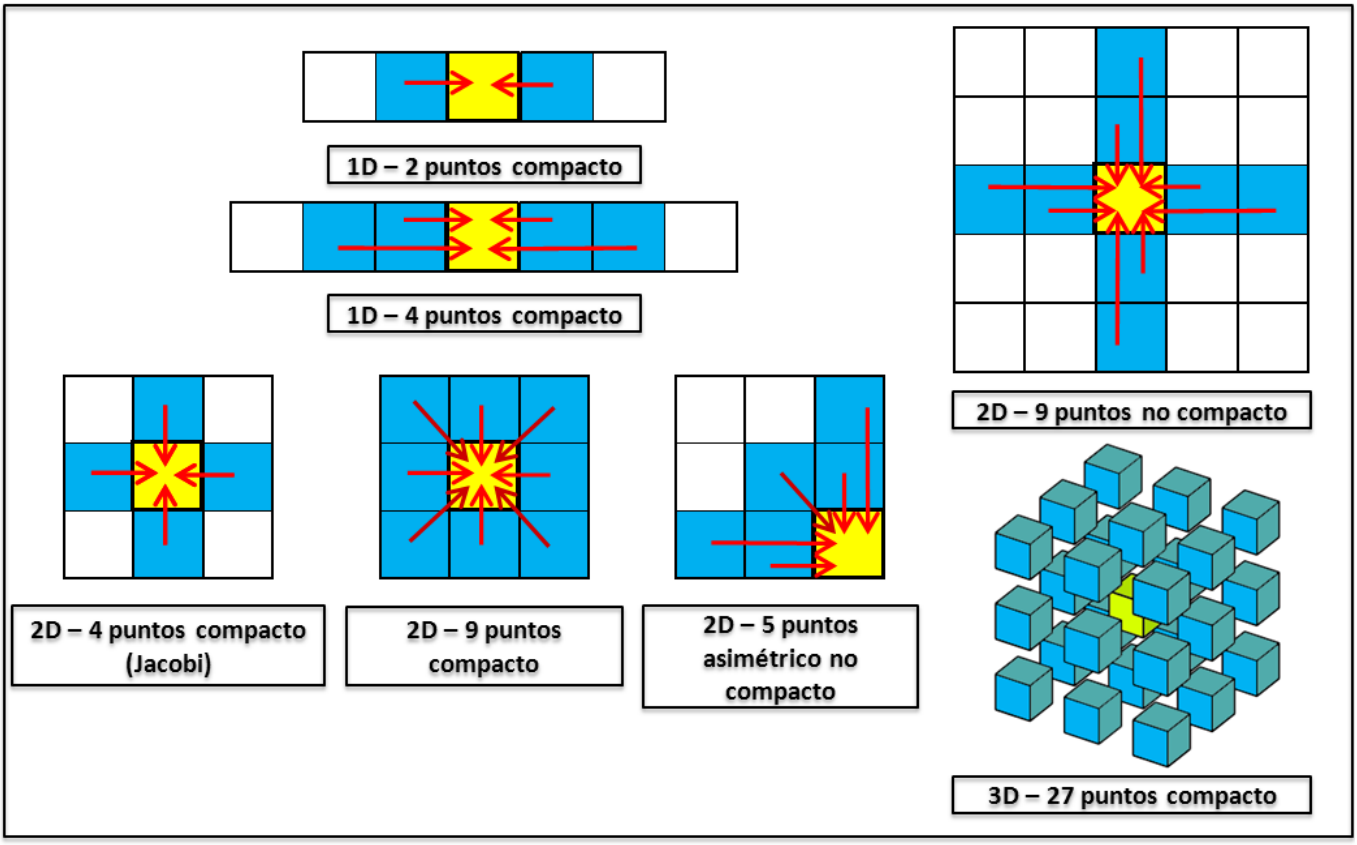

Figura 31: Tipos de Stencil soportados por nuestro programa

5.3 Desarrollo del programa

Nuestro programa comienza con la lectura de argumentos, para ello utilizamos la función *getopt()*, que nos permite introducir y analizar los argumentos de una forma estándar. Un ejemplo de entrada sería: *./stencil -r 10 -c 10 -d 10 -s 25 -t 1*. Esta línea indica que vamos a usar una matriz 3D de 10 filas (opción **-r** (*rows*)), 10 columnas (opción **-c** (*columns*)) y 10 de profundidad (opción **-d** (*depth*)), serán 25 iteraciones de cómputo (opción **-s** (*stages*)) y el tipo de Stencil es el 1 (opción **-t** (*type*)). El programa autodetecta el número de dimensiones dependiendo de las opciones de entrada, si solo detecta **-r**, es 1D, **-r** y **-c**, 2D y **-r**, **-c** y **-d**, 3 dimensiones. Para poder ver qué argumentos existen y el tipo asociado a cada Stencil para la opción **-t**, tenemos una última opción, **-h** (*help*), que imprime un manual de ayuda del programa, en el que destaca los tipos de Stencil que maneja el programa. Introducir la opción **-h** ignora los demás argumentos e imprime la ayuda solamente. El programa también detecta configuraciones que no se puedan dar, como introducir solamente filas y profundidad, en estos casos se imprime un mensaje de error. A continuación, entramos en una u otra parte de un switch que depende del número de dimensiones. Ahora es cuando empieza la parte de Celerity. La **Figura 32** muestra una de las tres opciones del switch, cuando es una matriz de 2 dimensiones. Para la explicación del programa usaremos matrices de 2 dimensiones. El código en los otros casos es muy similar.

```
auto tam = cl::sycl::range<2>(rows, columns);
celerity::buffer<float,2> buff_mat(tam);
celerity::buffer<float,2> buff_copy(tam);

// Init
initialization(q,buff_mat,buff_copy,rows,columns,depth,type);

//Start timer
q.slow_full_sync();
celerity::experimental::bench::begin("compute");
auto start = std::chrono::high_resolution_clock::now();

// Compute
int stage;
for ( stage=0; stage<stages; stage++ ) {
        //Update copy
        std::swap(buff_mat, buff_copy);

        // Compute iteration
        compute2D(q,buff_mat,buff_copy,rows,columns,type);
}
```

Figura 32: Código Stencil - Inicialización y bucle de cómputo

Primero creamos dos búferes con el tamaño que se ha introducido como entrada del programa. Antes de esta parte ya se ha creado la cola y se ha inicializado el entorno de ejecución de Celerity, por lo que podemos llamar a la función de inicialización de los búferes. Más adelante veremos esta función en detalle, pero por ahora nos vale con saber

que al acabar, tenemos nuestros dos búferes inicializados de la misma forma, es decir, *buff_mat* y *buff_copy* tienen los datos iniciales y son iguales. A continuación, tenemos una llamada a *slow_full_sync()*. Esta función sincroniza todos los nodos del clúster, es una barrera. De esta manera podemos empezar un contador de tiempo en todos los procesos. El contador de tiempo lo hacemos de dos maneras diferentes, mediante un evento de Celerity y mediante la librería Chrono [**46**]. Tenemos dos formas para probar posibles diferencias entre los dos métodos de medida de tiempo que se usan habitualmente en los ejemplos de Celerity. Lo siguiente es el bucle principal en el que en cada etapa se hace un intercambio de punteros entre los dos búferes y una llamada a la función de cómputo. La función de cómputo aplica el patrón Stencil a la matriz, pero no podemos usar una sola matriz, ya que si leemos y escribimos en la misma matriz, puede provocar resultados inconsistentes, debido a que hay hilos que tardan más o menos y puede ser que sobrescriba una posición que no ha sido leída todavía. Por esto tenemos dos búferes, de *buff_copy* leemos y en *buff_mat* escribimos, y en la siguiente iteración cambiamos los punteros, lo que significa que ahora *buff_copy* es *buff_mat* y viceversa. Y así todas las iteraciones. Cuando acaba el bucle, el resultado está en *buff_mat*. Utilizar la función *swap* de intercambio de punteros nos ahorra mucho tiempo de ejecución, la otra opción menos eficiente es un *kernel* Celerity de copia de *buff_mat* a *buff_copy*. Ya solo nos queda ver cuánto tiempo ha tardado en ejecutarse e imprimir el resultado, lo vemos en la **Figura 33**.

```
//End timer
q.slow_full_sync();
auto end = std::chrono::high_resolution_clock::now();
celerity::experimental::bench::end("compute");

int time = std::chrono::duration_cast<std::chrono::milliseconds>(end-start).count();
printf("\nTime: %8d ms\n\n",time);

// Write
q.submit(celerity::allow_by_ref,[&](celerity::handler& cgh) {
        auto result = buff_mat.get_access<cl::sycl::access::mode::read,
                cl::sycl::access::target::host_buffer>(cgh, celerity::access::all<2>());
        cgh.host_task(celerity::on_master_node, [=]() {
                for ( size_t i=0; i<rows; i++ ){
                        for ( size_t j=0; j<columns; j++ ){
                                printf("%014.4lf ", result[{i,j}]);
                        }
                        printf("\n");
                }
        });
});
```

Figura 33: Código Stencil - Impresión del tiempo de cómputo y del resultado

En este punto volvemos a sincronizar el clúster y medimos el tiempo de nuevo. Utilizar el evento de Celerity imprime directamente el tiempo que ha tardado cada proceso. La otra manera vuelve a capturar el instante de tiempo actual, y lo resta del instante de tiempo obtenido anteriormente y el resultado de la diferencia es el tiempo que ha pasado. Los resultados de los tiempos son muy similares de ambas maneras. Por último, tenemos una función que nos imprime la matriz resultado, pero esto lo tiene que

hacer el proceso maestro, por lo que utilizamos una *host_task*. Por supuesto, por debajo Celerity está enviando todos los trozos de matriz que tienen otros procesos al proceso 0 para que este pueda imprimir toda la matriz.

La parte de imprimir la matriz resultado solo se activa si el programa se usa en producción o cuando queremos comprobar si el resultado es correcto, comparándolo por ejemplo con el mismo desarrollado sin Celerity, solo con C++, lo que se conoce habitualmente como programa en secuencial, ya que no hay partes en paralelo. El tiempo que obtenemos es el tiempo de ejecución de la fase de cómputo. Toda la parte de lectura de argumentos y el *kernel* de inicialización no se tienen en cuenta a la hora de contar el tiempo.

Una vez explicado el programa principal, vamos a ver los dos *kernels* que hemos programado. La **Figura 34** muestra el *kernel* de inicialización de los búferes.

```
template <typename T, int dim>
void initialization(celerity::distr_queue& q, celerity::buffer<T,dim> mat,
  celerity::buffer<T,dim> copy,  long rows, int columns, int depth, int type){

  cl::sycl::range<dim> tam(mat.get_range());

  q.submit([=](celerity::handler& cgh){
    auto one_to_one=celerity::access::one_to_one<dim>();
    celerity::accessor writemat(mat, cgh, one_to_one, cl::sycl::write_only, cl::sycl::no_init);
    celerity::accessor writecopy(copy, cgh, one_to_one, cl::sycl::write_only, cl::sycl::no_init);
    cgh.parallel_for<class matInit>(tam,[=](cl::sycl::id<dim> tid){
      if(dim==2){
        int row=tid[0];
        int col=tid[1];
        if(row==0){ writemat[tid]=1;     writecopy[tid]=1;}
        else if(row==rows-1){ writemat[tid]=2;  writecopy[tid]=2;}
        else if(col==0){ writemat[tid]=3;       writecopy[tid]=3;}
        else if(col==columns-1){ writemat[tid]=4;       writecopy[tid]=4;}
        else{ writemat[tid]=0.0;        writecopy[tid]=0.0;}
```

Figura 34: Código Stencil - *Kernel* de inicialización de los búferes

Este *kernel* está diseñado mediante plantillas (*template*), de manera que los búferes pueden ser de cualquier tipo numérico (*int, long, float, double*,…) y de cualquier dimensión. Creamos dos *accessors*, uno para cada búfer, los dos solicitando el modo escritura y diciendo que los datos no están inicializados (*no_init*). Además, en los dos *accessors* usamos el mapeador *one-to-one*, que recordamos que indicaba que cada índice accede al mismo índice de su unidad de trabajo. En **Figura 34** la solo aparece el caso en que tenemos 2 dimensiones, hemos cortado los otros dos casos ya que son similares. En nuestro programa los búferes se inicializan de forma que la primera fila se rellena con 1, la última con 2, la primera columna con 3 y la última columna con 4. El resto de posiciones con 0. Esto es un ejemplo típico para las pruebas del Stencil tipo Jacobi para calcular la difusión de calor y que se puede generalizar a cualquier Stencil sin afectar al rendimiento.

El siguiente *kernel* es el más importante, pues obtener el menor tiempo posible de cómputo depende de lo eficiente que sea. La **Figura 35** muestra el *kernel* de cómputo, en concreto para el tipo de Stencil de 2D, 4 puntos compacto, el también denominado Jacobi

(llamado así porque resuelve la ecuación de Poisson para la difusión del calor usando el método iterativo de Jacobi).

```
template <typename T>
void compute2D1(celerity::distr_queue& q, celerity::buffer<T,2> mat,
  celerity::buffer<T,2> copy, long rows, int columns){

  cl::sycl::range<2> tam(mat.get_range());

  q.submit([=](celerity::handler& cgh){

    auto one_to_one=celerity::access::one_to_one<2>();
    auto neighborhood=celerity::access::neighborhood<2>(1,1);
    celerity::accessor readcopy(copy, cgh, neighborhood, cl::sycl::read_only);
    celerity::accessor writemat(mat, cgh, one_to_one, cl::sycl::write_only);

    cgh.parallel_for<class matCompute>(tam,[=](cl::sycl::id<2> tid){
      if(tid[0]!=0 && tid[0]!=rows-1 && tid[1]!=0 && tid[1]!=columns-1){
        writemat[tid]=(readcopy[{tid[0]-1,tid[1]}] + readcopy[{tid[0]+1,tid[1]}]
          + readcopy[{tid[0],tid[1]-1}] + readcopy[{tid[0],tid[1]+1}]) /4;
      }
    });
  });
}
```

Figura 35: Código Stencil - *Kernel* de cómputo para matriz 2D y 4 puntos compacto (Jacobi)

En este *kernel* accedemos al búfer *buff_mat* en modo escritura con el mapeador *one-to-one*. Al búfer *buff_copy* accedemos en modo lectura, pero con el mapeador *neighborhood<2>(1,1)*, lo que indica que Celerity sabe que vamos a leer de las posiciones a 1 vecino de distancia de la que estamos. Este mapeador es necesario en este caso ya que se da la circunstancia de que en algún caso, la posición de arriba o de abajo la tiene otro proceso, por lo que Celerity tiene que enviar la fila o columna que la contiene al proceso correcto. Las filas y columnas de los bordes de cada proceso que se envían y que comparten varios procesos se denominan halos. Ya en el *kernel*, un condicional evita que se calculen los bordes y las demás posiciones se calculan en base a la fórmula del Stencil que se haya elegido.

Tenemos programados múltiples *kernels* de cómputo, uno por cada dimensión. Estos tres *kernels* aceptan como argumento el tipo de Stencil y con un condicional ejecutan el tipo de Stencil correcto. El problema es que, por ejemplo, con 2 dimensiones tenemos cuatro tipos de Stencil diferentes. Sin embargo, añadir condiciones dentro de lo que es el *kernel* (lo que está dentro del *cgh.parallel_for*), hace que tarde más en ejecutarse. Para solucionarlo, además de la función global de cada dimensión, creamos una función distinta para cada tipo de Stencil, evitándonos así escribir el condicional del tipo. Igualmente tenemos un condicional, como hemos visto en la **Figura 35**, que hace que los bordes no se ejecuten. Pero hemos encontrado una manera de evitarnos ejecutar también este condicional, el código del nuevo *kernel* le vemos en la **Figura 36**:

```
cgh.parallel_for<class matCompute>(cl::sycl::range<2>(rows-2,columns-2),
        cl::sycl::id<2>(1,1),[=](cl::sycl::item<2> tid){

   writemat[tid]=(readcopy[{tid[0]-1,tid[1]}] + readcopy[{tid[0]+1,tid[1]}] +
         readcopy[{tid[0],tid[1]-1}] + readcopy[{tid[0],tid[1]+1}]) /4;
});
```

Figura 36: Código Stencil - *Kernel* de cómputo para matriz 2D y 4 puntos compacto (Jacobi)

Utilizamos un rango que va hasta *rows-2* y *columns-2*, es decir, que estamos eliminando el borde de abajo y de la derecha de la matriz y además un borde interior que sí que tendríamos que calcular. Pero para coger esta última parte de la matriz, utilizamos *id(1,1)*, que es un desplazamiento del rango seleccionado, en este caso desplazamiento de 1 fila hacia la abajo y de una columna hacia la derecha, lo que hace que el rango ahora ya evite los bordes y solo se calcule la parte de la matriz que deseamos. Esto reduce notablemente el tiempo de cómputo.

El código completo del programa se puede ver en el **Anexo V: Código del programa Stencil**. La **Figura 37** muestra el grafo de tareas generado por Celerity de nuestro programa, para una ejecución con 2 GPUs, de una matriz 2D de 32000 filas por 32000 columnas, el tipo de Stencil es Jacobi y 2 iteraciones. También está activada la parte de impresión del resultado. Además, la **Figura 38** muestra el grafo de comandos para la misma ejecución, las flechas verdes son las anti-dependencias.

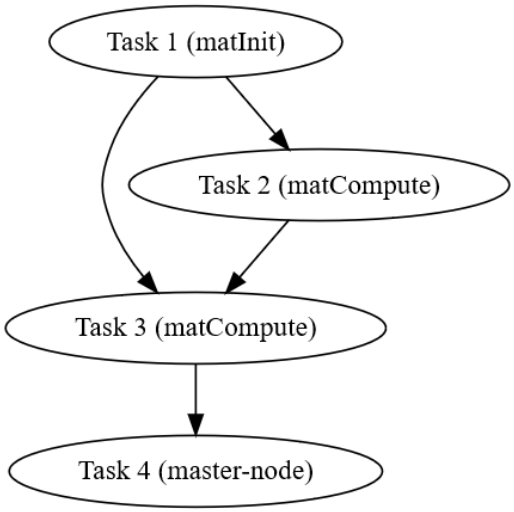

Figura 37: Grafo de tareas del programa Stencil, 32000x32000, Jacobi, 2 iteraciones

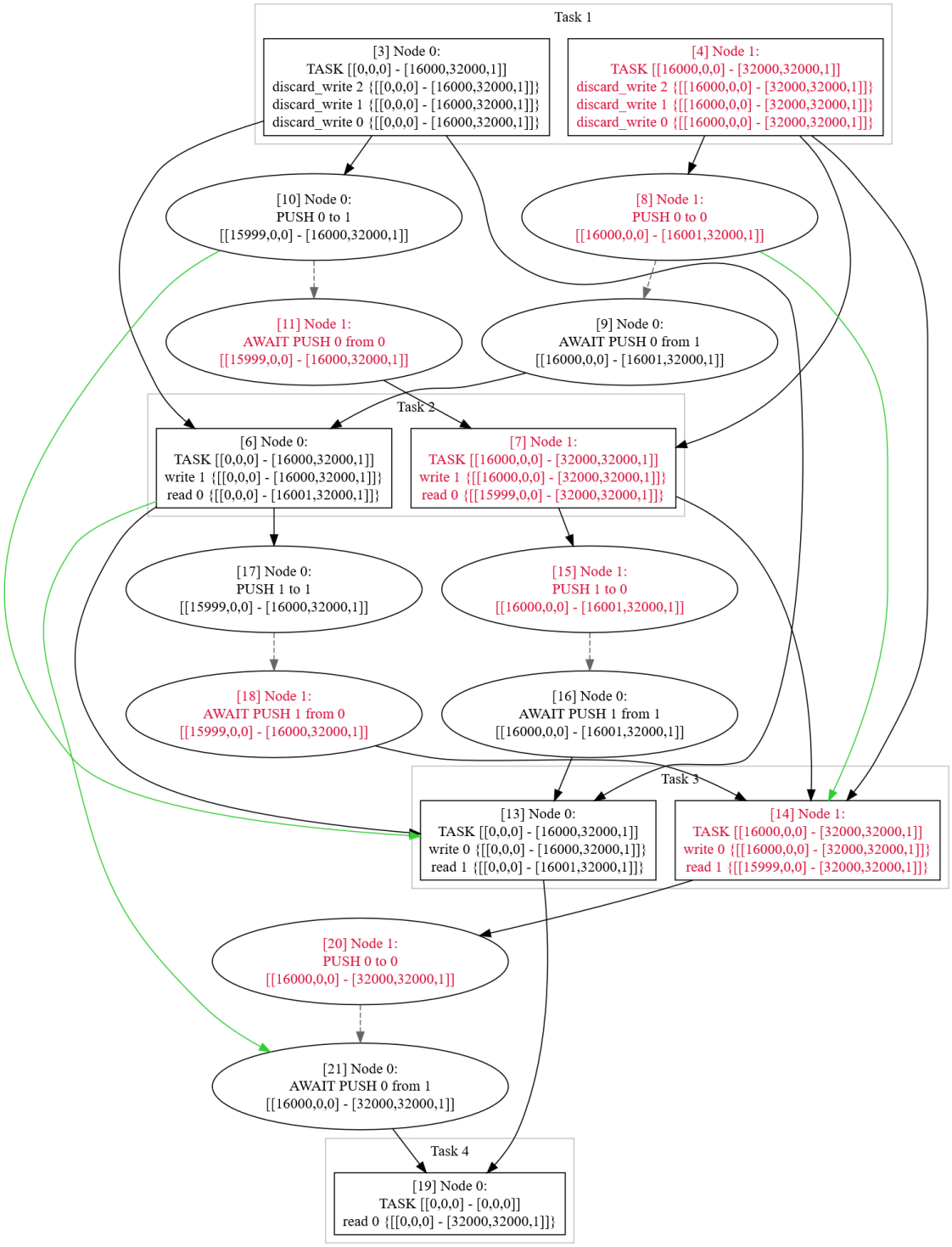

Figura 38: Grafo de comandos del programa Stencil, 32000x32000, Jacobi, 2 iteraciones

5.4 Experimentación

Una vez que tenemos el programa hecho, vamos a pasar a la parte de la experimentación, donde nuestro objetivo será validar o rechazar la hipótesis de que un

programa ejecutado en un clúster de aceleración es al menos tan eficiente si se programa y ejecuta con Celerity cómo si se hace utilizando Controller+Hitmap.

Por cada tipo de Stencil, vamos a hacer dos tipos de pruebas: las de escalabilidad fuerte, donde los vectores o matrices siempre tienen el mismo tamaño independientemente del número de GPUs con las que ejecutemos el programa, y las de escalabilidad débil, donde los vectores o matrices aumentan en tamaño proporcionalmente al número de GPUs con las que ejecutemos el programa. Las pruebas se harán primero en la maquina Manticore, la que cuenta con las GPUs más rápidas y potentes, pero solo tenemos 2 de ellas, modelo NVIDIA Tesla V100-PCIE-32GB. Utilizaremos también Medusa, formada por 3 GPUs NVIDIA GeForce GTX TITAN Black y 1 GPU NVIDIA GeForce GTX TITAN X, esta última es mejor que las TITAN Black. Por último, Hydra dispone de 2 GPUs NVIDIA GeForce GTX TITAN Black y 2 GPUs NVIDIA Tesla K40c, estas 2 últimas son las GPUs más antiguas. Además de las pruebas que realicemos con cada máquina independiente, haremos pruebas también en las que se combinen dos máquinas, con los consiguientes sobrecostes por las comunicaciones a través de la red. Cada prueba se repite 5 veces y se anota el tiempo que ha tardado en ejecutarse la parte de cómputo del programa, en segundos. Hemos elegido estos tamaños ya que son los tamaños máximos que podemos utilizar en nuestro clúster con Celerity. Cada prueba se repite solo 5 veces ya que los resultados son muy estables por lo que no son necesarias más repeticiones. Más adelante se mostrará un test de hipótesis para verificar que los resultados son estadísticamente significativos con solo 5 repeticiones. La **Figura 39** muestra los tamaños de los vectores y matrices de cada prueba por cada tipo de escalabilidad y por cada número de GPUs y máquina. En Medusa, X significa que se usa la TITAN X y B que se usa la TITAN Black. Se hacen siempre 1000 iteraciones (opción del programa **-s** 1000), que produce unos tiempos de ejecución suficientemente estables, donde pequeñas irregularidades o efectos estocásticos en una iteración se compensan y se hacen irrelevantes en el tiempo de cómputo total.

<table>
<tr><th colspan="7">Tamaños de los vectores/matrices de la experimentación</th></tr>
<tr><th rowspan="2">GPU</th><th colspan="2">1D</th><th colspan="2">2D</th><th colspan="2">3D</th></tr>
<tr><th>E. débil</th><th>E. fuerte</th><th>E. débil</th><th>E. fuerte</th><th>E. débil</th><th>E. fuerte</th></tr>
<tr><td>1 Manticore</td><td>1 024 000 000</td><td rowspan="2">1 024 000 000</td><td>32 000x32 000</td><td rowspan="2">32 000x32 000</td><td>1 000x1 000x1 000</td><td rowspan="2">1 000x1 000x1 000</td></tr>
<tr><td>2 Manticore</td><td>2 048 000 000</td><td>64 000x32 000</td><td>2 000x1 000x1 000</td></tr>
<tr><td>1 Medusa X</td><td>324 000 000</td><td rowspan="13">324 000 000</td><td>18 000x18 000</td><td rowspan="13">18 000x18 000</td><td>700x700x700</td><td rowspan="13">700x700x700</td></tr>
<tr><td>1 Medusa B</td><td>324 000 000</td><td>18 000x18 000</td><td>700x700x700</td></tr>
<tr><td>2 Medusa XB</td><td>648 000 000</td><td>36 000x18 000</td><td>1 400x700x700</td></tr>
<tr><td>2 Medusa BB</td><td>648 000 000</td><td>36 000x18 000</td><td>1 400x700x700</td></tr>
<tr><td>3 Medusa XBB</td><td>972 000 000</td><td>54 000x18 000</td><td>2 100x700x700</td></tr>
<tr><td>3 Medusa BBB</td><td>972 000 000</td><td>54 000x18 000</td><td>2 100x700x700</td></tr>
<tr><td>4 Medusa</td><td>1 296 000 000</td><td>72 000x18 000</td><td>2 800x700x700</td></tr>
<tr><td>1 Hydra</td><td>324 000 000</td><td>18 000x18 000</td><td>700x700x700</td></tr>
<tr><td>2 Hydra</td><td>648 000 000</td><td>36 000x18 000</td><td>1 400x700x700</td></tr>
<tr><td>3 Hydra</td><td>972 000 000</td><td>54 000x18 000</td><td>2 100x700x700</td></tr>
<tr><td>4 Hydra</td><td>1 296 000 000</td><td>72 000x18 000</td><td>2 800x700x700</td></tr>
<tr><td>4 (2) Manticore + (2) Medusa</td><td>1 296 000 000</td><td>72 000x18 000</td><td>2 800x700x700</td></tr>
<tr><td>8 (4) Hydra + (4) Medusa</td><td>2 592 000 000</td><td>144 000x18 000</td><td>5 600x700x700</td></tr>
</table>

Figura 39: Tamaños de los vectores y matrices para las pruebas de escalabilidad fuerte y débil

La versión del programa en Controller+Hitmap ha sido desarrollada por el grupo Trasgo, y serán ellos quién nos proporcionen los resultados de las pruebas ejecutadas con su herramienta. Empezando por el tipo de Stencil 2D 4 puntos compacto (Jacobi), la **Figura 40** muestra una gráfica con la media los tiempos de las 5 ejecuciones de cada prueba, en segundos, de todas las pruebas de escalabilidad fuerte realizadas con Celerity y con Controller+Hitmap. Todas las pruebas realizadas con Controller+Hitmap han sido realizadas solo en Medusa y Manticore, por problemas de configuración del control de afinidades de procesos a nodos NUMA en Controller no ha sido posible disponer de Hydra a tiempo de los resultados. Las tablas completas de este tipo de Stencil y de los demás tipos se encuentran en el **Anexo VI: Tablas de resultados Celerity** para Celerity y en el **Anexo VII: Tablas de resultados Controller+Hitmap** para Controller+Hitmap.

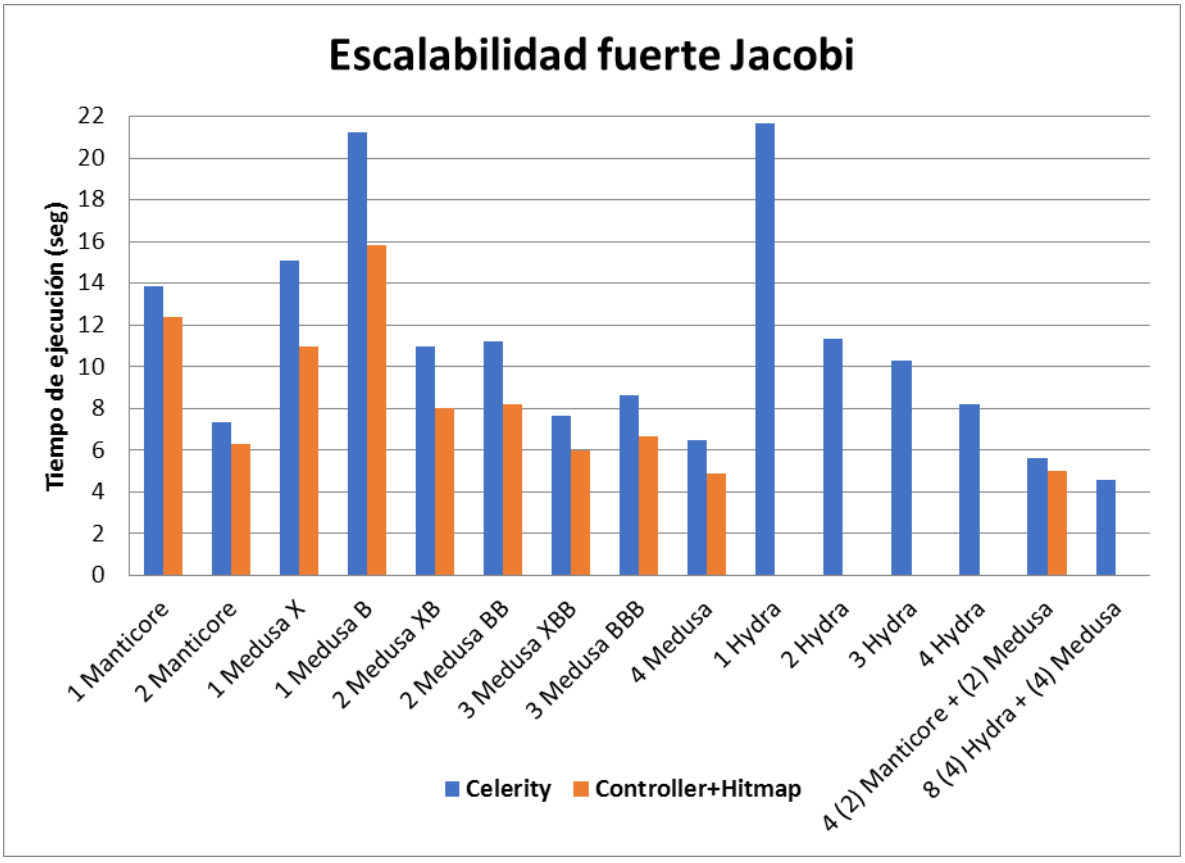

Figura 40: Comparación de tiempos en escalabilidad fuerte para el Stencil Jacobi

Podemos observar que en Manticore y con tamaños más grandes de matrices, los tiempos de ejecución son menores que en Medusa. Esto es por el modelo de GPU, que es más potente. Pero esto solo ocurre en el caso de Celerity. En Medusa, cuando utilizamos la GPU TITAN X con cualquier número de GPUs, es mejor siempre que usando solo las TITAN Black. Por lo demás son los resultados esperables, cuantas más GPUs, menor es el tiempo. Pero comparando los resultados con los obtenidos con Controller+Hitmap, los de Celerity son siempre peores en cualquier caso. La **Figura 41** muestra una gráfica con la media los tiempos, en segundos, de todas las pruebas de escalabilidad débil realizadas para el Stencil Jacobi con Celerity y con Controller+Hitmap.

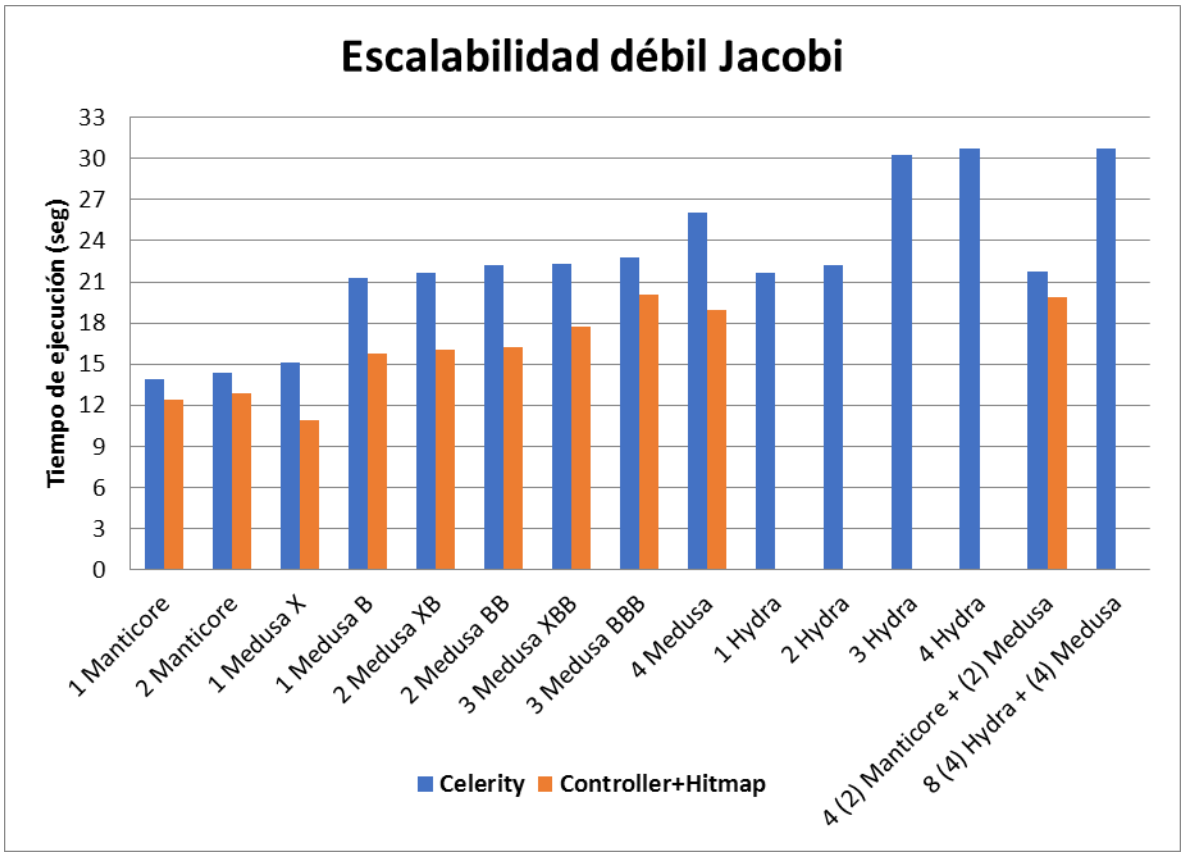

Figura 41: Comparación de tiempos en escalabilidad débil para el Stencil Jacobi

De nuevo, son unos resultados muy estables y que muestran una buena escalabilidad, donde a pesar de multiplicar x2 el tamaño de la matriz por cada GPU, los tiempos de ejecución aumentan muy poco. Y otra vez, en todas las pruebas, los tiempos son menores con la versión de Controller+Hitmap que con Celerity. Aunque está claro a la vista que los tiempos son menores, para poder determinar si la diferencia de medias de las 5 ejecuciones es significativa estadísticamente, comprobamos la hipótesis H0: $\mu 1=\mu 2$, donde $\mu 1$ es la media de los tiempos de ejecución para la prueba de escalabilidad débil Jacobi con 2 GPUs de Manticore en Celerity y $\mu 2$ es la media de los tiempos de ejecución para la prueba de escalabilidad débil Jacobi con 2 GPUs de Manticore en Controller+Hitmap. Para contrastar la hipótesis, asumiendo normalidad y varianzas iguales por simplicidad, el estadístico de contraste que nos permite comparar las medias es el estadístico T:

$$T = \frac{\bar{X} - \bar{Y}}{s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \sim_{H_0} t_{n_1+n_2-2}$$

Donde $\bar{X}$ es la media de las 5 ejecuciones para la prueba seleccionada en Celerity, $\bar{y}$ la media de las 5 ejecuciones para la prueba seleccionada en Controller+Hitmap, n_1 el número de ejecuciones de la prueba en Celerity, n_2 el número de ejecuciones de la prueba en Controller+Hitmap y Sp^2 es el siguiente estadístico:

$$s_p^2 = \frac{(n_1 - 1)s_x^2 + (n_2 - 1)s_y^2}{n_1 + n_2 - 2}$$

Donde Sx es la varianza muestral de X y Sy es la varianza muestral de Y. Tras realizar los cálculos, obtenemos que T=1052.434. Teniendo en cuenta que $t_{8,0.05}$=1.8595, se cumple que {T > $t_{8,0.05}$} y podemos concluir que la confianza de que la diferencia entre las medias sea 0 es menor que 0.05, por lo que rechazamos la hipótesis nula, lo que quiere decir que las medias son comparables y por lo tanto podemos asegurar que en el caso de escalabilidad débil con 2 GPUs en Manticore es más lento en Celerity que en Controller+Hitmap. En los demás casos podríamos hacer este análisis estadístico, pero como las medias son muy diferentes y la varianza muestral pequeña, no lo consideramos necesario.

Para el resto de tipos de Stencil en 2D, los resultados son muy parecidos. En los casos de 2D 9 puntos compacto y no compacto, las gráficas son similares a las de Jacobi, tanto en escalabilidad fuerte como en débil, aunque con los tiempos un poco más altos, ya que la carga de trabajo es más alta, sobre todo en el caso no compacto, ya que tiene que enviar dos filas. En cambio, para el tipo de Stencil 2D 5 puntos asimétrico no compacto encontramos unos resultados especialmente malos para Celerity. La **Figura 42** y la **Figura 43** muestran la comparación en los dos tipos de escalabilidad.

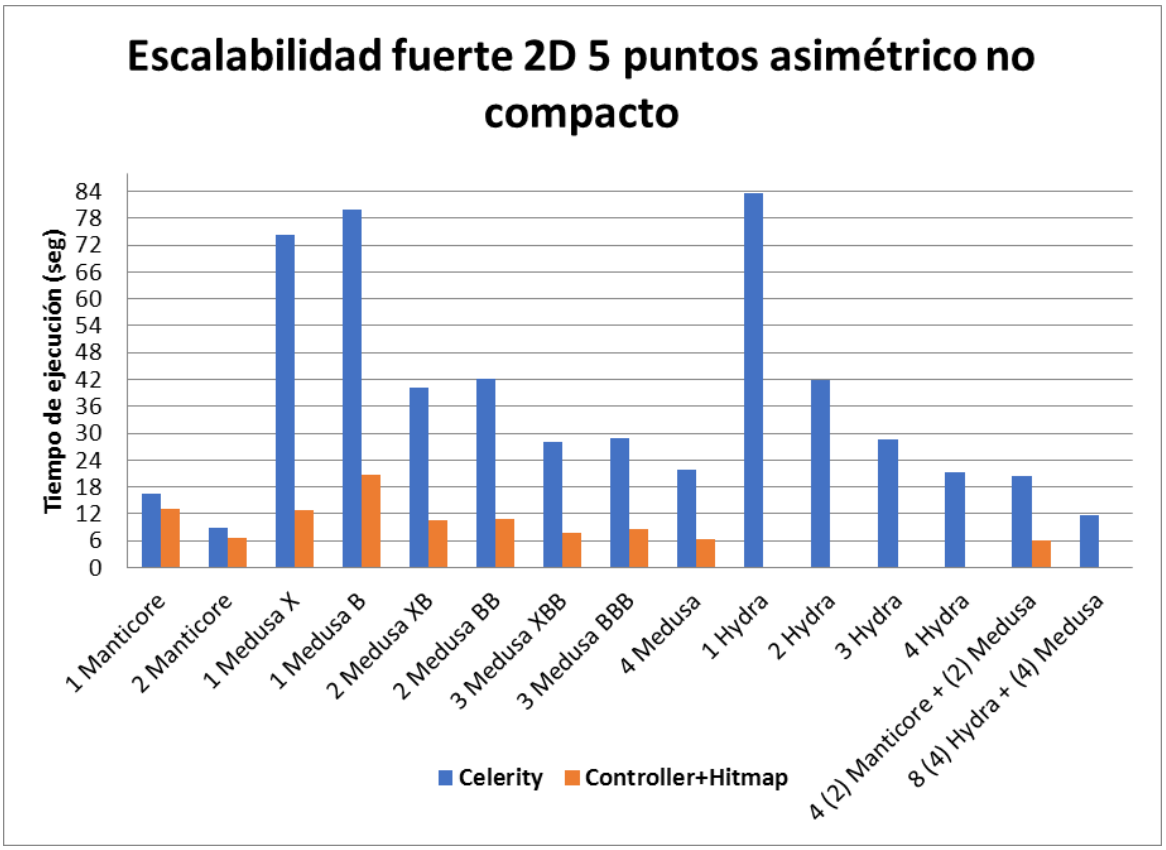

Figura 42: Comparación de tiempos en escalabilidad fuerte para el Stencil 2D 5 puntos asimétrico no compacto

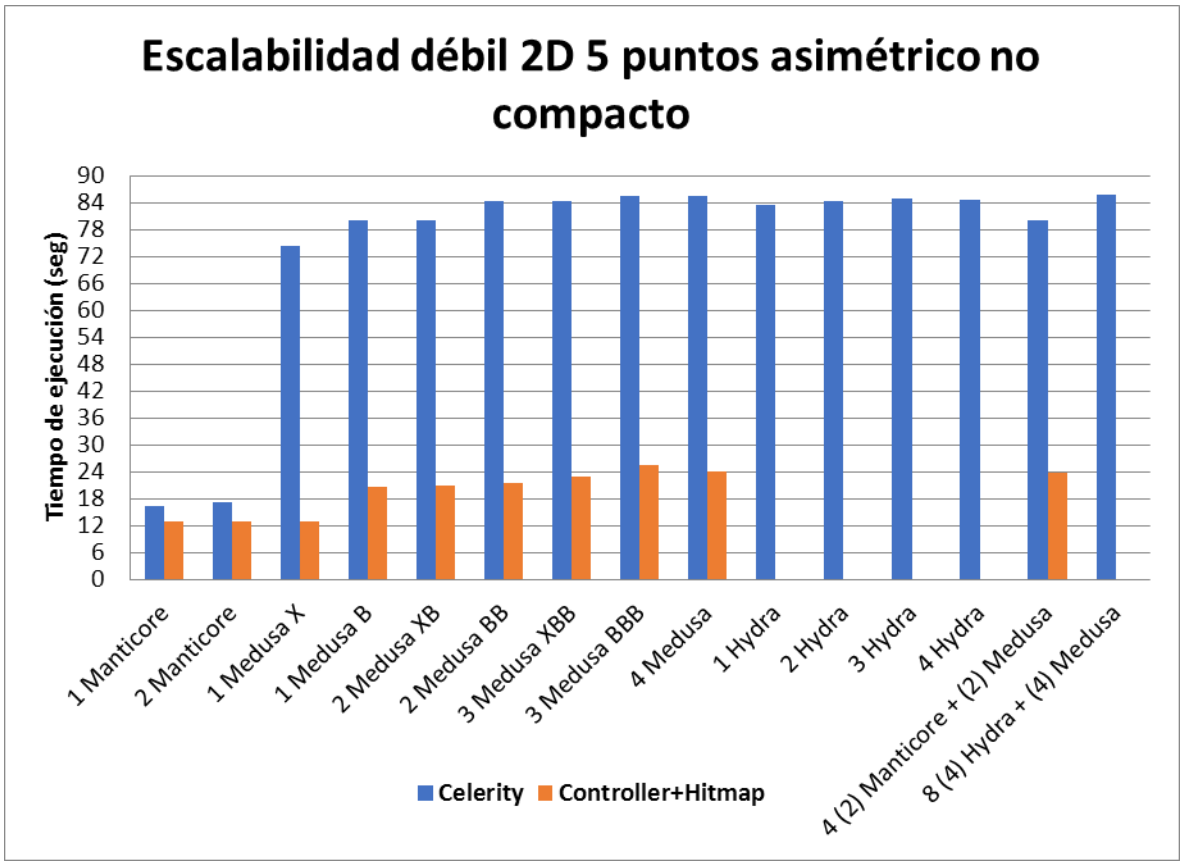

Figura 43: Comparación de tiempos en escalabilidad débil para el Stencil 2D 5 puntos asimétrico no compacto

Si bien cuando utilizamos solo la máquina Manticore, la que tiene las GPUs más nuevas y potentes de las 3 máquinas del clúster, los resultados son peores por bastantes segundos para Celerity, cuando utilizamos Medusa, los tiempos de Celerity se disparan hasta multiplicar x3 y x4 los tiempos con Controller+Hitmap. Aun así, la escalabilidad si es buena, ya que fuerte, a más GPUs, menor es el tiempo, y en débil, a más GPUs y a más tamaño, el tiempo aumenta muy poco. Estos tiempos tan altos en Medusa se podrían deber a que Celerity no optimiza bien para este tipo de arquitectura de NVIDIA, a día de hoy un poco antigua. Sin embargo, en los otros 3 tipos de Stencil 2D no ocurre este comportamiento. La causa principal es que el mapeador *neighborhood* de Celerity por defecto es simétrico. Siempre comunica en las dos direcciones. Pero si el Stencil es asimétrico solo hace falta comunicar en una dirección (que es lo que hace Controller+Hitmap) formando un *pipeline* más eficiente que las comunicaciones parecidas en las dos direcciones.

Pasando al tipo de Stencil 3D 27 puntos compacto, también encontramos unas diferencias de tiempo muy altas. La **Figura 44** muestra la gráfica de escalabilidad débil de este tipo de Stencil.

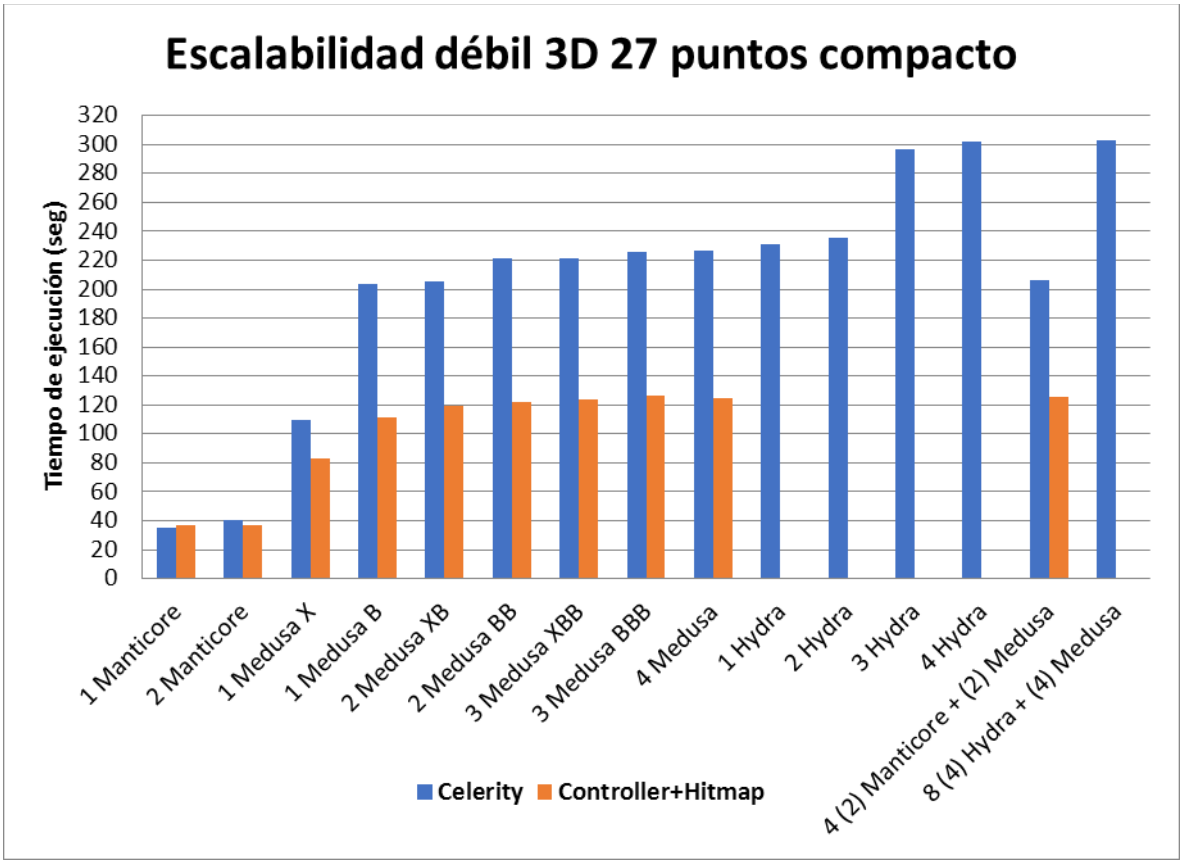

Figura 44: Comparación de tiempos en escalabilidad débil para el Stencil 3D 27 puntos compacto

Por primera vez Celerity es más rápido que Controller+Hitmap, en el caso de 1 GPU de Manticore, lo que tampoco es relevante. Los tiempos son tan altos ya que ahora no se envían filas entre nodos, sino matrices o capas de dos direcciones. Pero de nuevo, los tiempos en Celerity son extremadamente altos. La escalabilidad es correcta, pero en Hydra al llegar a 3 y 4 GPUs se dispara en comparación a con solo 2. Esta subida se puede deber a que con 2 GPUs usa las TITAN Black, pero al llegar a 3 y 4 también utiliza las Tesla K40, que tienen menor frecuencia de reloj y están en un nodo NUMA diferente.

Por último, en los casos de 1D también encontramos alguna comparación interesante. Como podemos observar en la **Figura 45** los resultados son muy similares, variando menos de un segundo, e incluso en los casos de Medusa con 1 y 3 GPUs donde se usa la TITAN Black, los tiempos en Celerity son menores. Con la combinación de 2 de Medusa y 2 de Manticore también ocurre lo mismo.

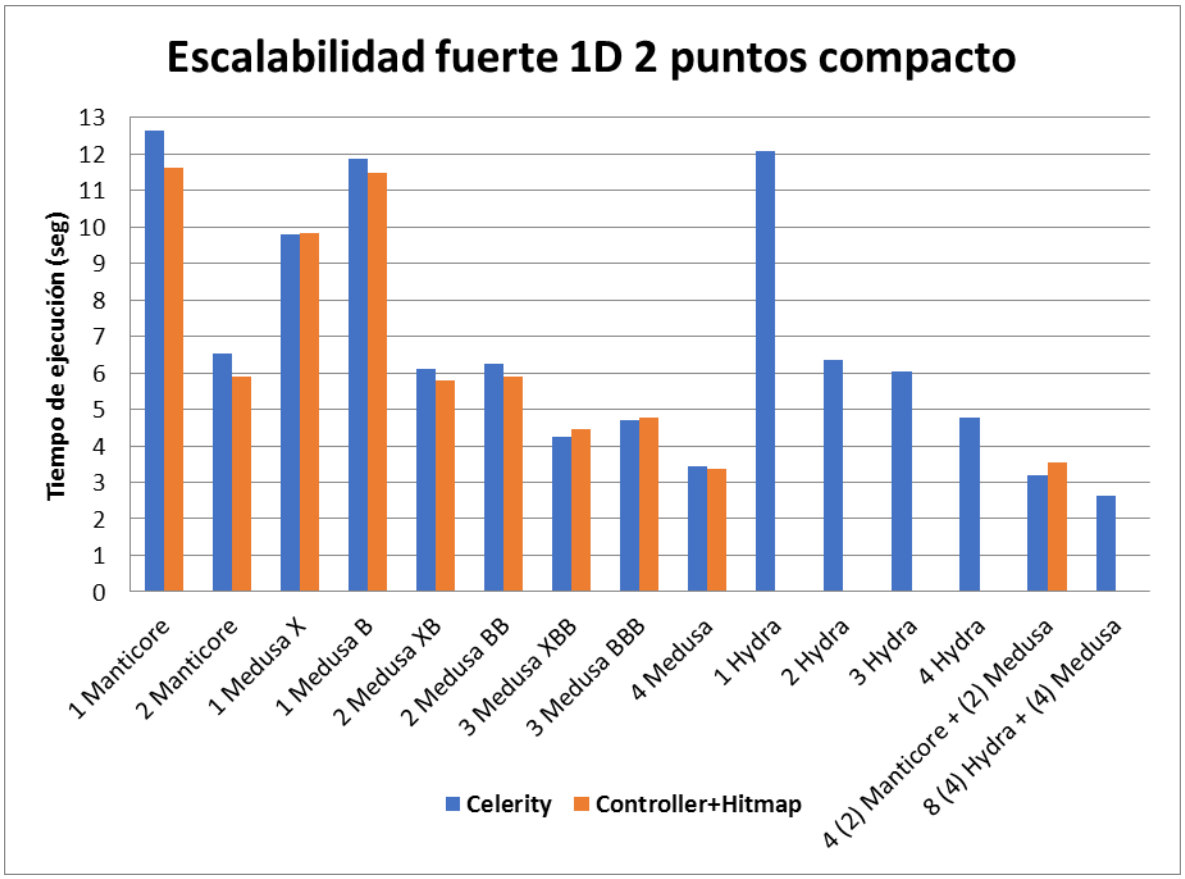

Figura 45: Comparación de tiempos en escalabilidad débil para el Stencil 1D 2 puntos compacto

¿Quiere decir esto que Celerity es mejor que Controller+Hitmap en este Stencil? No, ya que las diferencias, aunque estadísticamente son significativas, son muy pequeñas y además se producen solo en Medusa y solo en algunos casos. Cuando usamos Manticore solamente, gana Controller+Hitmap por casi un segundo. En escalabilidad débil también tenemos el mismo comportamiento, Celerity es más rápido que Controller+Hitmap por poco en los casos de 1 GPU y 3 GPUs de Medusa y es un 15% más rápido en la prueba combinada de Manticore y Medusa. En este último caso en que combinamos Medusa y Manticore, un 15% de mejora sí que podemos considerarlo significativo. Para el caso de 1D 4 puntos compacto siempre es mejor Controller+Hitmap en escalabilidad fuerte y débil, aunque Celerity es mejor en el caso de 3 GPUs de Medusa. En 1D, Controller+Hitmap solo puede computar tamaños de vectores que no superen en tamaño el número *int* máximo, aún no soporta índices *long* cómo Celerity.

Finalmente, podemos determinar que a rasgos generales, en todos los tipos de Stencil Controller+Hitmap es más rápido que Celerity, pero el estudio no ha terminado todavía.

5.4.1 La última comparación decisiva

Si bien todos los resultados anteriores son muy claros, para dotar de una mayor rigurosidad y relevancia a los resultados de nuestro trabajo, vamos a realizar algunas pruebas en el supercomputador MareNostrum 4 del BSC (*Barcelona Supercomputing*

Center), en concreto en el clúster de aceleración CTE-POWER del bloque de tecnologías emergentes [**47**].

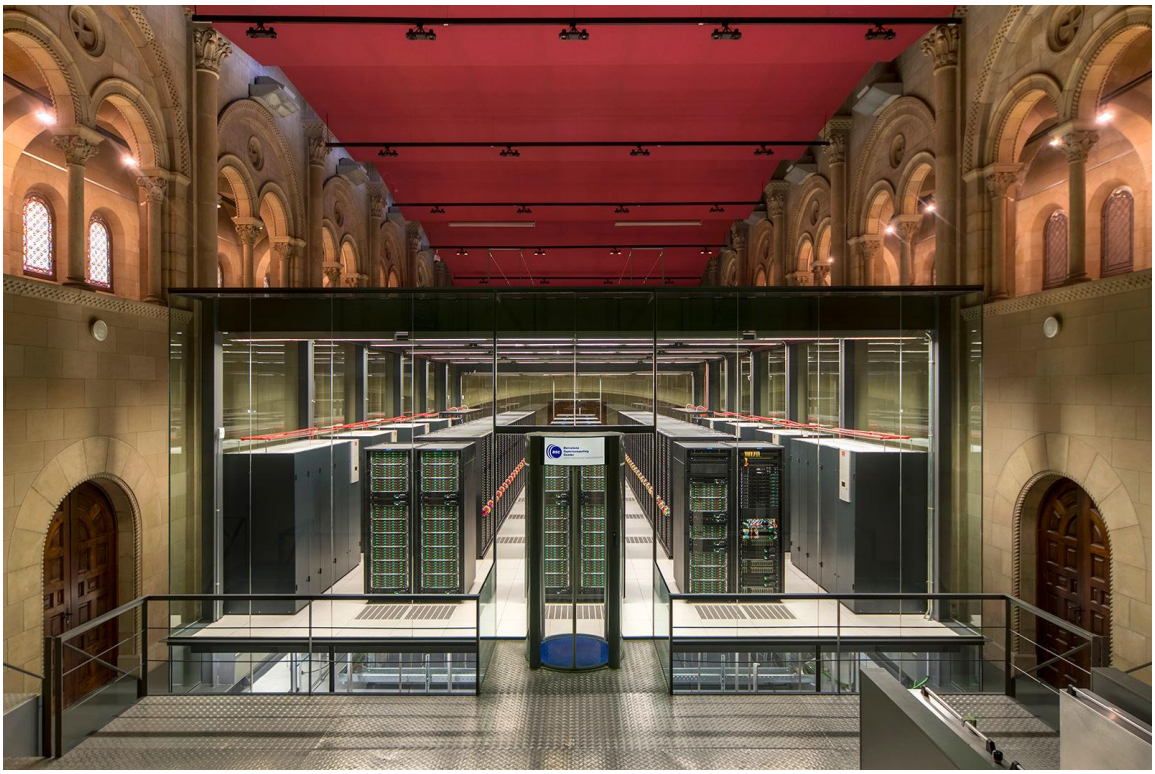

Figura 46: MareNostrum. Figura extraída de **[**47**]**

Este clúster está formado por procesadores IBM Power9, con *Red Hat Enterprise Linux Server 7.5 alternative* como sistema operativo. Está formado por 2 nodos de inicio de sesión y 52 nodos de computación. Cada nodo de computación tiene las siguientes características [**48**]:

- 2 x IBM Power9 8335-GTH @ 2.4GHz (3.0GHz *on turbo*, 20 núcleos y 4 hilos/núcleo, en total 160 hilos por nodo).
- 512GB de memoria principal distribuida en 16 dimms x 32GB @ 2666MHz.
- 2 x SSD 1.9TB como almacenamiento local.
- 2 x 3.2TB NVME.
- 4 x GPU NVIDIA V100 (Volta) con 16GB HBM2.
- Mellanox EDR de puerto único.
- GPFS vía enlace de fibra de 10 Gbit.

El clúster CTE-POWER también está gestionado mediante Slurm, y al ser utilizado por una gran cantidad de usuarios, tenemos que lanzar nuestros programas y esperar un tiempo indeterminado hasta obtener los resultados. Las pruebas que vamos a realizar también varían ligeramente en los tamaños de las matrices, sobre todo en los tamaños de 1D, para mantenernos en el rango de los enteros. Vamos a realizar pruebas con 1, 2, 4, 8, 16 y hasta 32 GPUs. Poder contar con tantas GPUs hace que obtengamos más resultados y más relevantes para comprobar la escalabilidad del programa. Podríamos utilizar

muchas más, pero no será necesario. Los tamaños de los vectores y las matrices para estas pruebas los podemos observar en la **Figura 47**. Siempre son 1000 iteraciones, menos en las pruebas de 1D, que son 10000 iteraciones por la menor carga por iteración en estos casos.

Tamaños de los vectores y matrices de la experimentación con CTE-POWER						
Número GPUs	1D		2D		3D	
	E. débil	E. fuerte	E. débil	E. fuerte	E. débil	E. fuerte
1 GPU	67 000 000	67 000 000	30 000x30 000	30 000x30 000	950x950x950	950x950x950
2 GPUs	134 000 000		60 000x30 000		1 197x1 197x1 197	
4 GPUs	268 000 000		60 000x60 000		1 508x1 508x1 508	
8 GPUs	536 000 000		120 000x60 000		1 900x1 900x1 900	
16 GPUs	1 072 000 000		120 000x120 000		2 394x2 394x2 394	
32 GPUs	2 144 000 000		240 000x120 000		3 016x3 016x3 016	

Figura 47: Tamaños de los vectores y matrices empleados para las pruebas de escalabilidad fuerte y débil en el CTE-POWER

Comenzando de nuevo con los tipos de Stencil en 2D, la **Figura 48** muestra una gráfica comparativa entre las dos herramientas para escalabilidad fuerte del Stencil Jacobi.

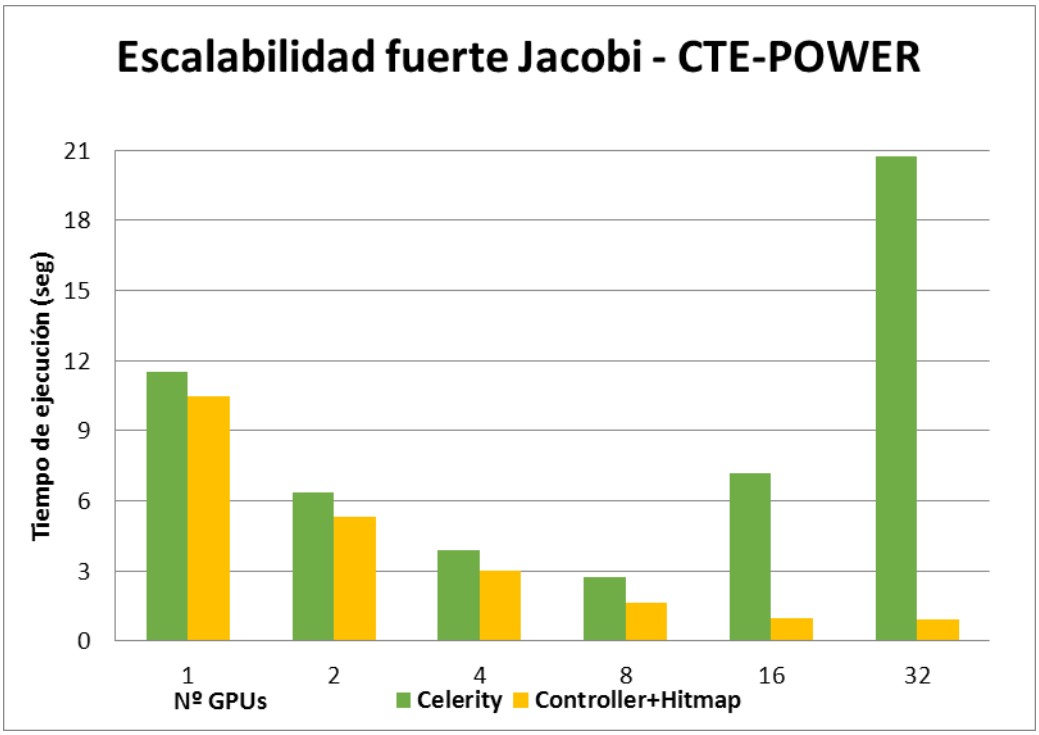

Figura 48: Comparación de tiempos en escalabilidad fuerte para el Stencil 2D Jacobi en el CTE-POWER

Observamos que hasta 8 GPUs, Controller+Hitmap es desde un 9% con 1 GPU hasta un 40% con 8 GPUs más rápido que Celerity, pero al pasar a 16 GPUs, la escalabilidad de Celerity se revierte y comienza a subir mucho en tiempo, llegando a multiplicar x20 el tiempo de Controller+Hitmap cuando usamos 32 GPUs. Cada nodo del CTE-POWER

está formado por 4 GPUs, por lo que cuando solicitamos por ejemplo 8 GPUs, se nos asignan 2 nodos. Las comunicaciones entre nodos son mucho más lentas que entre GPUs del mismo nodo. Pero usando solo 2 nodos no parece afectarle a Celerity, el problema empieza cuando usamos 4 o más nodos. Podemos entonces plantear que Celerity tiene un serio problema con las comunicaciones, aunque antes tenemos que ver la escalabilidad débil, **Figura 49**.

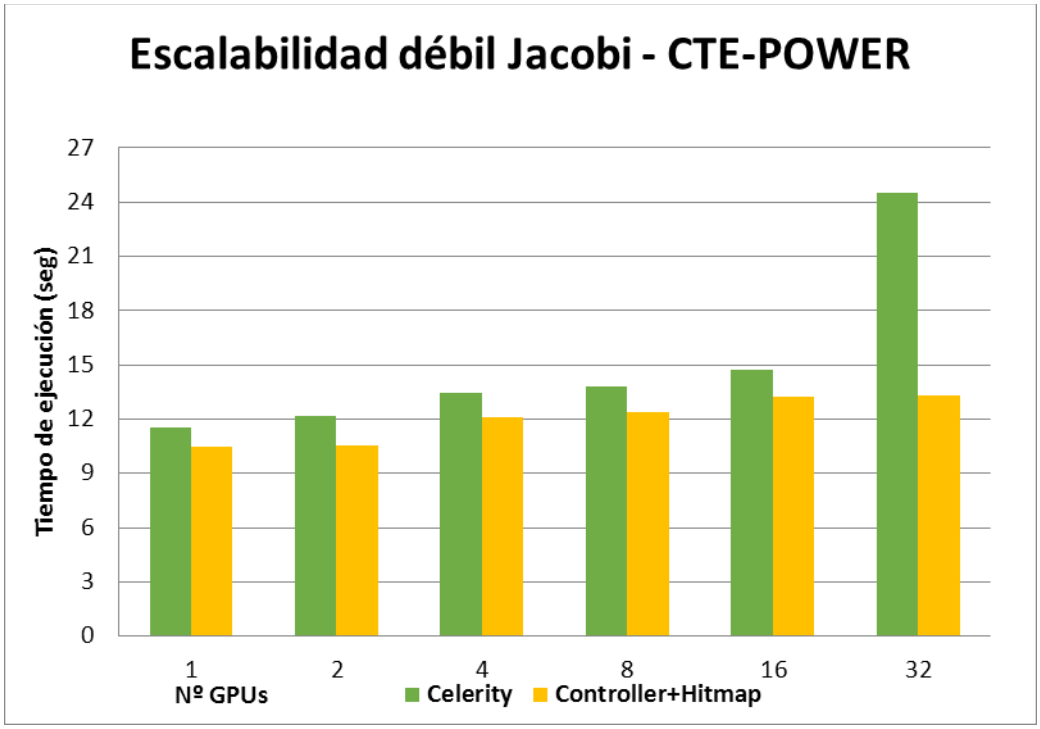

Figura 49: Comparación de tiempos en escalabilidad débil para el Stencil 2D Jacobi en el CTE-POWER

De nuevo, las diferencias de tiempos entre ambas herramientas no son muy grandes, de unos pocos segundos. Ahora parece que con 16 GPUs todavía no se aprecia un repunte muy alto, pero al llegar a 32 GPUs, de nuevo la escalabilidad se pierde completamente. Para los demás casos de Stencil 2D obtenemos los mismos resultados, no se notan apenas diferencias de tiempos como pasaba en las pruebas realizadas en el primer clúster que hemos utilizado. Viendo las gráficas parecen todas del mismo tipo de Stencil, lo que va confirmando nuestra teoría de que Celerity no optimiza bien el código para arquitecturas antiguas de GPUs, ya que en este clúster al igual que con la máquina Manticore del primer clúster, las GPUs son las mismas, Volta V100, y cuando usábamos Manticore en el primer clúster, los tiempos eran más o menos los mismos entre los 4 tipos de Stencil 2D, hecho que no ocurría con Medusa o Hydra.

Pasando a tipo de Stencil 3D 27 puntos compacto, en escalabilidad fuerte observamos la misma tendencia que con 2D, a partir de 16 GPUs, la escalabilidad se revierte, mientras que con Controller+Hitmap a partir de 16 GPUs aumenta muy levemente el tiempo, siendo prácticamente el mismo que usando 8 GPUs. De nuevo, esto es debido a

las comunicaciones internodo, pero que en Controller+Hitmap no afectan apenas. La **Figura 50** muestra la comparación en escalabilidad débil del Stencil 3D.

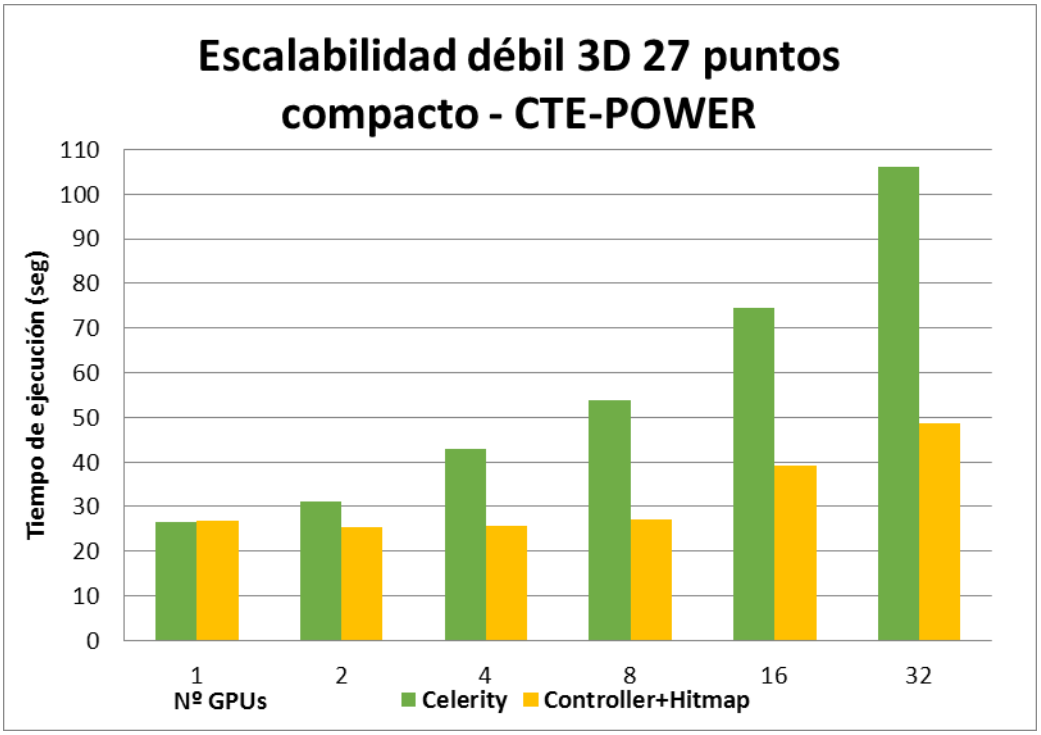

Figura 50: Comparación de tiempos en escalabilidad débil para el Stencil 3D 27 puntos compacto en el CTE-POWER

Utilizando solo 1 GPU, los tiempos de las dos herramientas son los mismos, como ya pasaba cuando utilizábamos 1 GPU de Manticore. Pero a partir de 1 GPU, vemos como Celerity empieza a aumentar de tiempo considerablemente hasta llegar a los 105 segundos con 32 GPUs, mientras que con Controller+Hitmap hasta 8 tiene una escalabilidad débil buena, pero después sufre también las consecuencias de las comunicaciones internodo, aunque sin llegar a los niveles de Celerity.

Por último, en el caso de los Stencil 1D, las pruebas que hemos realizado tienen un tamaño menor, ya que cuando lleguemos a 32 GPUs no debemos superar el máximo *int*, por lo que comenzamos con un tamaño de 67 000 000 (en 2D comenzábamos con 30 000 x 30 000 = 900 000 000). Como este tamaño es muy pequeño, aumentamos otro parámetro del programa, que es el número de iteraciones. En todas las pruebas se han realizado 1000 iteraciones, pero en estas vamos a usar 10 000 iteraciones. Empezar con tamaños menores puede hacer que obtengamos resultados interesantes. La **Figura 51** muestra una comparativa de los tiempos para el caso de escalabilidad fuerte en el Stencil 1D 2 puntos compacto.

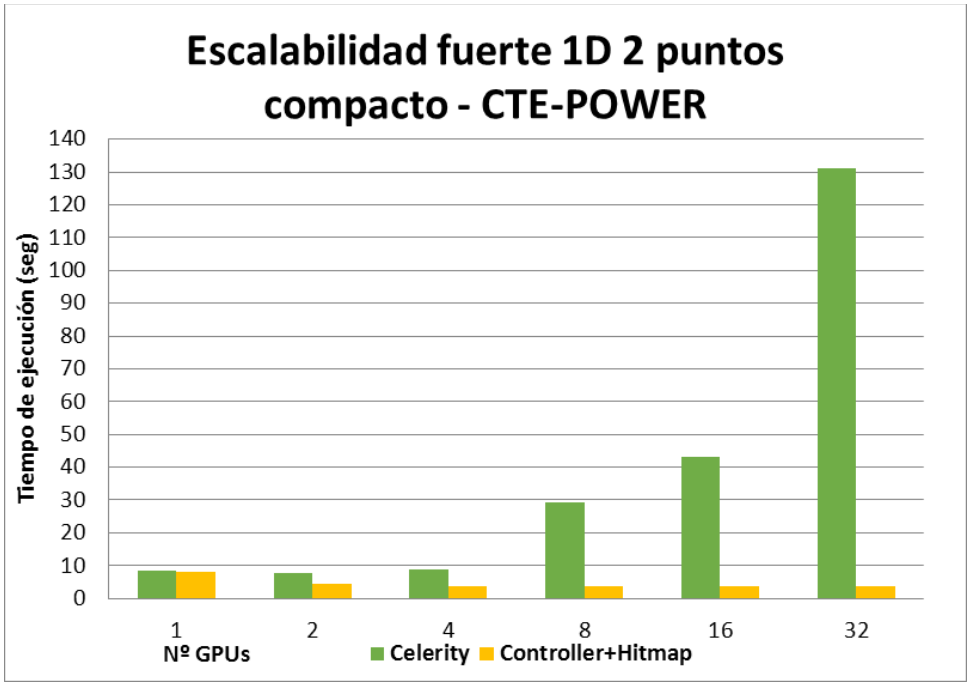

Figura 51: Comparación de tiempos en escalabilidad fuerte para el Stencil 1D 2 puntos compacto en el CTE-POWER

Hasta 4 GPUs, en Celerity la escalabilidad es bastante mala, ya que apenas disminuye el tiempo, pero ya con 8 GPUs los tiempos aumentan en exceso, mientras que con Controller+Hitmap la escalabilidad es buena, aunque se estanca a partir de 8 GPUs, pero sin punto de comparación con Celerity. La **Figura 52** muestra el mismo caso pero en escalabilidad débil.

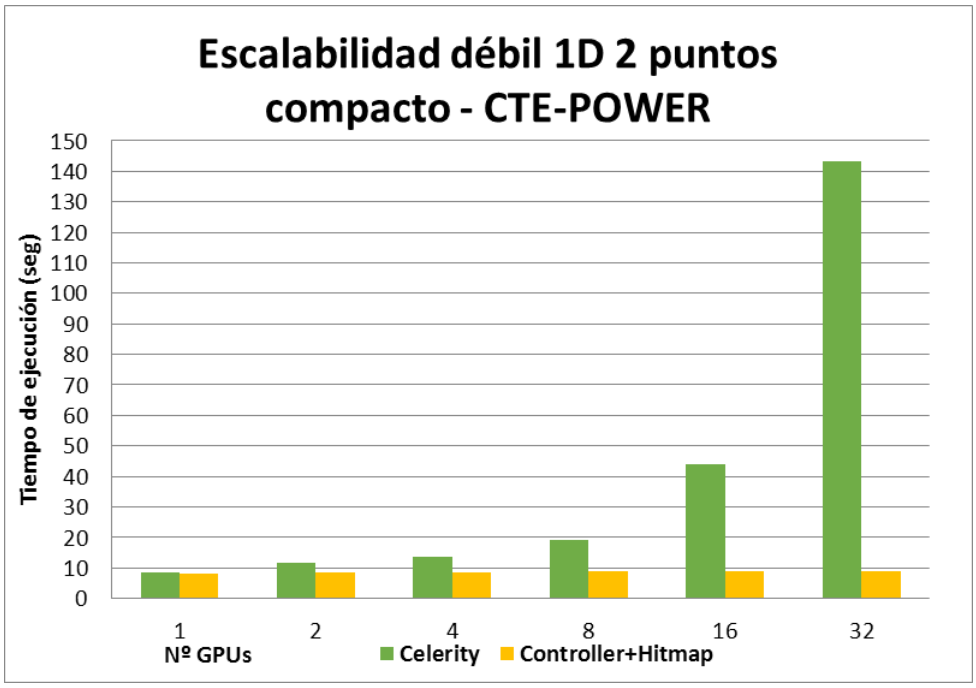

Figura 52: Comparación de tiempos en escalabilidad débil para el Stencil 1D 2 puntos compacto en el CTE-POWER

Con escalabilidad débil, Controller+Hitmap funciona bien, ya que solo aumenta en centésimas de segundo al aumentar el número de GPUs y el tamaño de los vectores. Por el contrario, Celerity escala más o menos bien hasta 4 GPUs y a partir de ese punto la escalabilidad se pierde y se llega a los mismos tiempos que en escalabilidad fuerte. En el caso del Stencil 1D 4 puntos compacto, los resultados son similares. De nuevo, podemos deducir que las comunicaciones entre distintos nodos afectan mucho al rendimiento de Celerity, sobre todo en casos como este en que los tamaños de las matrices son muy pequeños en comparación con el número de GPUs que se utilizan.

En la siguiente sección se muestran las conclusiones finales del estudio.

5.5 Conclusiones del estudio

Después de comparar todos los resultados, hemos comprobado que Celerity es peor en todos los tipos de Stencil programados y para cualquier número de GPUs. Las conclusiones más importantes que hemos obtenido son las siguientes:

- Celerity optimiza peor los códigos cuanto más antiguas sean sus GPUs, lo hemos podido comprobar ya que los tiempos en Manticore, arquitectura Volta, no eran altos y solo era más lento por unos segundos. En cambio, cuando usábamos Medusa e Hydra, arquitectura Kepler y Maxwell, los tiempos empeoraban considerablemente, mientras que con Controller+Hitmap no se producía tal suceso.
- Celerity no escala bien cuando se producen una gran cantidad de comunicaciones entre GPUs de distintos nodos, lo hemos comprobado con las pruebas realizadas en el BSC. Aunque con 8 GPUs, que eran solo 2 nodos, la escalabilidad era correcta, cuando usábamos 16 GPUs (4 nodos) y 32 GPUs (8 nodos), la escalabilidad desaparecía y los tiempos aumentaban enormemente. En 3D y escalabilidad débil, Controller+Hitmap también sufría con la escalabilidad, pero sin llegar a los extremos de Celerity. En escalabilidad fuerte en todas las dimensiones a partir de 8 GPUs los tiempos de Celerity empeoraban.
- Controller+Hitmap es una herramienta que todavía no está terminada, y por tanto hay pruebas que no se han podido realizar, como las de Hydra en el primer clúster y cómo las pruebas que nos hubiera gustado en el BSC en 1D, que nos obligaron a reducir los tamaños de los vectores para que entraran en un *int*.
- Programar esta aplicación Stencil en Celerity no es muy complicado hasta para un programador que nunca haya utilizado SYCL. En cambio, programar esta aplicación usando Controller+Hitmap es un proceso mucho más complicado y largo.

La **Figura 53** muestra una gráfica que recoge los tiempos medios de todas las pruebas de escalabilidad fuerte realizadas en el CTE-POWER, y la **Figura 54** en el caso de

escalabilidad débil. En estás gráficas no se han incluido los Stencil 1D ya que son distintos tamaños e iteraciones que el resto.

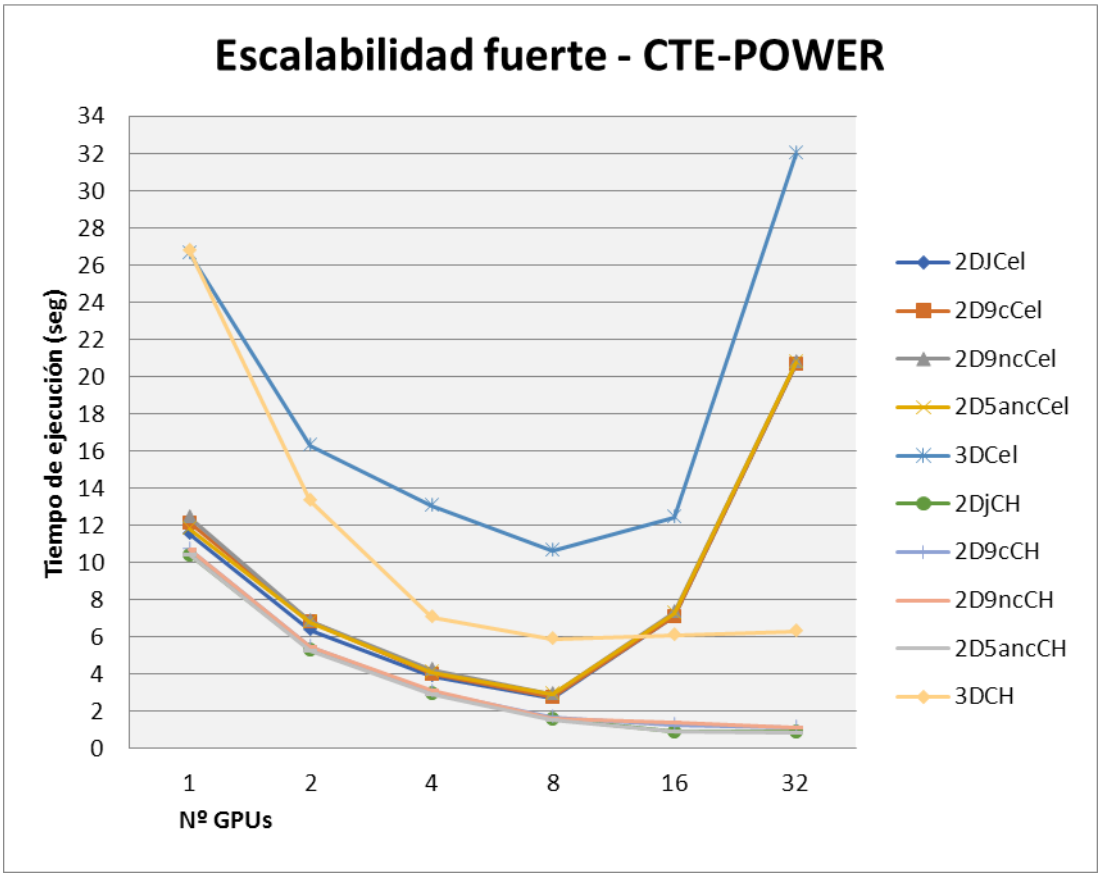

Figura 53: Gráfica resumen de la escalabilidad fuerte para todos los tipos de Stencil

Entonces cómo consigue obtener tan buen rendimiento la aplicación Controller+Hitmap. Con Celerity, tal y como veíamos en la **Figura 38**, en cada tarea se muestra con que rango de índices está lanzando el *kernel*. Lanza para todo el array (menos los bordes exteriores y los halos) y calcula el Stencil para cada posición. Después, ya puede sacar los bordes y mandarlos a otro proceso. En cambio, con Controller+Hitmap primero se manda un *kernel* con sólo el rango de índices de un borde interior, no del halo, sino la primera fila que habría que computar. Así para todos los bordes. Como son *grids* muy pequeños acaban enseguida. En cuanto terminan se inician las transferencias con los comandos de CUDA para sacar la memoria al *host*. Pero mientras esto se hace, se ha lanzado el *kernel* con el rango de índices de toda la que no son bordes interiores y que haya que enviar a ningún otro proceso, lo que es el centro del array. Y de esta manera se dispone de todo el tiempo que tarda ese *kernel* para terminar de recibir en el *host* los datos que salen de la GPU, mandarlos con MPI, recibir los que tienen que llegar e introducir en la GPU esos halos/bordes exteriores que se necesitan para la siguiente iteración. Ese es el truco de esta herramienta, el *kernel* se parte en varias transferencias de datos que se realizan mientras se ejecuta el *kernel* principal en cada iteración. Para cuando termina el *kernel* principal ya tenemos en la GPU colocados los

halos que vienen de otros procesos. Para poder implementar este comportamiento en Celerity, se tendría que tocar su código interno, no siendo trivial, pero sin ser muy complejo tampoco. Aplicando este cambio, suponemos que eliminaría su principal problema y podría tener rendimientos al menos similares a los de Controller+Hitmap.

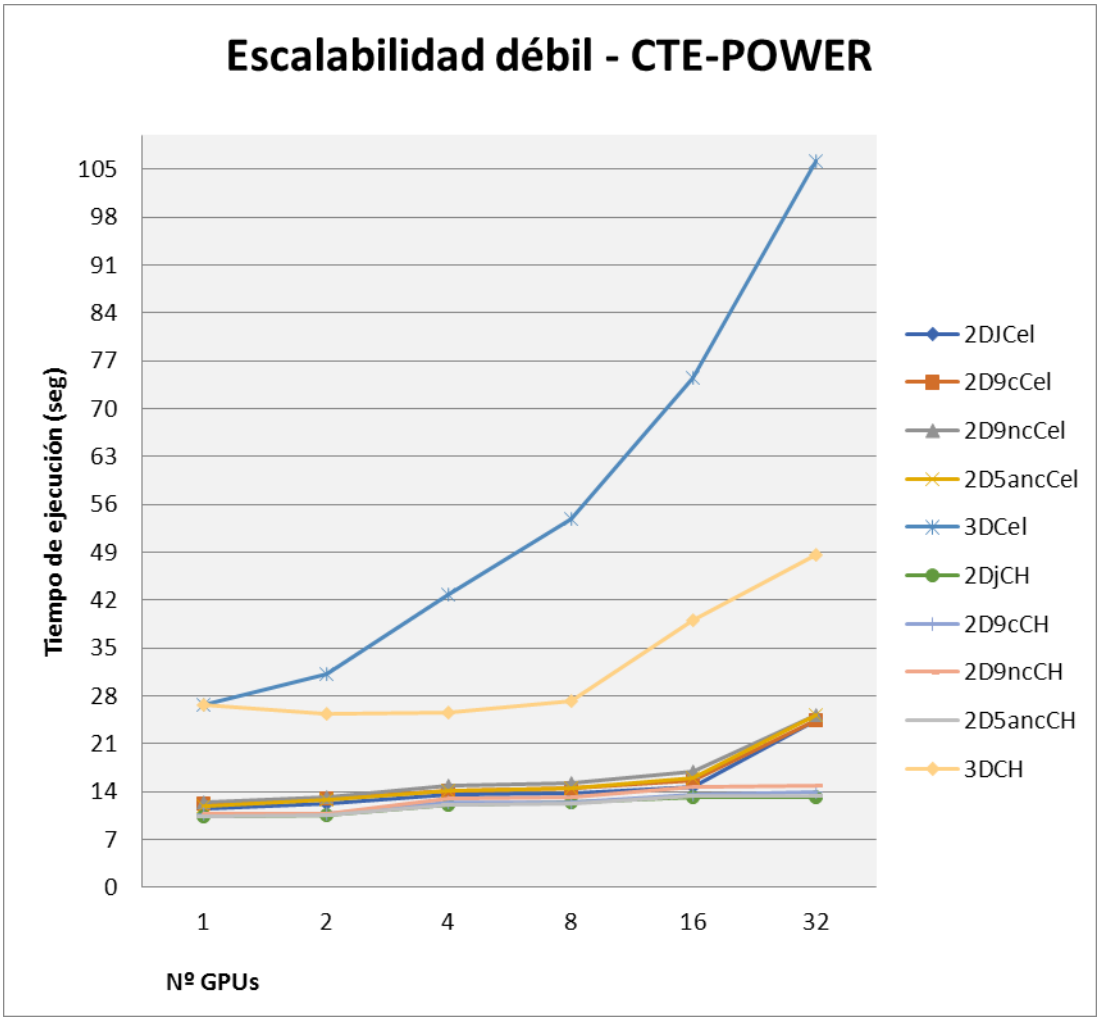

Figura 54: Gráfica resumen de la escalabilidad débil para todos los tipos de Stencil

6

Conclusiones y líneas futuras

6.1 Conclusiones

En el presente Trabajo de Fin de Máster se ha realizado un estudio y evaluación del modelo de programación heterogénea distribuida Celerity.

En este proyecto hemos pasado todas las etapas para la ejecución de un estudio experimental. Primero con la fase de investigación de las tecnologías y herramientas a utilizar. Aunque teníamos experiencia previa con MPI y CUDA, éramos noveles en SYCL y Celerity. Después pasamos a la fase de análisis de Celerity, donde primero mostramos cómo se compila e instala la herramienta y cómo se compila un programa escrito con ella. Además en esta fase explicamos cómo funciona por debajo Celerity y qué y por qué son tan importantes los mapeadores en Celerity. Tras tener las nociones más importantes de Celerity, comenzamos la parte del estudio experimental, donde comenzamos con la programación de la aplicación que vamos a evaluar, un programa que calcula diferentes tipos de patrones Stencil en vectores y matrices en 1, 2 y 3 dimensiones. Con la aplicación terminada y optimizada al máximo, empezamos la experimentación en un clúster heterogéneo con 3 máquinas y diferentes GPUs, tanto en número como en modelo. La comparación se hizo con la aplicación Controller+Hitmap, desarrollada por el grupo Trasgo. Como los resultados fueron tan diferentes, realizamos otras pruebas en el clúster CTE-POWER del supercomputador barcelonés BSC. Los resultados mostraron que siempre era más rápida la versión del programa con Controller+Hitmap que con Celerity, para todos los tipos de Stencil, siendo las transferencias entre nodos el punto crítico de Celerity, sobre todo cuando se utilizaban una gran cantidad de nodos de cómputo.

Los resultados preliminares generados durante el desarrollo de este TFM han sido utilizados para un estudio experimental similar que ha dado lugar a la publicación de un póster en el congreso Supercomputing 2021 (SC'21). Este congreso es de categoría GGS 1/CORE A+, que es una de las categorías más importante de congresos de informática, y el SC es uno de los mejores congresos relacionados con la supercomputación. El título del póster es *Towards an efficient parallel skeleton for generic iterative stencil computations in distributed GPUs* y sus autores son Manuel de Castro, Inmaculada Santamaria-Valenzuela, Sergio Miguel-López (el autor de este TFM), Yuri Torres y

Arturo Gonzalez-Escribano. El póster presenta un *skeleton* paralelo para el cómputo de patrones Stencil en un entorno de GPUs distribuidas, soporta cualquier tipo de Stencil geométrico de cualquier dimensión. La herramienta está construida por encima de Controller+Hitmap, por lo que permite un particionamiento de datos y comunicaciones transparentes y una superposición de la comunicación de datos y la computación [**49**].

6.2 Líneas Futuras

En principio, el título de este TFM, *Celerity: el futuro de la programación paralela en memoria distribuida*, parece que no tiene sentido, ya que hemos visto que la herramienta Controller+Hitmap es más eficiente y rápida que Celerity. Sin embargo, Celerity tiene un valor muy importante, se basa en el estándar ampliamente utilizado SYCL. Además, el nivel de abstracción del lenguaje, especialmente gracias a los mapeadores, es mayor que en soluciones similares como Controller+Hitmap. Con Celerity te olvidas de dividir la carga de trabajo, de los envíos y de otras operaciones de MPI debido a la memoria distribuida y te centras en lo importante, la parte de cómputo. Controller+Hitmap necesita desarrollar una capa de abstracción extra para ser tan fácil de programar como Celerity y Celerity necesita implementar algunas soluciones técnicas y conceptuales en su sistema de ejecución y generación de grafos de tareas similares a Controller+Hitmap.

Desde este TFM y desde el grupo Trasgo, tras la publicación definitiva del póster (y su correspondiente artículo), proponemos una colaboración a los desarrolladores de Celerity, para implementar el funcionamiento de Controller+Hitmap en Celerity, de manera que consigamos con Celerity solapar las comunicaciones y la computación, con lo que reduciremos el tiempo del programa en Celerity de manera considerable. Si se llegase a implementar, entonces si podríamos decir que Celerity es el futuro de la programación paralela en memoria distribuida para los clústeres de aceleración.

7 Bibliografía

[1] Rachel Courtland. (2015) Gordon Moore: The Man Whose Name Means Progress. [Online]. https://spectrum.ieee.org/gordon-moore-the-man-whose-name-means-progress

[2] TOP 500 - Junio 2021. [Online]. https://www.top500.org/lists/top500/2021/06/, Último acceso: Septiembre 2021

[3] GREEN 500 - Junio 2021. [Online]. https://www.top500.org/lists/green500/2021/06/, Último acceso: Septiembre 2021

[4] Message Passing Interface Forum. (2021, Junio) MPI: A Message-Passing Interface Standard Version 4.0. [Online]. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, Último acceso: Septiembre 2021

[5] CUDA Zone. [Online]. https://developer.nvidia.com/cuda-zone, Último acceso: Septiembre 2021

[6] OpenMP. [Online]. https://www.openmp.org/, Último acceso: Septiembre 2021

[7] Peter Thoman, Philip Salzmann, Biagio Cosenza, and Thomas Fahringer, "Celerity: High-level C++ for Accelerator Clusters," *Euro-Par*, 2019.

[8] Philipp Gschwandtner, Philip Salzmann, and Peter Thoman, "Celerity High-level C++ for Accelerator Clusters," *Talk at AHPC*, 2020.

[9] The Khronos SYCL Working Group. (2021, Marzo) SYCL 2020 Specification (revision 3). [Online]. https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf, Último acceso: Septiembre 2021

[10] Arturo Gonzalez-Escribano, Yuri Torres, Javier Fresno, and Diego R. Llanos, "An

Extensible System for Multilevel Automatic Data Partition and Mapping," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, Mayo 2014.

[11] Ana Moreton-Fernandez, Hector Ortega-Arranz, and Arturo Gonzalez–Escribano, "Controllers: An abstraction to ease the use of hardware accelerators," *The International Journal of High Performance Computing Applications*, vol. 32, no. 6, pp. 838-853, Mayo 2018.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier, "STARPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *LNCS*, no. 5704, pp. 863-874, 2009.

[13] August Ernstsson, Lu Li, and Christoph Kessler, "SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems," *International Journal of Parallel Programming*, no. 46, pp. 62-80, Enero 2017.

[14] Christian Trott and Carter Edwards, "Kokkos: The C++ Performance Portability Programming Model," 2017.

[15] Biagio Peccerillo and Sandro Bartolini, "PHAST - A portable high-level modern C++ programming library for GPUs and multi-cores," Julio 2018.

[16] Michael Bauer, Sean Treichler, Elliot Slaughter, and Alex Aiken, "Legion: expressing locality and independence with logical regions," *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012.

[17] Sunita Chandrasekaran and Guido Juckeland, *OpenACC for Programmers*.: Addison-Wesley Professional, 2017.

[18] Philipp Kegel, Michel Steuwer, and Sergei Gorlatch, "dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems," *IPDPSW 26th*, pp. 174-186, 2012.

[19] Rod Burns, John Lawson, Duncan McBain, and Daniel Soutar, "Accelerated Neural Networks on OpenCL Devices Using SYCL-DNN," *International Workshop on*

OpenCL (IWOCL '19), Mayo 2019.

[20] Cheng Tan et al., "ARENA: Asynchronous Reconfigurable Accelerator Ring to Enable Data-Centric Parallel Computing," Abril 2021.

[21] Jorge González-Domínguez, Roberto R. Expósito, and Verónica Bolón-Canedo, "CUDA-JMI: Acceleration of Feature Selection on Heterogeneous Systems," *Future Generation Computer Systems*, Octubre 2020.

[22] About CMake - Overview. [Online]. https://cmake.org/overview/, Último acceso: Septiembre 2021

[23] The LLVM Compiler Infrastructure. [Online]. https://llvm.org/, Último acceso: Septiembre 2021

[24] Francisco Javier Ceballos Sierra, *C/C++ Curso de Programación*, Tercera ed.: Ra-Ma, 2007.

[25] GCC, the GNU Compiler Collection. [Online]. https://gcc.gnu.org/, Último acceso: Septiembre 2021

[26] Página web de Boost C++. [Online]. https://www.boost.org/, Último acceso: Septiembre 2021

[27] GNU MPC. [Online]. https://directory.fsf.org/wiki/Mpc, Último acceso: Septiembre 2021

[28] MPICH. [Online]. https://www.mpich.org/, Último acceso: Septiembre 2021

[29] Open MPI. [Online]. https://www.open-mpi.org/, Último acceso: Septiembre 2021

[30] MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. [Online]. http://mvapich.cse.ohio-state.edu/

[31] NVIDIA Corporation. (2021, Junio) CUDA C Programming Guide - v11.4.0. [Online]. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html,

Último acceso: Septiembre 2021

[32] M Hernández et al., "Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs," *PLoS ONE*, vol. 8, no. 4, Abril 2013.

[33] Arturo González Escribano. (2018) CUDA - Introducción al modelo de programación.

[34] KHRONOS Group. SYCL. [Online]. https://www.khronos.org/sycl/, Último acceso: Septiembre 2021

[35] Aksel Alpay and Vincent Heuveline, "SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL," *Proceedings of the International Workshop on OpenCL (IWOCL '20)*, no. 8, Abril 2020, https://doi.org/10.1145/3388333.3388658.

[36] hipSYCL - a SYCL implementation for CPUs and GPUs. [Online]. https://github.com/illuhad/hipSYCL, Último acceso: Septiembre 2021

[37] Celerity - Getting Started. [Online]. https://celerity.github.io/docs/getting-started, Último acceso: Septiembre 2021

[38] Celerity Overview - Celerity Execution Timeline. [Online]. https://celerity.github.io/docs/overview, Último acceso: Septiembre 2021

[39] Gabriel Rodriguez-Canal, Yuri Torres, Francisco J. Andújar, and Arturo Gonzalez-Escribano, "Efficient heterogeneous programming with FPGAs using the Controller model," *The Journal of Supercomputing*, Abril 2021.

[40] Grupo Trasgo. [Online]. https://trasgo.infor.uva.es/, Último acceso: Septiembre 2021

[41] Building and installing hipSYCL. [Online]. https://github.com/illuhad/hipSYCL/blob/develop/doc/installing.md, Último acceso: Septiembre 2021

[42] Celerity - Installation. [Online]. https://celerity.github.io/docs/installation, Último acceso: Septiembre 2021

[43] Philip Salzmann, "High-level C++ for Accelerator Clusters - Introducing the Celerity Runtime," Febrero 2019.

[44] Distr_io.cc - Celerity. [Online]. https://github.com/celerity/celerity-runtime/blob/master/examples/distr_io/distr_io.cc, Último acceso: Septiembre 2021

[45] Biagio Cosenza. Celerity Bench. [Online]. https://github.com/bcosenza/celerity-bench

[46] Chrono - Time Library. [Online]. https://www.cplusplus.com/reference/chrono/, Último acceso: Septiembre 2021

[47] MareNostrum. [Online]. https://www.bsc.es/marenostrum/marenostrum, Último acceso: Septiembre 2021

[48] CTE-POWER User's Guide. [Online]. https://www.bsc.es/user-support/power.php, Último acceso: Septiembre 2021

[49] Manuel de Castro, Inmaculada Santamaria-Valenzuela, Sergio Miguel-López, Yuri Torres, and Arturo Gonzalez-Escribano, "Towards an efficient parallel skeleton for generic iterative stencil computations in distributed GPUs," *Supercomputing Conference*, Noviembre 2021.

Anexo I: Ejemplo MPI

En este Anexo mostramos el código completo del programa de ejemplo de MPI:

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
int main(int argc, char *argv[]) {

        MPI_Init(&argc, &argv);

        int num_procs, rank;
        char maquina[MPI_MAX_PROCESSOR_NAME];
        int maquina_len;

        MPI_Get_processor_name(maquina, &maquina_len);
        MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        printf("Proceso: %d de %d, en maquina: %s\n", rank, num_procs, maquina);

        int totalSize=2000000;
        int size=totalSize/num_procs;
        int begin=size*rank;
        long *enteros=(long *)malloc(sizeof(long) * size);
        int i;
        long suma=0;

        for(i=0;i<size;i++) enteros[i]=i + begin;
        for(i=0;i<size;i++) suma=suma + enteros[i];

        if(rank==1) MPI_Send(&suma,1,MPI_LONG,rank-1,288,MPI_COMM_WORLD);

        long sumaRecv;
        MPI_Status status;
        if(rank==0)
                MPI_Recv(&sumaRecv,1,MPI_LONG,rank+1,288,MPI_COMM_WORLD,&status);

        if(rank==0){
                suma=suma + sumaRecv;
                printf("La suma es: %ld\n", suma);
        }

        MPI_Finalize();
        return 0;
}
```

Anexo II: Ejemplo CUDA

En este Anexo mostramos el código completo del programa de ejemplo de CUDA:

```
#include <stdio.h>
#include <cuda.h>

__global__ void vectorAdd(int *A, int *B, int *C, int elements){

    int tid=threadIdx.x + blockIdx.x * blockDim.x;
    if (tid<elements){
        C[tid]= A[tid]+B[tid];
    }
}

int main(){

        cudaSetDevice(0);
        cudaError_t error;
        int elements=10000;
        int tamBlockX=1024;
        int tamGridX;

        if(elements%tamBlockX==0){
                tamGridX=elements/tamBlockX;
        }else{
                tamGridX=elements/tamBlockX+1;
        }
        dim3 tamBlock(tamBlockX,1);
        dim3 tamGrid(tamGridX,1);

        int *A=(int *) malloc(sizeof(int) * (size_t) elements);
        int *B=(int *) malloc(sizeof(int) * (size_t) elements);
        int *C=(int *) malloc(sizeof(int) * (size_t) elements);
        int *Agpu, *Bgpu, *Cgpu;

        error=cudaMalloc((void**) &Agpu, sizeof(int) * (size_t) elements);
        if(error!=cudaSuccess)
          printf("Error malloc A: %s\n", cudaGetErrorString(error));
        error=cudaMalloc((void**) &Bgpu, sizeof(int) * (size_t) elements);
        if(error!=cudaSuccess)
          printf("Error malloc B: %s\n", cudaGetErrorString(error));
        error=cudaMalloc((void**) &Cgpu, sizeof(int) * (size_t) elements);
        if(error!=cudaSuccess)
          printf("Error malloc C: %s\n", cudaGetErrorString(error));

        for(int i=0; i<elements; i++){
                A[i]=i;
```

```
                B[i]=i+1;
        }

        error=cudaMemcpy(Agpu, A, sizeof(int) * elements,
cudaMemcpyHostToDevice);
        if(error!=cudaSuccess)
          printf("Error cudaMemcpy A: %s\n", cudaGetErrorString(error));
        error=cudaMemcpy(Bgpu, B, sizeof(int) * elements,
cudaMemcpyHostToDevice);
        if(error!=cudaSuccess)
          printf("Error cudaMemcpy B: %s\n", cudaGetErrorString(error));

        vectorAdd<<<tamGrid,tamBlock>>>(Agpu, Bgpu, Cgpu, elements);

        error=cudaMemcpy(C, Cgpu, sizeof(int) * elements,
cudaMemcpyDeviceToHost);
        if(error!=cudaSuccess)
          printf("Error cudaMemcpy C: %s\n", cudaGetErrorString(error));

        for(int i=0; i<elements; i++){
                printf("%d ", C[i]);
        }
        printf("\n");

        cudaFree(Agpu);
        cudaFree(Bgpu);
        cudaFree(Cgpu);
        cudaDeviceReset();
}
```

Anexo III: Ejemplo SYCL

En este Anexo mostramos el código completo del programa de ejemplo de SYCL:

```
#include <CL/sycl.hpp>

std::vector<float> add(cl::sycl::queue& q, const std::vector<float>& A,
  const std::vector<float>& B){

   std::vector<float> C(A.size());
   cl::sycl::range<1> elements{A.size()};

   {
   cl::sycl::buffer<float> buff_A(A.data(), A.size());
   cl::sycl::buffer<float> buff_B(B.data(), B.size());
   cl::sycl::buffer<float> buff_C(C.data(), C.size());

   q.submit([&](cl::sycl::handler& cgh){
      auto access_A = buff_A.get_access<cl::sycl::access::mode::read>(cgh);
      auto access_B = buff_B.get_access<cl::sycl::access::mode::read>(cgh);
      auto access_C = buff_C.get_access<cl::sycl::access::mode::write>(cgh);

      cgh.parallel_for<class vectorAdd>(elements, [=] (cl::sycl::id<1> tid) {
         access_C[tid] = access_A[tid] + access_B[tid];
      });
   });
   }
   return C;
}

int main(){

        cl::sycl::queue q;

        std::vector<float> A = {1.0, 2.0, 3.0, 4.0, 5.0};
        std::vector<float> B = {1.0, 2.0, 3.0, 4.0, 5.0};

        auto C = add(q, A, B);

        for(int i=0; i<C.size();i++){
                printf("%lf ", C[i]);
        }
}
```

Anexo IV: Ejemplo Celerity

En este Anexo mostramos el código completo del programa de ejemplo de Celerity:

```
#include <CL/sycl.hpp>
#include <celerity/celerity.h>

void add(celerity::distr_queue& q, celerity::buffer<float,1> A,
  celerity::buffer<float,1> B, celerity::buffer<float,1> C){

   cl::sycl::range<1> elements{A.get_range()};

   q.submit([=](celerity::handler& cgh){
      auto one_to_one = celerity::access::one_to_one<1>();
      celerity::accessor access_A{A, cgh, one_to_one, cl::sycl::read_only};
      celerity::accessor access_B{B, cgh, one_to_one, cl::sycl::read_only};
      celerity::accessor access_C{C, cgh, one_to_one, cl::sycl::write_only,
cl::sycl::no_init};

      cgh.parallel_for<class vectorAdd>(elements, [=] (cl::sycl::id<1> tid) {
         access_C[tid] = access_A[tid] + access_B[tid];
      });
   });
}

int main(){
   celerity::distr_queue q;

   std::vector<float> A = {1.0, 2.0, 3.0, 4.0, 5.0};
   std::vector<float> B = {1.0, 2.0, 3.0, 4.0, 5.0};

   celerity::buffer<float,1> buff_A(A.data(), A.size());
   celerity::buffer<float,1> buff_B(B.data(), B.size());
   celerity::buffer<float,1> buff_C(A.size());

   add(q, buff_A, buff_B, buff_C);

   q.submit([=](celerity::handler& cgh){

      auto C = buff_C.get_access<cl::sycl::access::mode::read,
        cl::sycl::access::target::host_buffer>(cgh, celerity::access::all<1>());

      cgh.host_task(celerity::on_master_node, [=]() {
         for(int i=0; i<A.size(); i++)
            printf("%lf\n", C[i]);
        });
   });
}
```

Anexo V: Código del programa Stencil

En este Anexo mostramos el código completo de nuestro programa Stencil, desarrollado en Celerity:

```
#include <iostream>
#include <chrono>
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <CL/sycl.hpp>
#include <celerity/celerity.h>

//Funcion para imprimir la ayuda
void printHelp(){
 printf("\njacobi -r <rows> [-c <columns> -d <depth> -s <numIter> -t
<stencilType>] [-h for help]\n");
 printf("Stencil code for 1D, 2D and 3D matrix\n");
 printf("Use -r -c and -d to determine the rows, columns and depth of
the matrix. -s is the number for iterations and -t is the type of
stencil (default type is 1). The list with the different types of
stencil is displayed at the end.\n");
 printf("For 1D array, use -r <rows> option with -s and -t\n");
 printf("For 2D array, use -r <rows> and -c <columns> options with -s
and -t\n");
 printf("For 3D array, use -r <rows>, -c <columns> and -d <depth>
options with -s and -t\n\n");
 printf("Stencil types supported:\n\n");
 printf("1D stencil:\n\n");
 printf("-t 1: 1D, 2 points compact { 1, X, 1 }\n\n");
 printf("-t 2: 1D, 4 points compact { 0.5, 1, X, 1, 0.5}\n\n");
 printf("2D stencil:\n\n");
 printf("-t 1: 2D, 4 points compact (Jacobi)\n");
 printf("{ 0, 1, 0,\n");
 printf("  1, X, 1,\n");
 printf("  0, 1, 0 }\n\n");
 printf("-t 2: 2D, 9 points compact\n");
 printf("{ 1, 4, 1,\n");
 printf("  4, X, 4,\n");
 printf("  1, 4, 1 }\n\n");
 printf("-t 3: 2D, 9 points non-compact\n");
 printf("{ 0, 0, 1, 0, 0,\n");
 printf("  0, 0, 4, 0, 0,\n");
 printf("  1, 4, X, 4, 1,\n");
 printf("  0, 0, 4, 0, 0,\n");
```

```
 printf("  0, 0, 1, 0, 0 }\n\n");
 printf("-t 4: 2D, 5 points asymmetric non-compact\n");
 printf("{ 0, 0,   1,\n");
 printf("  0, 0.5, 2,\n");
 printf("  1, 2,   X }\n\n");
 printf("3D stencil:\n\n");
 printf("-t 1: 3D, 27 points compact\n");
 printf("{ 1, 1, 1,    1, 1, 1,    1, 1, 1,\n");
 printf("  1, 1, 1,    1, 1, 1,    1, 1, 1,\n");
 printf("  1, 1, 1,    1, 1, 1,    1, 1, 1  }\n\n");
}

//Funcion de inicializacion de la matriz
template <typename T, int dim>
void initialization(celerity::distr_queue& q, celerity::buffer<T,dim>
mat, celerity::buffer<T,dim> copy, long rows, int columns, int depth,
int type){

  cl::sycl::range<dim> tam{mat.get_range()};

  q.submit([=](celerity::handler& cgh){
  auto one_to_one=celerity::access::one_to_one<dim>();
  celerity::accessor writemat{mat, cgh, one_to_one,
cl::sycl::write_only, cl::sycl::no_init};
  celerity::accessor writecopy{copy, cgh, one_to_one,
cl::sycl::write_only, cl::sycl::no_init};
  cgh.parallel_for<class matInit>(tam,[=](cl::sycl::id<dim> tid){
   if(dim==2){
    int row=tid[0];
    int col=tid[1];
    if(row==0){ writemat[tid]=1;      writecopy[tid]=1;}
    else if(row==rows-1){ writemat[tid]=2;    writecopy[tid]=2;}
    else if(col==0){ writemat[tid]=3; writecopy[tid]=3;}
    else if(col==columns-1){ writemat[tid]=4; writecopy[tid]=4;}
    else{ writemat[tid]=0.0;   writecopy[tid]=0.0;}
    if(type==3 || type==4){
     if(row==1){ writemat[tid]=1;     writecopy[tid]=1;}
     else if(row==rows-2){ writemat[tid]=2;   writecopy[tid]=2;}
     else if(col==1 && row!=0 && row!=rows-1){ writemat[tid]=3;
        writecopy[tid]=3;}
     else if(col==columns-2 && row!=0 && row!=rows-1){
       writemat[tid]=4; writecopy[tid]=4;}
    }
   }
   if(dim==1){
    int element=tid[0];
    if(element==0){ writemat[tid]=1;  writecopy[tid]=1;}
    else if(element==rows-1){ writemat[tid]=2;       writecopy[tid]=2;}
    else{ writemat[tid]=0.0;   writecopy[tid]=0.0;}
```

```
    if(type==2){
     if(element==1){ writemat[tid]=1; writecopy[tid]=1;}
     else if(element==rows-2){ writemat[tid]=2;       writecopy[tid]=2;}
    }
   }
   if(dim==3){
    int row=tid[0];
    int col=tid[1];
    int dep=tid[2];
    if(row==0){ writemat[tid]=1;       writecopy[tid]=1;}
    else if(row==rows-1){ writemat[tid]=2;    writecopy[tid]=2;}
    else if(col==0){ writemat[tid]=3; writecopy[tid]=3;}
    else if(col==columns-1){ writemat[tid]=4; writecopy[tid]=4;}
    else if(dep==0){ writemat[tid]=5; writecopy[tid]=5;}
    else if(dep==depth-1){ writemat[tid]=6;   writecopy[tid]=6;}
    else{ writemat[tid]=0.0;   writecopy[tid]=0.0;}
   }
  });
 });
}
template <typename T>
void compute2D1(celerity::distr_queue& q, celerity::buffer<T,2> mat,
celerity::buffer<T,2> copy, long rows, int columns){

 cl::sycl::range<2> tam{mat.get_range()};

 q.submit([=](celerity::handler& cgh){
  auto one_to_one=celerity::access::one_to_one<2>();
  auto neighborhood=celerity::access::neighborhood<2>(1,1);
  celerity::accessor readcopy{copy, cgh, neighborhood,
cl::sycl::read_only};
  celerity::accessor writemat{mat, cgh, one_to_one,
cl::sycl::write_only};
  cgh.parallel_for<class matCompute>(cl::sycl::range<2>(rows-2,columns-
2),cl::sycl::id<2>(1,1),[=](cl::sycl::item<2> tid){
   writemat[tid]=(readcopy[{tid[0]-1,tid[1]}] +
readcopy[{tid[0]+1,tid[1]}] + readcopy[{tid[0],tid[1]-1}] +
readcopy[{tid[0],tid[1]+1}]) /4;
  });
 });
}
template <typename T>
void compute2D2(celerity::distr_queue& q, celerity::buffer<T,2> mat,
celerity::buffer<T,2> copy, long rows, int columns){

 cl::sycl::range<2> tam{mat.get_range()};

 q.submit([=](celerity::handler& cgh){
  auto one_to_one=celerity::access::one_to_one<2>();
```

```
  auto neighborhood=celerity::access::neighborhood<2>(1,1);
  celerity::accessor readcopy{copy, cgh, neighborhood,
cl::sycl::read_only};
  celerity::accessor writemat{mat, cgh, one_to_one,
cl::sycl::write_only};
  cgh.parallel_for<class matCompute>(cl::sycl::range<2>(rows-2,columns-
2),cl::sycl::id<2>(1,1),[=](cl::sycl::item<2> tid){
  writemat[tid]=(readcopy[{tid[0]-1,tid[1]}]*4 +
readcopy[{tid[0]+1,tid[1]}]*4 + readcopy[{tid[0],tid[1]-1}]*4 +
readcopy[{tid[0],tid[1]+1}]*4 +
     readcopy[{tid[0]-1,tid[1]-1}] + readcopy[{tid[0]-1,tid[1]+1}] +
readcopy[{tid[0]+1,tid[1]-1}] + readcopy[{tid[0]+1,tid[1]+1}]) /20;
  });
 });
}

template <typename T>
void compute2D3(celerity::distr_queue& q, celerity::buffer<T,2> mat,
celerity::buffer<T,2> copy, long rows, int columns){

 cl::sycl::range<2> tam{mat.get_range()};

 q.submit([=](celerity::handler& cgh){
  auto one_to_one=celerity::access::one_to_one<2>();
  auto neighborhood=celerity::access::neighborhood<2>(2,2);
  celerity::accessor readcopy{copy, cgh, neighborhood,
cl::sycl::read_only};
  celerity::accessor writemat{mat, cgh, one_to_one,
cl::sycl::write_only};
  cgh.parallel_for<class matCompute>(cl::sycl::range<2>(rows-4,columns-
4),cl::sycl::id<2>(2,2),[=](cl::sycl::item<2> tid){
   writemat[tid]=(readcopy[{tid[0]-1,tid[1]}]*4 +
readcopy[{tid[0]+1,tid[1]}]*4 + readcopy[{tid[0],tid[1]-1}]*4 +
readcopy[{tid[0],tid[1]+1}]*4 +
     readcopy[{tid[0]-2,tid[1]}] + readcopy[{tid[0]+2,tid[1]}] +
readcopy[{tid[0],tid[1]-2}] + readcopy[{tid[0],tid[1]+2}]) /20;
  });
 });
}

template <typename T>
void compute2D4(celerity::distr_queue& q, celerity::buffer<T,2> mat,
celerity::buffer<T,2> copy, long rows, int columns){

 cl::sycl::range<2> tam{mat.get_range()};

 q.submit([=](celerity::handler& cgh){
  auto one_to_one=celerity::access::one_to_one<2>();
  auto neighborhood=celerity::access::neighborhood<2>(2,2);
```

```
  celerity::accessor readcopy{copy, cgh, neighborhood,
cl::sycl::read_only};
  celerity::accessor writemat{mat, cgh, one_to_one,
cl::sycl::write_only};
  cgh.parallel_for<class matCompute>(cl::sycl::range<2>(rows-4,columns-
4),cl::sycl::id<2>(2,2),[=](cl::sycl::item<2> tid){
   writemat[tid]=(readcopy[{tid[0]-2,tid[1]}] + readcopy[{tid[0]-
1,tid[1]-1}]*0.5 + readcopy[{tid[0]-1,tid[1]}]*2 +
     readcopy[{tid[0],tid[1]-2}] + readcopy[{tid[0],tid[1]-1}]*2) /6.5;
  });
 });
}

template <typename T>
void compute1D1(celerity::distr_queue& q, celerity::buffer<T,1> mat,
celerity::buffer<T,1> copy, long elements){

 cl::sycl::range<1> tam{mat.get_range()};

 q.submit([=](celerity::handler& cgh){
  auto one_to_one=celerity::access::one_to_one<1>();
  auto neighborhood=celerity::access::neighborhood<1>(1);
  celerity::accessor readcopy{copy, cgh, neighborhood,
cl::sycl::read_only};
  celerity::accessor writemat{mat, cgh, one_to_one,
cl::sycl::write_only};
  cgh.parallel_for<class matCompute>(cl::sycl::range<1>(elements-
2),cl::sycl::id<1>(1),[=](cl::sycl::item<1> tid){
   writemat[tid]=(readcopy[{tid[0]-1}] + readcopy[{tid[0]+1}]) /2;
  });
 });
}

template <typename T>
void compute1D2(celerity::distr_queue& q, celerity::buffer<T,1> mat,
celerity::buffer<T,1> copy, long elements){

 cl::sycl::range<1> tam{mat.get_range()};

 q.submit([=](celerity::handler& cgh){
  auto one_to_one=celerity::access::one_to_one<1>();
  auto neighborhood=celerity::access::neighborhood<1>(2);
  celerity::accessor readcopy{copy, cgh, neighborhood,
cl::sycl::read_only};
  celerity::accessor writemat{mat, cgh, one_to_one,
cl::sycl::write_only};
  cgh.parallel_for<class matCompute>(cl::sycl::range<1>(elements-
4),cl::sycl::id<1>(2),[=](cl::sycl::item<1> tid){
```

```
   writemat[tid]=(readcopy[{tid[0]-2}]*0.5 + readcopy[{tid[0]-1}] +
readcopy[{tid[0]+1}] + readcopy[{tid[0]+2}]*0.5) /3;
  });
 });
}

//Funcion de computo de la matriz con 3 dimensiones
template <typename T>
void compute3D(celerity::distr_queue& q, celerity::buffer<T,3> mat,
celerity::buffer<T,3> copy, long rows, int columns, int depth){

 cl::sycl::range<3> tam{mat.get_range()};

 q.submit([=](celerity::handler& cgh){
  auto one_to_one=celerity::access::one_to_one<3>();
  auto neighborhood=celerity::access::neighborhood<3>(1,1,1);
  celerity::accessor rc{copy, cgh, neighborhood, cl::sycl::read_only};
  celerity::accessor writemat{mat, cgh, one_to_one,
cl::sycl::write_only};
  cgh.parallel_for<class matCompute>(cl::sycl::range<3>(rows-2,
columns-2, depth-2),cl::sycl::id<3>(1,1,1),[=](cl::sycl::item<3> tid){
   writemat[tid]=(rc[{tid[0],tid[1],tid[2]}] +
    rc[{tid[0]-1,tid[1],tid[2]}] + rc[{tid[0]+1,tid[1],tid[2]}] +
rc[{tid[0],tid[1]-1,tid[2]}] + rc[{tid[0],tid[1]+1,tid[2]}] +
rc[{tid[0],tid[1],tid[2]-1}] +
    rc[{tid[0],tid[1],tid[2]+1}] +
    rc[{tid[0]-1,tid[1]-1,tid[2]}] + rc[{tid[0]-1,tid[1]+1,tid[2]}] +
rc[{tid[0]-1,tid[1],tid[2]-1}] + rc[{tid[0]-1,tid[1],tid[2]+1}] +
rc[{tid[0],tid[1]-1,tid[2]-1}] +
    rc[{tid[0],tid[1]-1,tid[2]+1}] +
    rc[{tid[0],tid[1]-1,tid[2]-1}] + rc[{tid[0],tid[1]+1,tid[2]+1}] +
rc[{tid[0]+1,tid[1]-1,tid[2]}] + rc[{tid[0]+1,tid[1]+1,tid[2]}] +
rc[{tid[0]+1,tid[1],tid[2]-1}] +
    rc[{tid[0]+1,tid[1],tid[2]+1}] +
    rc[{tid[0]-1,tid[1]-1,tid[2]-1}] + rc[{tid[0]-1,tid[1]-1,tid[2]+1}]
+ rc[{tid[0]-1,tid[1]+1,tid[2]-1}] + rc[{tid[0]-1,tid[1]+1,tid[2]+1}] +
rc[{tid[0]+1,tid[1]-1,tid[2]-1}] +
    rc[{tid[0]+1,tid[1]-1,tid[2]+1}] + rc[{tid[0]+1,tid[1]+1,tid[2]-1}]
+ rc[{tid[0]+1,tid[1]+1,tid[2]+1}]) /27;
  });
 });
}
int main(int argc, char *argv[]) {

 long rows=0;;
 int columns=0;
 int depth=0;
 int stages=0;
 int type=1;
```

```
int dim=0;
int c;

celerity::detail::runtime::init(&argc, &argv);
int num_procs;
MPI_Comm_size( MPI_COMM_WORLD, &num_procs );

celerity::distr_queue q;

while ((c = getopt (argc, argv, "r:c:d:s:t:h")) != -1){
 switch (c){
  case 'r':
   rows = atol(optarg);
  break;
  case 'c':
   columns = atoi(optarg);
  break;
  case 'd':
   depth = atoi(optarg);
  break;
  case 's':
   stages = atoi(optarg);
  break;
  case 't':
   type = atoi(optarg);
  break;
  case 'h':
   q.submit([=](celerity::handler& cgh) {
    cgh.host_task(celerity::on_master_node, [=]() {
     printHelp();
    });
   });
  return 0;
  break;
  default:
   abort ();
  }
}

if(rows<=0) dim=0;
else if(columns==0 || columns==1) dim=1;
else if(depth==0 || depth==1) dim=2;
else dim=3;
if(columns<0 || depth<0) dim=0;
if(dim==1 && (type<=0 || type>=3)) dim=0;
if(dim==2 && (type<=0 || type>=5)) dim=0;
if(dim==3 && type!=1) type=1;
```

```
 celerity::experimental::bench::log_user_config({{"rows",
std::to_string(rows)}, {"columns", std::to_string(columns)}, {"depth",
std::to_string(depth)}, {"stages", std::to_string(stages)}, {"type",
std::to_string(type)}});

switch (dim){
  case 1:{
   auto tam = cl::sycl::range<1>(rows);
   celerity::buffer<float,1> buff_mat(tam);
   celerity::buffer<float,1> buff_copy(tam);

   // Init
   initialization(q,buff_mat,buff_copy,rows,columns,depth,type);

   //Start timer
   q.slow_full_sync();
   celerity::experimental::bench::begin("compute");
   auto start = std::chrono::high_resolution_clock::now();

   // Compute
   int stage;
   for ( stage=0; stage<stages; stage++ ) {
    //Update copy
    std::swap(buff_mat, buff_copy);

    // Compute iteration
    compute1D1(q,buff_mat,buff_copy,rows);
   }

   //End timer
   q.slow_full_sync();
   auto end = std::chrono::high_resolution_clock::now();
   celerity::experimental::bench::end("compute");

   int time =
std::chrono::duration_cast<std::chrono::milliseconds>(end-
start).count();
   printf("\nTime: %8d ms\n\n",time);

   // Write
   /*q.submit(celerity::allow_by_ref,[&](celerity::handler& cgh) {
        auto result = buff_mat.get_access<cl::sycl::access::mode::read,
cl::sycl::access::target::host_buffer>(cgh,
celerity::access::all<1>());
        cgh.host_task(celerity::on_master_node, [=]() {
      for ( size_t i=0; i<rows; i++ ){
       printf("%014.4lf ", result[{i}]);
      }
      printf("\n");
```

```
        });
      });*/
  }break;
  case 2:{
   auto tam = cl::sycl::range<2>(rows, columns);
   celerity::buffer<float,2> buff_mat(tam);
   celerity::buffer<float,2> buff_copy(tam);

   // Init
   initialization(q,buff_mat,buff_copy,rows,columns,depth,type);

   //Start timer
   q.slow_full_sync();
   celerity::experimental::bench::begin("compute");
   auto start = std::chrono::high_resolution_clock::now();

   // Compute
   int stage;
   for ( stage=0; stage<stages; stage++ ) {
    //Update copy
    std::swap(buff_mat, buff_copy);

    // Compute iteration
    compute2D1(q,buff_mat,buff_copy,rows,columns);
   }

   //End timer
   q.slow_full_sync();
   auto end = std::chrono::high_resolution_clock::now();
   celerity::experimental::bench::end("compute");

   int time =
std::chrono::duration_cast<std::chrono::milliseconds>(end-
start).count();
   printf("\nTime: %8d ms\n\n",time);

   // Write
   /*q.submit(celerity::allow_by_ref,[&](celerity::handler& cgh) {
        auto result = buff_mat.get_access<cl::sycl::access::mode::read,
cl::sycl::access::target::host_buffer>(cgh,
celerity::access::all<2>());
        cgh.host_task(celerity::on_master_node, [=]() {
     //FILE* fichero;
     //fichero = fopen("w", "w+");
     for ( size_t i=0; i<rows; i++ ){
      for ( size_t j=0; j<columns; j++ ){
       printf("%014.4lf ", result[{i,j}]);
       //fprintf(fichero,"%014.4lf ", result[{i,j}]);
      }
```

```
      //fprintf(fichero,"\n");
      printf("\n");
     }
        });
     });*/
  }break;
  case 3:{
   //struct timeval initTime;
   auto tam = cl::sycl::range<3>(rows, columns, depth);
   celerity::buffer<float,3> buff_mat(tam);
   celerity::buffer<float,3> buff_copy(tam);

   // Init
   initialization(q,buff_mat,buff_copy,rows,columns,depth,type);

   //Start timer
   q.slow_full_sync();
   celerity::experimental::bench::begin("compute");
   auto start = std::chrono::high_resolution_clock::now();

   // Compute
   int stage;
   for ( stage=0; stage<stages; stage++ ) {
    //Update copy
    //copy(q,buff_mat,buff_copy);
    std::swap(buff_mat, buff_copy);

    // Compute iteration
    compute3D(q,buff_mat,buff_copy,rows,columns,depth);
   }

   //End timer
   q.slow_full_sync();
   auto end = std::chrono::high_resolution_clock::now();
   celerity::experimental::bench::end("compute");

   int time =
std::chrono::duration_cast<std::chrono::milliseconds>(end-
start).count();
   printf("\nTime: %8d ms\n\n",time);

   // Write
   /*q.submit(celerity::allow_by_ref,[&](celerity::handler& cgh) {
        auto result = buff_mat.get_access<cl::sycl::access::mode::read,
cl::sycl::access::target::host_buffer>(cgh,
celerity::access::all<3>());
        cgh.host_task(celerity::on_master_node, [=]() {
      for ( size_t i=0; i<rows; i++ ){
       for ( size_t j=0; j<columns; j++ ){
```

```
        for ( size_t k=0; k<depth; k++ ){
         printf("%014.4lf ", result[{i,j,k}]);
        }
        printf("\n");
       }
       printf("\n");
      }
        });
     });*/
  }break;
  default:
   q.submit([=](celerity::handler& cgh) {
    cgh.host_task(celerity::on_master_node, [=]() {
     printf("Only 1, 2 or 3 dimensions\n");
     fprintf( stderr, "Usage: %s -r <rows> -s <numIter> -t
<stencilType>[ -c <columns> -d <depth>] [-h for help]\n", argv[0] );
    });
   });
  break;
 }

 //End
 return 0;
}
```

Anexo VI: Tablas de resultados Celerity

En este Anexo mostramos todas las tablas con los resultados obtenidos, en segundos, de las todas las pruebas ejecutadas con Celerity:

Pruebas escalabilidad fuerte (Celerity), 2D 4 puntos compacto (Jacobi)						
GPU	1	2	3	4	5	Media
1 Manticore	13.857	13.861	13.864	13.861	13.859	13.860
2 Manticore	7.354	7.355	7.361	7.371	7.360	7.360
1 Medusa X	15.099	15.110	15.094	15.087	15.144	15.107
1 Medusa B	21.252	21.239	21.245	21.276	21.233	21.249
2 Medusa XB	10.973	10.925	10.956	10.974	10.930	10.952
2 Medusa BB	11.211	11.260	11.267	11.214	11.235	11.237
3 Medusa XBB	7.690	7.632	7.645	7.697	7.646	7.662
3 Medusa BBB	8.633	8.674	8.639	8.672	8.651	8.654
4 Medusa	6.517	6.432	6.500	6.579	6.490	6.504
1 Hydra	21.649	21.641	21.645	21.674	21.644	21.651
2 Hydra	11.308	11.312	11.323	11.308	11.315	11.313
3 Hydra	10.292	10.312	10.286	10.279	10.305	10.295
4 Hydra	8.198	8.226	8.191	8.222	8.217	8.211
4 (2) Manticore + (2) Medusa	5.571	5.589	5.622	5.640	5.636	5.612
8 (4) Hydra + (4) Medusa	4.559	4.542	4.557	4.541	4.558	4.551

Figura 55: Tabla de resultados para las pruebas de escalabilidad fuerte en el Stencil 2D 4 puntos compacto (Jacobi) con Celerity

Pruebas escalabilidad débil (Celerity), 2D 4 puntos compacto (Jacobi)						
GPU	1	2	3	4	5	Media
1 Manticore	13.857	13.861	13.864	13.861	13.859	13.860
2 Manticore	14.367	14.355	14.364	14.358	14.366	14.362
1 Medusa X	15.099	15.110	15.094	15.087	15.144	15.107
1 Medusa B	21.252	21.239	21.245	21.276	21.233	21.249
2 Medusa XB	21.652	21.629	21.734	21.643	21.663	21.664
2 Medusa BB	22.238	22.212	22.206	22.230	22.218	22.221
3 Medusa XBB	22.400	22.247	22.289	22.345	22.234	22.303
3 Medusa BBB	25.914	25.948	25.932	25.945	9.937	22.735
4 Medusa	26.079	26.077	26.089	26.084	26.075	26.081
1 Hydra	21.649	21.641	21.645	21.674	21.644	21.651
2 Hydra	22.270	22.242	22.289	22.271	22.265	22.267
3 Hydra	30.242	30.284	30.281	30.276	30.238	30.264
4 Hydra	30.725	30.702	30.709	30.713	30.788	30.727
4 (2) Manticore + (2) Medusa	21.734	21.746	21.759	21.744	21.736	21.744
8 (4) Hydra + (4) Medusa	30.759	30.856	30.639	30.779	30.678	30.742

Figura 56: Tabla de resultados para las pruebas de escalabilidad débil en el Stencil 2D 4 puntos compacto (Jacobi) con Celerity

Pruebas escalabilidad fuerte (Celerity), 2D 9 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	15.329	15.246	15.327	15.329	15.327	15.312
2 Manticore	8.112	8.137	8.123	8.101	8.106	8.116
1 Medusa X	23.528	23.390	23.422	23.440	23.435	23.443
1 Medusa B	37.484	37.455	37.475	37.497	37.472	37.477
2 Medusa XB	19.058	18.830	19.070	18.945	19.056	18.992
2 Medusa BB	19.701	19.699	19.702	19.662	19.735	19.700
3 Medusa XBB	13.215	13.176	13.154	13.162	13.193	13.180
3 Medusa BBB	14.324	14.324	14.354	14.320	14.331	14.331
4 Medusa	10.848	10.928	10.876	10.856	10.925	10.887
1 Hydra	38.603	38.608	38.584	38.644	38.568	38.601
2 Hydra	19.719	19.744	19.732	19.746	19.743	19.737
3 Hydra	18.048	18.108	18.095	18.049	18.051	18.070
4 Hydra	14.019	14.020	14.100	14.025	14.103	14.053
4 (2) Manticore + (2) Medusa	9.770	9.710	9.773	9.699	9.788	9.748
8 (4) Hydra + (4) Medusa	7.441	7.455	7.468	7.432	7.489	7.457

Figura 57: Tabla de resultados para las pruebas de escalabilidad fuerte en el Stencil 2D 9 puntos compacto con Celerity

Pruebas escalabilidad débil (Celerity), 2D 9 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	15.329	15.246	15.327	15.329	15.327	15.312
2 Manticore	15.835	15.823	15.829	15.847	15.834	15.834
1 Medusa X	23.528	23.390	23.422	23.440	23.435	23.443
1 Medusa B	37.484	37.455	37.475	37.497	37.472	37.477
2 Medusa XB	37.781	37.818	37.835	37.794	37.820	37.810
2 Medusa BB	39.193	39.235	39.176	39.155	39.232	39.198
3 Medusa XBB	39.212	39.231	39.234	39.218	39.222	39.223
3 Medusa BBB	43.157	43.134	43.157	43.137	43.164	43.150
4 Medusa	43.146	43.164	43.149	43.173	43.154	43.157
1 Hydra	38.603	38.608	38.584	38.644	38.568	38.601
2 Hydra	39.024	39.031	39.003	39.052	39.108	39.044
3 Hydra	53.397	53.440	53.422	53.467	53.422	53.430
4 Hydra	53.808	53.819	53.769	53.907	53.790	53.819
4 (2) Manticore + (2) Medusa	37.959	37.923	38.001	37.998	37.931	37.962
8 (4) Hydra + (4) Medusa	53.860	53.863	53.945	53.764	53.832	53.853

Figura 58: Tabla de resultados para las pruebas de escalabilidad débil en el Stencil 2D 9 puntos compacto con Celerity

Pruebas escalabilidad fuerte (Celerity), 2D 9 puntos no compacto						
GPU	**1**	**2**	**3**	**4**	**5**	**Media**
1 Manticore	17.217	17.186	17.219	17.117	17.214	17.191
2 Manticore	9.240	9.221	9.267	9.246	9.220	9.239
1 Medusa X	25.426	25.435	25.445	25.431	25.554	25.458
1 Medusa B	33.937	34.048	33.956	33.992	33.925	33.972
2 Medusa XB	17.878	17.919	17.911	17.874	17.901	17.897
2 Medusa BB	18.478	18.426	18.420	18.473	18.453	18.450
3 Medusa XBB	12.483	12.438	12.464	12.445	12.432	12.452
3 Medusa BBB	13.746	13.724	13.734	13.721	13.756	13.736
4 Medusa	10.331	10.267	10.246	10.300	10.234	10.276
1 Hydra	34.946	34.924	34.934	34.946	34.929	34.936
2 Hydra	18.288	18.399	18.332	18.310	18.291	18.324
3 Hydra	16.792	16.736	16.745	16.791	16.738	16.760
4 Hydra	13.267	13.269	13.245	13.277	13.270	13.266
4 (2) Manticore + (2) Medusa	9.166	9.111	9.134	9.127	9.163	9.140
8 (4) Hydra + (4) Medusa	7.186	7.182	7.176	7.189	7.193	7.185

Figura 59: Tabla de resultados para las pruebas de escalabilidad fuerte en el Stencil 2D 9 puntos no compacto con Celerity

Pruebas escalabilidad débil (Celerity), 2D 9 puntos no compacto						
GPU	**1**	**2**	**3**	**4**	**5**	**Media**
1 Manticore	17.217	17.186	17.219	17.117	17.214	17.191
2 Manticore	17.858	17.871	17.886	17.871	17.887	17.875
1 Medusa X	25.426	25.435	25.445	25.431	25.554	25.458
1 Medusa B	33.937	34.048	33.956	33.992	33.925	33.972
2 Medusa XB	35.638	35.553	35.523	35.610	35.574	35.580
2 Medusa BB	36.607	36.613	36.628	36.610	36.615	36.615
3 Medusa XBB	36.589	36.598	36.594	36.582	36.599	36.592
3 Medusa BBB	41.078	41.004	41.089	41.014	41.047	41.046
4 Medusa	40.842	40.645	40.767	40.743	40.821	40.764
1 Hydra	34.946	34.924	34.934	34.946	34.929	34.936
2 Hydra	36.413	36.050	36.312	36.051	36.213	36.208
3 Hydra	49.343	49.350	49.364	49.340	49.352	49.350
4 Hydra	50.076	50.055	50.064	50.054	50.065	50.063
4 (2) Manticore + (2) Medusa	35.514	35.593	35.523	35.586	35.528	35.549
8 (4) Hydra + (4) Medusa	50.001	50.020	50.003	50.023	50.010	50.011

Figura 60: Tabla de resultados para las pruebas de escalabilidad débil en el Stencil 2D 9 puntos no compacto con Celerity

Pruebas escalabilidad fuerte (Celerity), 2D 5 puntos asimétrico no compacto						
GPU	1	2	3	4	5	Media
1 Manticore	16.560	16.526	16.558	16.554	16.530	16.546
2 Manticore	8.926	8.932	8.870	8.938	8.876	8.908
1 Medusa X	74.271	74.424	74.358	74.281	74.411	74.349
1 Medusa B	80.034	79.963	80.069	79.982	80.014	80.012
2 Medusa XB	40.080	40.067	40.112	40.103	40.129	40.098
2 Medusa BB	42.210	42.262	42.245	42.291	42.215	42.245
3 Medusa XBB	28.164	28.215	28.167	28.206	28.174	28.185
3 Medusa BBB	29.084	28.855	28.831	28.841	28.901	28.902
4 Medusa	21.799	21.795	21.793	21.813	21.689	21.778
1 Hydra	83.567	83.574	83.337	83.429	83.623	83.506
2 Hydra	41.717	42.186	42.090	41.985	42.003	41.996
3 Hydra	28.463	28.463	28.587	28.485	28.537	28.507
4 Hydra	21.329	21.327	21.335	21.312	21.325	21.326
4 (2) Manticore + (2) Medusa	20.664	20.411	20.512	20.453	20.329	20.474
8 (4) Hydra + (4) Medusa	11.863	11.720	11.821	11.734	11.712	11.770

Figura 61: Tabla de resultados para las pruebas de escalabilidad fuerte en el Stencil 2D 5 puntos asimétrico no compacto con Celerity

Pruebas escalabilidad débil (Celerity), 2D 5 puntos asimétrico no compacto						
GPU	1	2	3	4	5	Media
1 Manticore	16.560	16.526	16.558	16.554	16.530	16.546
2 Manticore	17.214	17.218	17.218	17.217	17.218	17.217
1 Medusa X	74.271	74.424	74.358	74.281	74.411	74.349
1 Medusa B	80.034	79.963	80.069	79.982	80.014	80.012
2 Medusa XB	80.037	79.912	80.219	79.937	80.061	80.033
2 Medusa BB	84.362	84.444	84.330	84.381	84.412	84.386
3 Medusa XBB	84.403	84.399	84.400	84.394	84.405	84.400
3 Medusa BBB	85.671	85.678	85.679	85.623	85.693	85.669
4 Medusa	85.647	85.651	85.647	85.653	85.650	85.650
1 Hydra	83.567	83.574	83.337	83.429	83.623	83.506
2 Hydra	83.855	84.695	83.956	84.538	84.636	84.336
3 Hydra	85.098	84.697	84.783	84.823	85.005	84.881
4 Hydra	84.749	84.576	84.679	84.791	84.894	84.738
4 (2) Manticore + (2) Medusa	80.064	80.101	80.043	80.075	80.059	80.068
8 (4) Hydra + (4) Medusa	85.718	85.614	85.882	85.656	85.685	85.711

Figura 62: Tabla de resultados para las pruebas de escalabilidad débil en el Stencil 2D 5 puntos asimétrico no compacto con Celerity

Pruebas escalabilidad fuerte (Celerity), 1D 2 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	12.655	12.650	12.654	12.648	12.638	12.649
2 Manticore	6.539	6.531	6.545	6.535	6.528	6.536
1 Medusa X	9.791	9.790	9.789	9.793	9.787	9.790
1 Medusa B	11.871	11.859	11.865	11.871	11.860	11.865
2 Medusa XB	6.066	6.156	6.104	6.073	6.084	6.097
2 Medusa BB	6.243	6.237	6.244	6.238	6.237	6.240
3 Medusa XBB	4.241	4.245	4.246	4.251	4.241	4.245
3 Medusa BBB	4.765	4.643	4.642	4.787	4.674	4.702
4 Medusa	3.435	3.430	3.431	3.430	3.432	3.432
1 Hydra	12.063	12.059	12.054	12.064	12.067	12.061
2 Hydra	6.344	6.347	6.343	6.350	6.344	6.346
3 Hydra	6.032	6.030	6.031	6.035	6.032	6.032
4 Hydra	4.765	4.767	4.766	4.765	4.765	4.766
4 (2) Manticore + (2) Medusa	3.177	3.234	3.215	3.194	3.202	3.204
8 (4) Hydra + (4) Medusa	2.647	2.639	2.629	2.640	2.643	2.640

Figura 63: Tabla de resultados para las pruebas de escalabilidad fuerte en el Stencil 1D 2 puntos compacto con Celerity

Pruebas escalabilidad débil (Celerity), 1D 2 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	12.655	12.650	12.654	12.648	12.647	12.651
2 Manticore	12.910	12.915	12.909	12.919	12.920	12.915
1 Medusa X	9.791	9.790	9.789	9.793	9.787	9.790
1 Medusa B	11.871	11.859	11.865	11.871	11.860	11.865
2 Medusa XB	12.074	12.258	12.099	12.111	12.128	12.134
2 Medusa BB	12.492	12.539	12.502	12.497	12.525	12.511
3 Medusa XBB	12.509	12.499	12.501	12.513	12.489	12.502
3 Medusa BBB	14.363	14.335	14.378	14.330	14.340	14.349
4 Medusa	14.252	14.250	14.267	14.272	14.252	14.259
1 Hydra	12.063	12.059	12.054	12.064	12.067	12.061
2 Hydra	12.555	12.556	12.553	12.560	12.558	12.556
3 Hydra	17.739	17.735	17.741	17.732	17.735	17.736
4 Hydra	17.921	17.943	17.947	17.936	17.929	17.935
4 (2) Manticore + (2) Medusa	12.154	12.258	12.169	12.241	12.203	12.205
8 (4) Hydra + (4) Medusa	17.957	17.956	17.955	17.967	17.954	17.958

Figura 64: Tabla de resultados para las pruebas de escalabilidad débil en el Stencil 1D 2 puntos compacto con Celerity

Pruebas escalabilidad fuerte (Celerity), 1D 4 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	12.717	12.710	12.711	12.717	12.711	12.713
2 Manticore	6.548	6.564	6.547	6.566	6.561	6.557
1 Medusa X	69.383	69.749	69.491	69.518	69.820	69.592
1 Medusa B	60.353	60.234	60.375	60.259	60.354	60.315
2 Medusa XB	34.813	34.712	34.829	34.915	34.819	34.818
2 Medusa BB	31.968	31.939	32.001	31.948	31.929	31.957
3 Medusa XBB	23.255	23.273	23.210	23.267	23.260	23.253
3 Medusa BBB	21.795	21.734	21.802	21.712	21.827	21.774
4 Medusa	17.526	17.521	17.521	17.523	17.530	17.524
1 Hydra	63.563	63.615	63.593	63.624	63.581	63.595
2 Hydra	32.031	32.551	32.259	32.347	32.113	32.260
3 Hydra	21.526	21.532	21.527	21.539	21.534	21.532
4 Hydra	16.219	16.256	16.276	16.287	16.277	16.263
4 (2) Manticore + (2) Medusa	17.543	17.586	17.549	17.599	17.583	17.572
8 (4) Hydra + (4) Medusa	9.008	9.013	9.015	9.007	9.007	9.010

Figura 65: Tabla de resultados para las pruebas de escalabilidad fuerte en el Stencil 1D 4 puntos compacto con Celerity

Pruebas escalabilidad débil (Celerity), 1D 4 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	12.717	12.710	12.711	12.717	12.711	12.713
2 Manticore	12.976	12.997	12.986	12.984	12.996	12.988
1 Medusa X	69.383	69.749	69.491	69.518	69.820	69.592
1 Medusa B	60.353	60.234	60.375	60.259	60.354	60.315
2 Medusa XB	69.680	69.918	69.720	69.940	69.699	69.791
2 Medusa BB	63.857	63.963	63.872	63.954	64.043	63.938
3 Medusa XBB	69.701	69.893	69.921	69.773	69.884	69.834
3 Medusa BBB	64.819	64.805	64.808	64.825	64.812	64.814
4 Medusa	69.897	69.950	69.960	69.902	69.911	69.924
1 Hydra	63.563	63.615	63.593	63.624	63.581	63.595
2 Hydra	64.028	63.912	64.034	63.993	64.012	63.996
3 Hydra	64.105	64.111	64.099	64.115	64.100	64.106
4 Hydra	64.113	64.115	64.111	64.111	64.117	64.113
4 (2) Manticore + (2) Medusa	69.963	69.952	69.969	69.954	69.963	69.960
8 (4) Hydra + (4) Medusa	69.918	70.100	70.110	70.003	69.992	70.025

Figura 66: Tabla de resultados para las pruebas de escalabilidad débil en el Stencil 1D 4 puntos compacto con Celerity

Pruebas escalabilidad fuerte (Celerity), 3D 27 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	35.205	35.188	35.212	35.186	35.173	35.193
2 Manticore	22.988	22.995	22.964	22.951	22.982	22.976
1 Medusa X	109.421	109.398	109.562	109.273	109.368	109.404
1 Medusa B	204.354	203.294	204.211	203.843	203.929	203.926
2 Medusa XB	102.885	102.723	102.941	102.882	102.873	102.861
2 Medusa BB	106.788	106.652	106.859	106.780	106.771	106.770
3 Medusa XBB	72.132	72.101	72.178	72.155	72.133	72.140
3 Medusa BBB	77.856	77.885	77.810	77.852	77.806	77.842
4 Medusa	58.003	58.016	58.142	58.129	58.008	58.060
1 Hydra	230.910	230.952	231.001	230.924	230.903	230.938
2 Hydra	111.342	111.329	111.357	111.347	111.384	111.352
3 Hydra	100.405	100.381	100.406	100.423	100.415	100.406
4 Hydra	80.314	80.312	80.342	80.322	80.310	80.320
4 (2) Manticore + (2) Medusa	52.114	52.459	52.240	52.111	52.236	52.232
8 (4) Hydra + (4) Medusa	47.615	47.612	47.640	47.633	47.619	47.624

Figura 67: Tabla de resultados para las pruebas de escalabilidad fuerte en el Stencil 3D 27 puntos compacto con Celerity

Pruebas escalabilidad débil (Celerity), 3D 27 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	35.205	35.188	35.212	35.186	35.173	35.193
2 Manticore	40.694	40.643	40.650	40.642	40.683	40.662
1 Medusa X	109.421	109.398	109.562	109.273	109.368	109.404
1 Medusa B	204.354	203.294	204.211	203.843	203.929	203.926
2 Medusa XB	205.125	205.127	205.244	205.096	205.133	205.145
2 Medusa BB	221.459	221.452	221.329	221.526	221.470	221.447
3 Medusa XBB	221.386	221.381	221.344	221.398	221.377	221.377
3 Medusa BBB	225.340	225.323	225.360	225.345	225.378	225.349
4 Medusa	226.603	226.733	226.663	226.609	226.650	226.652
1 Hydra	230.910	230.952	231.001	230.924	230.903	230.938
2 Hydra	235.029	235.230	235.111	235.069	235.103	235.108
3 Hydra	296.997	297.004	297.113	296.989	296.999	297.020
4 Hydra	301.999	302.241	302.193	302.006	302.117	302.111
4 (2) Manticore + (2) Medusa	206.078	206.236	206.041	206.155	206.090	206.120
8 (4) Hydra + (4) Medusa	302.952	302.973	302.884	303.002	302.993	302.961

Figura 68: Tabla de resultados para las pruebas de escalabilidad débil en el Stencil 3D 27 puntos compacto con Celerity

Pruebas escalabilidad fuerte 2D (Celerity) - CTE-POWER				
2D Jacobi	1	2	3	Media
1 GPU (30000x30000)	11.526	11.515	11.528	11.523
2 GPUs (30000x30000)	6.374	6.376	6.352	6.367
4 GPUs (30000x30000)	3.863	3.880	3.868	3.870
8 GPUs (30000x30000)	2.736	2.750	2.747	2.744
16 GPUs (30000x30000)	7.098	7.133	7.232	7.154
32 GPUs (30000x30000)	20.814	20.754	20.609	20.726
2D 9 puntos compacto	1	2	3	Media
1 GPU (30000x30000)	12.209	12.174	12.143	12.175
2 GPUs (30000x30000)	7.013	6.720	6.924	6.886
4 GPUs (30000x30000)	4.082	4.046	4.040	4.056
8 GPUs (30000x30000)	2.779	2.740	2.770	2.763
16 GPUs (30000x30000)	7.144	7.109	7.123	7.125
32 GPUs (30000x30000)	20.778	20.624	20.667	20.690
2D 9 puntos no compacto	1	2	3	Media
1 GPU (30000x30000)	12.444	12.473	12.448	12.455
2 GPUs (30000x30000)	6.866	6.880	6.873	6.873
4 GPUs (30000x30000)	4.294	4.229	4.283	4.269
8 GPUs (30000x30000)	2.914	2.908	2.901	2.908
16 GPUs (30000x30000)	7.367	7.376	7.351	7.365
32 GPUs (30000x30000)	20.835	20.764	20.701	20.767
2D 5 puntos asimétrico no compacto	1	2	3	Media
1 GPU (30000x30000)	11.790	11.788	11.799	11.792
2 GPUs (30000x30000)	6.644	6.930	6.777	6.784
4 GPUs (30000x30000)	4.097	4.068	4.056	4.074
8 GPUs (30000x30000)	2.918	2.893	2.892	2.901
16 GPUs (30000x30000)	7.275	7.300	7.291	7.289
32 GPUs (30000x30000)	20.658	20.963	20.776	20.799

Figura 69: Tabla de resultados para las pruebas de escalabilidad fuerte 2D en el CTE-POWER del BSC con Celerity

Pruebas escalabilidad fuerte 3D (Celerity) - CTE-POWER				
3D 27 puntos compacto	1	2	3	Media
1 GPU (950x950x950)	26.632	26.714	26.585	26.644
2 GPUs (950x950x950)	16.252	16.415	16.283	16.317
4 GPUs (950x950x950)	13.087	13.044	13.085	13.072
8 GPUs (950x950x950)	10.687	10.669	10.672	10.676
16 GPUs (950x950x950)	12.412	12.436	12.558	12.469
32 GPUs (950x950x950)	31.078	32.345	32.562	31.995

Figura 70: Tabla de resultados para las pruebas de escalabilidad fuerte 3D en el CTE-POWER del BSC con Celerity

Pruebas escalabilidad débil 2D (Celerity) - CTE-POWER				
2D Jacobi	**1**	**2**	**3**	**Media**
1 GPU (30000x30000)	11.526	11.515	11.528	11.523
2 GPUs (60000x30000)	12.184	12.208	12.205	12.199
4 GPUs (60000x60000)	13.454	13.454	13.536	13.481
8 GPUs (120000x60000)	13.796	13.758	13.783	13.779
16 GPUs (120000x120000)	14.711	14.693	14.701	14.702
32 GPUs (240000x120000)	24.490	24.576	24.532	24.533
2D 9 puntos compacto	**1**	**2**	**3**	**Media**
1 GPU (30000x30000)	12.209	12.174	12.143	12.175
2 GPUs (60000x30000)	13.020	12.921	13.001	12.981
4 GPUs (60000x60000)	14.199	14.198	14.198	14.198
8 GPUs (120000x60000)	14.553	14.442	14.528	14.508
16 GPUs (120000x120000)	15.589	15.538	15.549	15.559
32 GPUs (240000x120000)	24.453	24.432	24.443	24.443
2D 9 puntos no compacto	**1**	**2**	**3**	**Media**
1 GPU (30000x30000)	12.444	12.473	12.448	12.455
2 GPUs (60000x30000)	13.200	13.194	13.203	13.199
4 GPUs (60000x60000)	14.953	14.966	14.951	14.957
8 GPUs (120000x60000)	15.328	15.298	15.316	15.314
16 GPUs (120000x120000)	16.902	16.971	16.952	16.942
32 GPUs (240000x120000)	25.148	25.170	25.175	25.164
2D 5 puntos asimétrico no compacto	**1**	**2**	**3**	**Media**
1 GPU (30000x30000)	11.790	11.788	11.799	11.792
2 GPUs (60000x30000)	12.698	12.756	12.793	12.749
4 GPUs (60000x60000)	14.140	14.125	14.123	14.129
8 GPUs (120000x60000)	14.476	14.453	14.479	14.469
16 GPUs (120000x120000)	16.176	15.936	16.004	16.039
32 GPUs (240000x120000)	25.117	25.374	25.308	25.266

Figura 71: Tabla de resultados para las pruebas de escalabilidad débil 2D en el CTE-POWER del BSC con Celerity

Pruebas escalabilidad débil 3D (Celerity) - CTE-POWER				
3D 27 puntos compacto	**1**	**2**	**3**	**Media**
1 GPU (950x950x950)	26.632	26.714	26.585	26.644
2 GPUs (1197x1197x1197)	31.179	31.244	31.076	31.166
4 GPUs (1508x1508x1508)	42.998	42.823	42.854	42.892
8 GPUs (1900x1900x1900)	53.939	53.962	53.999	53.967
16 GPUs (2394x2394x2394)	74.608	74.149	74.452	74.403
32 GPUs (3016x3016x3016)	106.492	105.863	106.146	106.167

Figura 72: Tabla de resultados para las pruebas de escalabilidad débil 3D en el CTE-POWER del BSC con Celerity

Pruebas escalabilidad fuerte 1D (Celerity) - CTE-POWER				
1D 2 puntos compacto	1	2	3	Media
1 GPU (67000000)	8.567	8.570	8.580	8.572
2 GPUs (67000000)	7.646	7.956	7.781	7.794
4 GPUs (67000000)	8.774	8.639	8.723	8.712
8 GPUs (67000000)	28.434	29.099	29.880	29.138
16 GPUs (67000000)	43.409	43.054	43.059	43.174
32 GPUs (67000000)	130.700	132.247	130.915	131.287
1D 4 puntos compacto	1	2	3	Media
1 GPU (67000000)	8.611	8.589	8.602	8.601
2 GPUs (67000000)	8.069	7.949	7.971	7.996
4 GPUs (67000000)	8.743	8.790	8.749	8.761
8 GPUs (67000000)	17.916	18.160	17.983	18.020
16 GPUs (67000000)	49.659	46.204	48.267	48.043
32 GPUs (67000000)	142.138	130.711	132.592	135.147

Figura 73: Tabla de resultados para las pruebas de escalabilidad fuerte 1D en el CTE-POWER del BSC con Celerity

Pruebas escalabilidad débil 1D (Celerity) - CTE-POWER				
1D 2 puntos compacto	1	2	3	Media
1 GPU (67000000)	8.567	8.570	8.580	8.572
2 GPUs (134000000)	11.624	11.561	11.653	11.613
4 GPUs (268000000)	13.786	13.670	13.680	13.712
8 GPUs (536000000)	19.570	18.183	19.294	19.016
16 GPUs (1072000000)	43.655	44.130	44.547	44.111
32 GPUs (2144000000)	145.694	140.183	144.038	143.305
1D 4 puntos compacto	1	2	3	Media
1 GPU (67000000)	8.611	8.589	8.602	8.601
2 GPUs (134000000)	11.268	11.398	11.352	11.339
4 GPUs (268000000)	13.717	13.723	13.736	13.725
8 GPUs (536000000)	20.178	18.381	19.413	19.324
16 GPUs (1072000000)	46.397	48.176	46.504	47.026
32 GPUs (2144000000)	138.505	140.109	139.103	139.239

Figura 74: Tabla de resultados para las pruebas de escalabilidad débil 1D en el CTE-POWER del BSC con Celerity

Anexo VII: Tablas de resultados Controller+Hitmap

En este Anexo mostramos todas las tablas con los resultados obtenidos, en segundos, de las todas las pruebas ejecutadas con Controller+Hitmap:

Pruebas escalabilidad fuerte (Controller+Hitmap), 2D 4 puntos compacto (Jacobi)						
GPU	1	2	3	4	5	Media
1 Manticore	12.395	12.395	12.395	12.396	12.395	12.395
2 Manticore	6.249	6.250	6.239	6.249	6.234	6.244
1 Medusa X	10.920	10.972	10.950	10.946	10.953	10.948
1 Medusa B	15.823	15.831	15.853	15.829	15.829	15.833
2 Medusa XB	7.967	8.228	7.914	8.033	7.945	8.017
2 Medusa BB	9.116	9.116	9.119	9.123	9.118	9.118
3 Medusa XBB	5.948	5.949	5.965	5.959	5.961	5.956
3 Medusa BBB	6.653	6.638	6.580	6.654	6.654	6.636
4 Medusa	4.570	4.560	4.555	4.537	4.560	4.556
1 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
2 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
3 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 (2) Manticore + (2) Medusa	4.775	4.764	4.770	4.766	4.768	4.769
8 (4) Hydra + (4) Medusa	0.000	0.000	0.000	0.000	0.000	0.000

Figura 75: Tabla de resultados para las pruebas de escalabilidad fuerte en el Stencil 2D 4 puntos compacto (Jacobi) con Controller+Hitmap

Pruebas escalabilidad débil (Controller+Hitmap), 2D 4 puntos compacto (Jacobi)						
GPU	1	2	3	4	5	Media
1 Manticore	12.395	12.395	12.395	12.396	12.395	12.395
2 Manticore	12.917	12.845	12.905	12.807	12.791	12.853
1 Medusa X	10.932	10.982	11.025	11.023	11.027	10.998
1 Medusa B	15.827	15.849	15.992	15.901	15.888	15.891
2 Medusa XB	16.189	16.145	16.171	16.212	16.262	16.196
2 Medusa BB	18.666	18.457	18.452	18.701	18.742	18.604
3 Medusa XBB	17.772	17.787	17.882	17.914	17.803	17.831
3 Medusa BBB	20.139	20.050	20.069	20.089	20.162	20.102
4 Medusa	19.001	19.193	19.078	19.193	19.174	19.128
1 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
2 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
3 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 (2) Manticore + (2) Medusa	18.858	18.827	18.889	18.886	18.881	18.868
8 (4) Hydra + (4) Medusa	0.000	0.000	0.000	0.000	0.000	0.000

Figura 76: Tabla de resultados para las pruebas de escalabilidad débil en el Stencil 2D 4 puntos compacto (Jacobi) con Controller+Hitmap

Pruebas escalabilidad fuerte (Controller+Hitmap), 2D 9 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	13.597	13.596	13.596	13.600	13.595	13.597
2 Manticore	6.905	6.909	6.916	6.920	6.896	6.909
1 Medusa X	20.844	20.844	20.842	20.844	20.843	20.844
1 Medusa B	27.438	27.453	27.449	27.460	27.464	27.453
2 Medusa XB	13.874	14.007	14.047	14.025	13.933	13.977
2 Medusa BB	14.144	14.138	14.318	14.806	14.130	14.307
3 Medusa XBB	10.006	10.067	10.014	10.028	10.026	10.028
3 Medusa BBB	10.882	10.826	10.904	10.883	10.905	10.880
4 Medusa	8.073	8.048	8.143	8.103	7.990	8.072
1 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
2 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
3 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 (2) Manticore + (2) Medusa	8.428	8.251	8.291	8.257	8.253	8.296
8 (4) Hydra + (4) Medusa	0.000	0.000	0.000	0.000	0.000	0.000

Figura 77: Tabla de resultados para las pruebas de escalabilidad fuerte en el Stencil 2D 9 puntos compacto con Controller+Hitmap

Pruebas escalabilidad débil (Controller+Hitmap), 2D 9 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	13.597	13.596	13.596	13.600	13.595	13.597
2 Manticore	13.680	13.681	13.673	13.737	13.683	13.691
1 Medusa X	20.844	20.844	20.842	20.844	20.843	20.844
1 Medusa B	27.438	27.453	27.449	27.460	27.464	27.453
2 Medusa XB	27.785	28.094	27.973	27.674	27.790	27.863
2 Medusa BB	29.121	28.251	28.372	28.576	28.238	28.512
3 Medusa XBB	29.831	29.791	29.820	29.772	29.780	29.799
3 Medusa BBB	32.492	32.620	32.581	32.630	32.569	32.578
4 Medusa	31.327	31.465	31.564	31.467	31.461	31.457
1 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
2 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
3 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 (2) Manticore + (2) Medusa	32.498	32.645	32.633	32.651	32.628	32.611
8 (4) Hydra + (4) Medusa	0.000	0.000	0.000	0.000	0.000	0.000

Figura 78: Tabla de resultados para las pruebas de escalabilidad débil en el Stencil 2D 9 puntos compacto con Controller+Hitmap

Pruebas escalabilidad fuerte (Controller+Hitmap), 2D 9 puntos no compacto						
GPU	1	2	3	4	5	Media
1 Manticore	13.643	13.644	13.646	13.644	13.643	13.644
2 Manticore	6.939	6.940	6.939	6.939	6.943	6.940
1 Medusa X	20.767	20.769	20.768	20.770	20.769	20.769
1 Medusa B	27.317	27.336	27.296	27.345	27.340	27.327
2 Medusa XB	13.786	13.978	13.760	14.069	13.978	13.914
2 Medusa BB	14.155	14.151	14.153	14.155	14.125	14.148
3 Medusa XBB	10.019	10.049	10.023	10.007	10.013	10.022
3 Medusa BBB	10.983	10.964	10.978	10.979	10.976	10.976
4 Medusa	7.976	8.184	8.018	8.150	7.980	8.062
1 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
2 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
3 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 (2) Manticore + (2) Medusa	8.290	8.283	8.283	8.296	8.539	8.338
8 (4) Hydra + (4) Medusa	0.000	0.000	0.000	0.000	0.000	0.000

Figura 79: Tabla de resultados para las pruebas de escalabilidad fuerte en el Stencil 2D 9 puntos no compacto con Controller+Hitmap

Pruebas escalabilidad débil (Controller+Hitmap), 2D 9 puntos no compacto						
GPU	1	2	3	4	5	Media
1 Manticore	13.643	13.644	13.646	13.644	13.643	13.644
2 Manticore	13.741	13.737	13.726	13.822	13.723	13.750
1 Medusa X	20.767	20.769	20.768	20.770	20.769	20.769
1 Medusa B	27.317	27.336	27.296	27.345	27.340	27.327
2 Medusa XB	27.980	27.784	27.974	27.559	27.565	27.773
2 Medusa BB	28.199	28.088	28.108	28.160	28.729	28.257
3 Medusa XBB	29.643	29.620	29.685	29.645	29.689	29.656
3 Medusa BBB	32.775	32.797	32.794	32.811	32.810	32.797
4 Medusa	31.667	31.664	31.575	31.621	31.624	31.630
1 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
2 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
3 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 (2) Manticore + (2) Medusa	32.793	32.922	32.869	32.739	32.827	32.830
8 (4) Hydra + (4) Medusa	0.000	0.000	0.000	0.000	0.000	0.000

Figura 80: Tabla de resultados para las pruebas de escalabilidad débil en el Stencil 2D 9 puntos no compacto con Controller+Hitmap

Pruebas escalabilidad fuerte (Controller+Hitmap), 2D 5 puntos asimétrico no compacto						
GPU	1	2	3	4	5	Media
1 Manticore	12.962	12.964	12.966	12.963	12.966	12.964
2 Manticore	6.598	6.596	6.589	6.593	6.594	6.594
1 Medusa X	12.922	12.920	12.920	12.918	12.919	12.920
1 Medusa B	20.774	20.768	20.767	20.560	20.770	20.728
2 Medusa XB	10.599	10.682	10.720	10.587	10.660	10.649
2 Medusa BB	10.825	10.925	10.828	10.815	10.815	10.842
3 Medusa XBB	7.776	7.779	7.764	7.767	7.778	7.773
3 Medusa BBB	8.494	8.471	8.477	8.479	8.475	8.479
4 Medusa	6.230	6.354	6.217	6.309	6.265	6.275
1 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
2 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
3 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 (2) Manticore + (2) Medusa	6.619	5.994	5.994	5.998	5.994	6.120
8 (4) Hydra + (4) Medusa	0.000	0.000	0.000	0.000	0.000	0.000

Figura 81: Tabla de resultados para las pruebas de escalabilidad fuerte en el Stencil 2D 5 puntos asimétrico no compacto con Controller+Hitmap

Pruebas escalabilidad débil (Controller+Hitmap), 2D 5 puntos asimétrico no compacto						
GPU	1	2	3	4	5	Media
1 Manticore	12.962	12.964	12.966	12.963	12.966	12.964
2 Manticore	13.046	13.048	13.046	13.049	13.054	13.049
1 Medusa X	12.922	12.920	12.920	12.918	12.919	12.920
1 Medusa B	20.774	20.768	20.767	20.560	20.770	20.728
2 Medusa XB	21.157	21.284	21.138	21.038	21.011	21.125
2 Medusa BB	21.598	21.539	21.861	21.467	21.481	21.589
3 Medusa XBB	23.003	22.968	22.983	22.970	22.988	22.982
3 Medusa BBB	25.489	25.471	25.481	25.471	25.487	25.480
4 Medusa	24.479	24.248	24.090	23.975	24.421	24.243
1 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
2 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
3 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 (2) Manticore + (2) Medusa	23.850	23.853	23.858	23.852	23.844	23.852
8 (4) Hydra + (4) Medusa	0.000	0.000	0.000	0.000	0.000	0.000

Figura 82: Tabla de resultados para las pruebas de escalabilidad débil en el Stencil 2D 5 puntos asimétrico no compacto con Controller+Hitmap

Pruebas escalabilidad fuerte (Controller+Hitmap), 1D 2 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	11.607	11.606	11.604	11.608	11.607	11.606
2 Manticore	5.900	5.896	5.896	5.900	5.907	5.900
1 Medusa X	9.827	9.827	9.826	9.828	9.826	9.827
1 Medusa B	11.450	11.489	11.448	11.491	11.489	11.473
2 Medusa XB	5.828	5.758	5.800	5.790	5.847	5.805
2 Medusa BB	5.766	5.894	6.195	5.817	5.868	5.908
3 Medusa XBB	4.458	4.459	4.459	4.468	4.472	4.463
3 Medusa BBB	4.761	4.768	4.780	4.765	4.746	4.764
4 Medusa	3.391	3.431	3.403	3.361	3.293	3.376
1 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
2 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
3 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 (2) Manticore + (2) Medusa	3.565	3.519	3.539	3.507	3.507	3.528
8 (4) Hydra + (4) Medusa	0.000	0.000	0.000	0.000	0.000	0.000

Figura 83: Tabla de resultados para las pruebas de escalabilidad fuerte en el Stencil 1D 2 puntos compacto con Controller+Hitmap

Pruebas escalabilidad débil (Controller+Hitmap), 1D 2 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	11.608	11.606	11.610	11.606	11.606	11.607
2 Manticore	11.625	11.685	11.701	11.701	11.705	11.683
1 Medusa X	9.826	9.826	9.826	9.826	9.826	9.826
1 Medusa B	11.454	11.492	11.486	11.497	11.451	11.476
2 Medusa XB	11.713	12.276	11.852	11.687	11.880	11.882
2 Medusa BB	11.921	11.812	12.190	11.966	12.075	11.993
3 Medusa XBB	13.351	13.349	13.352	13.335	13.351	13.348
3 Medusa BBB	14.595	14.620	14.606	14.612	14.637	14.614
4 Medusa	13.401	13.475	13.481	13.373	13.560	13.458
1 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
2 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
3 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 (2) Manticore + (2) Medusa	14.373	14.482	14.518	14.556	14.524	14.491
8 (4) Hydra + (4) Medusa	0.000	0.000	0.000	0.000	0.000	0.000

Figura 84: Tabla de resultados para las pruebas de escalabilidad débil en el Stencil 1D 2 puntos compacto con Controller+Hitmap

Pruebas escalabilidad fuerte (Controller+Hitmap), 1D 4 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	11.628	11.628	11.628	11.626	11.628	11.628
2 Manticore	5.917	5.943	5.913	5.911	5.914	5.919
1 Medusa X	65.830	66.176	66.181	66.284	66.248	66.144
1 Medusa B	58.446	58.477	58.472	59.563	58.479	58.688
2 Medusa XB	32.958	33.070	32.945	32.897	32.928	32.960
2 Medusa BB	31.206	31.112	31.206	31.441	31.387	31.270
3 Medusa XBB	22.453	22.542	22.452	22.584	22.463	22.499
3 Medusa BBB	21.541	21.551	21.559	21.544	21.690	21.577
4 Medusa	16.717	16.611	16.749	16.785	16.709	16.714
1 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
2 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
3 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 (2) Manticore + (2) Medusa	16.143	16.064	16.134	16.134	16.139	16.123
8 (4) Hydra + (4) Medusa	0.000	0.000	0.000	0.000	0.000	0.000

Figura 85: Tabla de resultados para las pruebas de escalabilidad fuerte en el Stencil 1D 4 puntos compacto con Controller+Hitmap

Pruebas escalabilidad débil (Controller+Hitmap), 1D 4 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	11.627	11.629	11.629	11.629	11.627	11.628
2 Manticore	11.707	11.702	11.707	11.703	11.704	11.705
1 Medusa X	66.526	66.611	66.929	66.550	66.824	66.688
1 Medusa B	58.493	59.471	58.372	59.439	58.436	58.842
2 Medusa XB	69.413	69.739	69.779	69.176	69.534	69.528
2 Medusa BB	62.748	62.852	63.252	63.352	63.072	63.055
3 Medusa XBB	71.082	71.112	71.250	70.927	71.116	71.097
3 Medusa BBB	66.846	66.923	67.501	66.964	68.271	67.301
4 Medusa	70.464	70.188	70.482	70.135	70.339	70.321
1 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
2 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
3 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 (2) Manticore + (2) Medusa	67.995	68.062	68.828	68.543	68.248	68.335
8 (4) Hydra + (4) Medusa	0.000	0.000	0.000	0.000	0.000	0.000

Figura 86: Tabla de resultados para las pruebas de escalabilidad débil en el Stencil 1D 4 puntos compacto con Controller+Hitmap

Pruebas escalabilidad fuerte (Controller+Hitmap), 3D 27 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	37.076	37.077	37.078	37.076	37.077	37.077
2 Manticore	18.553	18.547	18.551	18.550	18.551	18.550
1 Medusa X	82.797	82.919	83.058	83.199	83.122	83.019
1 Medusa B	110.934	111.555	111.372	111.605	111.902	111.473
2 Medusa XB	53.314	53.584	53.468	53.492	53.360	53.444
2 Medusa BB	57.669	57.981	57.778	58.012	58.088	57.906
3 Medusa XBB	38.756	38.789	38.747	38.687	38.593	38.715
3 Medusa BBB	42.344	42.440	42.341	42.418	42.479	42.404
4 Medusa	31.738	31.706	31.617	31.671	31.514	31.649
1 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
2 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
3 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 (2) Manticore + (2) Medusa	31.979	32.201	31.991	31.881	32.082	32.027
8 (4) Hydra + (4) Medusa	0.000	0.000	0.000	0.000	0.000	0.000

Figura 87: Tabla de resultados para las pruebas de escalabilidad fuerte en el Stencil 3D 27 puntos compacto con Controller+Hitmap

Pruebas escalabilidad débil (Controller+Hitmap), 3D 27 puntos compacto						
GPU	1	2	3	4	5	Media
1 Manticore	37.076	37.077	37.078	37.076	37.077	37.077
2 Manticore	37.273	37.285	37.283	37.278	37.289	37.282
1 Medusa X	82.797	82.919	83.058	83.199	83.122	83.019
1 Medusa B	110.934	111.555	111.372	111.605	111.902	111.473
2 Medusa XB	119.232	119.292	119.459	119.255	118.918	119.231
2 Medusa BB	121.942	123.356	121.967	122.088	122.108	122.292
3 Medusa XBB	123.279	123.373	125.032	123.352	123.122	123.632
3 Medusa BBB	126.436	126.448	126.422	126.440	126.443	126.438
4 Medusa	124.936	125.005	124.994	124.820	125.353	125.022
1 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
2 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
3 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 Hydra	0.000	0.000	0.000	0.000	0.000	0.000
4 (2) Manticore + (2) Medusa	125.989	124.922	124.933	125.921	126.026	125.558
8 (4) Hydra + (4) Medusa	0.000	0.000	0.000	0.000	0.000	0.000

Figura 88: Tabla de resultados para las pruebas de escalabilidad débil en el Stencil 3D 27 puntos compacto con Controller+Hitmap

Pruebas escalabilidad fuerte 2D (Controller+Hitmap) - CTE-POWER				
2D Jacobi	1	2	3	Media
1 GPU (30000x30000)	10.458	10.456	10.459	10.457
2 GPUs (30000x30000)	5.335	5.295	5.297	5.309
4 GPUs (30000x30000)	2.974	2.942	3.036	2.984
8 GPUs (30000x30000)	1.633	1.657	1.634	1.641
16 GPUs (30000x30000)	0.986	0.963	0.940	0.963
32 GPUs (30000x30000)	0.938	0.927	0.920	0.928
2D 9 puntos compacto	1	2	3	Media
1 GPU (30000x30000)	10.725	10.722	10.724	10.724
2 GPUs (30000x30000)	5.479	5.438	5.442	5.453
4 GPUs (30000x30000)	3.189	2.997	3.030	3.072
8 GPUs (30000x30000)	1.639	1.687	1.680	1.669
16 GPUs (30000x30000)	1.905	0.983	0.955	1.281
32 GPUs (30000x30000)	0.955	1.153	1.249	1.119
2D 9 puntos no compacto	1	2	3	Media
1 GPU (30000x30000)	10.773	10.733	10.732	10.746
2 GPUs (30000x30000)	5.477	5.435	5.436	5.449
4 GPUs (30000x30000)	3.078	3.208	3.041	3.109
8 GPUs (30000x30000)	1.655	1.623	1.618	1.632
16 GPUs (30000x30000)	2.354	0.997	0.985	1.445
32 GPUs (30000x30000)	1.134	1.051	1.313	1.166
2D 5 puntos asimétrico no compacto	1	2	3	Media
1 GPU (30000x30000)	10.475	10.439	10.441	10.451
2 GPUs (30000x30000)	5.274	5.290	5.293	5.285
4 GPUs (30000x30000)	2.932	2.828	2.936	2.898
8 GPUs (30000x30000)	1.578	1.548	1.561	1.562
16 GPUs (30000x30000)	0.945	0.874	0.963	0.927
32 GPUs (30000x30000)	0.855	0.770	0.860	0.828

Figura 89: Tabla de resultados para las pruebas de escalabilidad fuerte 2D en el CTE-POWER del BSC con Controller+Hitmap

Pruebas escalabilidad fuerte 3D (Controller+Hitmap) - CTE-POWER				
3D 27 puntos compacto	1	2	3	Media
1 GPU (950x950x950)	26.825	26.750	26.724	26.766
2 GPUs (950x950x950)	13.345	13.345	13.355	13.348
4 GPUs (950x950x950)	7.077	7.102	7.005	7.061
8 GPUs (950x950x950)	5.907	5.904	5.867	5.893
16 GPUs (950x950x950)	6.179	6.083	6.102	6.122
32 GPUs (950x950x950)	6.327	6.405	6.249	6.327

Figura 90: Tabla de resultados para las pruebas de escalabilidad fuerte 3D en el CTE-POWER del BSC con Controller+Hitmap

Pruebas escalabilidad débil 2D (Controller+Hitmap) - CTE-POWER				
2D Jacobi	1	2	3	Media
1 GPU (30000x30000)	10.458	10.456	10.459	10.457
2 GPUs (60000x30000)	10.528	10.565	10.530	10.541
4 GPUs (60000x60000)	12.117	12.162	12.027	12.102
8 GPUs (120000x60000)	12.251	12.295	12.536	12.361
16 GPUs (120000x120000)	13.213	13.389	13.113	13.238
32 GPUs (240000x120000)	13.256	13.255	13.338	13.283
2D 9 puntos compacto	1	2	3	Media
1 GPU (30000x30000)	10.725	10.722	10.724	10.724
2 GPUs (60000x30000)	10.825	10.819	10.794	10.813
4 GPUs (60000x60000)	12.445	12.610	12.378	12.478
8 GPUs (120000x60000)	12.424	12.468	12.401	12.431
16 GPUs (120000x120000)	13.578	13.800	13.514	13.631
32 GPUs (240000x120000)	13.962	13.896	13.758	13.872
2D 9 puntos no compacto	1	2	3	Media
1 GPU (30000x30000)	10.773	10.733	10.732	10.746
2 GPUs (60000x30000)	10.804	10.818	10.832	10.818
4 GPUs (60000x60000)	12.985	12.901	13.021	12.969
8 GPUs (120000x60000)	13.030	13.101	13.323	13.151
16 GPUs (120000x120000)	14.795	14.660	14.570	14.675
32 GPUs (240000x120000)	15.165	14.730	14.641	14.845
2D 5 puntos asimétrico no compacto	1	2	3	Media
1 GPU (30000x30000)	10.475	10.439	10.441	10.451
2 GPUs (60000x30000)	10.504	10.512	10.506	10.508
4 GPUs (60000x60000)	12.272	12.007	12.094	12.124
8 GPUs (120000x60000)	12.190	12.274	12.203	12.222
16 GPUs (120000x120000)	13.320	13.294	13.264	13.293
32 GPUs (240000x120000)	13.310	13.573	13.406	13.429

Figura 91: Tabla de resultados para las pruebas de escalabilidad débil 2D en el CTE-POWER del BSC con Controller+Hitmap

Pruebas escalabilidad débil 3D (Controller+Hitmap) - CTE-POWER				
3D 27 puntos compacto	1	2	3	Media
1 GPU (950x950x950)	26.825	26.750	26.724	26.766
2 GPUs (1197x1197x1197)	25.402	25.450	25.423	25.425
4 GPUs (1508x1508x1508)	25.599	25.747	25.569	25.638
8 GPUs (1900x1900x1900)	27.097	27.528	26.985	27.203
16 GPUs (2394x2394x2394)	38.461	39.222	39.651	39.111
32 GPUs (3016x3016x3016)	48.479	48.484	49.102	48.688

Figura 92: Tabla de resultados para las pruebas de escalabilidad débil 3D en el CTE-POWER del BSC con Controller+Hitmap

Pruebas escalabilidad fuerte 1D (Controller+Hitmap) - CTE-POWER				
1D 2 puntos compacto	1	2	3	Media
1 GPU (67000000)	7.975	7.981	7.976	7.977
2 GPUs (67000000)	4.463	4.436	4.443	4.447
4 GPUs (67000000)	3.598	3.599	3.591	3.596
8 GPUs (67000000)	3.660	3.657	3.663	3.660
16 GPUs (67000000)	3.698	3.757	3.744	3.733
32 GPUs (67000000)	3.845	3.829	3.828	3.834
1D 4 puntos compacto	1	2	3	Media
1 GPU (67000000)	7.962	7.946	8.013	7.974
2 GPUs (67000000)	4.433	4.428	4.430	4.431
4 GPUs (67000000)	3.636	3.690	3.626	3.651
8 GPUs (67000000)	3.691	3.708	3.666	3.689
16 GPUs (67000000)	3.802	3.970	3.719	3.830
32 GPUs (67000000)	3.876	3.815	4.007	3.899

Figura 93: Tabla de resultados para las pruebas de escalabilidad fuerte 1D en el CTE-POWER del BSC con Controller+Hitmap

Pruebas escalabilidad débil 1D (Controller+Hitmap) - CTE-POWER				
1D 2 puntos compacto	1	2	3	Media
1 GPU (67000000)	7.975	7.981	7.976	7.977
2 GPUs (134000000)	8.612	8.236	8.673	8.507
4 GPUs (268000000)	8.689	8.681	8.666	8.679
8 GPUs (536000000)	8.746	8.722	8.773	8.747
16 GPUs (1072000000)	8.846	8.803	8.835	8.828
32 GPUs (2144000000)	8.788	8.848	8.842	8.826
1D 4 puntos compacto	1	2	3	Media
1 GPU (67000000)	7.962	7.946	8.013	7.974
2 GPUs (134000000)	8.328	8.576	8.664	8.523
4 GPUs (268000000)	8.670	8.630	8.640	8.647
8 GPUs (536000000)	8.660	8.648	8.694	8.667
16 GPUs (1072000000)	8.782	8.760	8.694	8.745
32 GPUs (2144000000)	8.827	8.745	8.744	8.772

Figura 94: Tabla de resultados para las pruebas de escalabilidad débil 1D en el CTE-POWER del BSC con Controller+Hitmap