Improving the Programmability of Heterogeneous Systems by means of Libraries

Moisés Viñas Buceta

PHD THESIS

2016

UNIVERSIDADE DA CORUÑA

Department of Electronics and Systems
University of A Coruña, Spain

Improving the Programmability of Heterogeneous Systems by means of Libraries

Moisés Viñas Buceta

PhD Thesis

March 2016

PhD Advisors:

Basilio B. Fraguela Rodríguez
Diego Andrade Canosa

PhD Program in Information Technology Research

UNIVERSIDADE DA CORUÑA

Dr. Basilio Bernardo Fraguela Rodríguez
Profesor Titular de Universidad
Dpto. de Electrónica y Sistemas
Universidade de A Coruña

Dr. Diego Andrade Canosa
Profesor Contratado Doctor
Dpto. de Electrónica y Sistemas
Universidade de A Coruña

CERTIFICAN

Que la memoria titulada "*Improving the Programmability of Heterogeneous Systems by means of Libraries*" ha sido realizada por D. Moisés Viñas Buceta bajo nuestra dirección en el Departamento de Electrónica y Sistemas de la Universidade da Coruña, y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Ingeniería Informática con la Mención de Doctor Internacional.

En A Coruña, a 11 de Marzo de 2016

Fdo.: Basilio B. Fraguela Rodríguez
Director de la Tesis Doctoral

Fdo.: Diego Andrade Canosa
Director de la Tesis Doctoral

Fdo.: Moisés Viñas Buceta
Autor de la Tesis Doctoral

Agradecimientos

En primer lugar me gustaría agradecer especialmente el arduo trabajo y la dedicación de mis directores Basilio y Diego, en cuya compañía sin duda alguna, ha sido un honor trabajar. Asimismo agradecer también a Ramón Doallo su valiosa colaboración en el desarrollo de esta Tesis y a todos los compañeros del Grupo de Arquitectura de Computadores por los grandes momentos compartidos tanto dentro del laboratorio como fuera de él.

Agradecimiento especial a mis padres, por su ejemplo y a Cris por ser mi motivación diaria.

I would like to thank the people of the 7th floor of the EWI in Delft for their kindness and friendship during my stay in The Netherlands. Specially to Ana Lucia, Henk and Jie. Again, thanks to all. I have also words of thanks to Zeki Bozkus for his collaboration and contribution to this work.

At last, I am also grateful with the following organisms for funding and supporting this work: the Computer Architecture Group, the Electronic and Systems Department and the University of A Coruña; the Galician Government under the projects Consolidation Program of Competitive Reference Groups with references 2010/06 and GRC2013/055; the Ministry of Economy and Competitiveness of Spain under the projects with references TIN2010-16735, TIN2013-42148-P; the european networks HiPEAC(FP7, refs. ICT-217068 and ICT-287759) and NESUS (COST Action, ref. IC1305) and the Spanish networks CAPAP-H.

Moisés

Resumo

O emprego de dispositivos heteroxéneos coma co-procesadores en entornos de computación de altas prestacións (HPC) medrou ininterrompidamente nos últimos anos debido ás súas excelentes propiedades en termos de rendemento e consumo de enerxía. A maior dispoñibilidade de sistemas HPC híbridos conlevou de forma natural a necesidade de desenrolar ferramentas de programación adecuadas para eles, sendo CUDA e OpenCL as máis amplamente empregadas na actualidade. Desafortunadamente, estas ferramentas son relativamente de baixo nivel, o cal emparellado co maior número de detalles que deben de ser controlados cando se programan aceleradoras, fai da programación destes sistemas mediante elas, moito máis complexa que a programación tradicional de CPUs. Isto levou á proposta de alternativas de máis alto nivel para facilitar a programación de dispositivos heteroxéneos. Esta tesis contribúe neste campo presentando dúas librerías que melloran amplamente a programabilidade de sistemas heteroxéneos en C++, permitindo aos usuarios centrarse no que hai que facer en vez de nas tarefas de baixo nivel. As nosas propostas, a librería *Heterogeneous Programming Library* (HPL) e a librería *Heterogeneous Hierarchically Tiled Arrays* (H^2TA), están deseñadas para nodos con unha ou máis aceleradoras, e para clusters heteroxéneos, respectivamente. Ambas librerías, demostraron ser capaces de incrementar a productividade dos usuarios mellorando a programabilidade dos seus códigos, e ó mesmo tempo, lograr un rendemento semellante ó de solucións de máis baixo nivel.

Abstract

The usage of heterogeneous devices as co-processors in high performance computing (HPC) environments has steadily grown during the last years due to their excellent properties in terms of performance and energy consumption. The larger availability of hybrid HPC systems naturally led to the need to develop suitable programming tools for them, being the most widely used nowadays CUDA and OpenCL. Unfortunately, these tools are relatively low level, which coupled with the large number of details that must be managed when programming accelerators, makes the programming of these systems using them much more complex than that of traditional CPUs. This has led to the proposal of higher level alternatives that facilitate the programming of heterogeneous devices. This thesis contributes to this field presenting two libraries that largely improve the programmability of heterogeneous systems in C++, helping users to focus on what to do rather than on low level tasks. These two libraries, the Heterogeneous Programming Library (HPL) and the Heterogeneous Hierarchically Tiled Arrays (H^2TA), are well suited to nodes with one or more accelerators, and to heterogeneous clusters, respectively. Both libraries have proven to be able to increase the productivity of the users improving the programmability of their codes, and at the same time, achieving performance similar to that of lower level solutions.

Resumen

El empleo de dispositivos heterogéneos como co-procesadores en entornos de computación de altas prestaciones (HPC) ha crecido ininterrumpidamente durante los últimos años debido a sus excelentes propiedades en términos de rendimiento y consumo de energía. La mayor disponibilidad de sistemas HPC híbridos conllevó de forma natural la necesidad de desarrollar herramientas de programación adecuadas para ellos, siendo CUDA y OpenCL las más ampliamente utilizadas en la actualidad. Desafortunadamente, estas herramientas son relativamente de bajo nivel, lo cual emparejado con el mayor número de detalles que han de ser controlados cuando se programan acceleradoras, hacen de la programación de estos sistemas mediante ellas mucho más compleja que la programación tradicional de CPUs. Esto ha llevado a la propuesta de alternativas de más alto nivel para facilitar la programación de dispositivos heterogéneos. Esta tesis contribuye a este campo presentando dos librerías que mejoran ampliamente la programabilidad de sistemas heterogéneos en C++, permitiendo a los usuarios centrarse en lo que hay que hacer en vez de en las tareas de bajo nivel. Nuestras propuestas, la librería *Heterogeneous Programming Library* (HPL) y la librería *Heterogeneous Hierarchically Tiled Arrays* (H^{2}TA), están diseñadas para nodos con una o más aceleradoras, y para clusters heterogéneos, respectivamente. Ambas librerías, han demostrado ser capaces de incrementar la productividad de los usuarios mejorando la programabilidad de sus códigos, y al mismo tiempo, lograr un rendimiento similar al de soluciones de más bajo nivel.

Prólogo

La programación de dispositivos heterogéneos supone un desafío para los usuarios por la necesidad de enfrentarse a nuevas plataformas con nuevas herramientas y lenguajes de programación, además de exigirles especificar y gestionar muchos más aspectos. Así, entre los inconvenientes más notorios con los que ha de lidiar el programador a la hora de manejar un co-procesador, se encuentra el tedioso manejo de dos espacios de memoria separados, el del procesador y el del co-procesador. Este manejo engloba la creación de buffers en ambos espacios y el mantenimiento de la coherencia de memoria de las estructuras de datos mantenidas en ellos, lo cual implica a su vez la disposición razonada de puntos de sincronización en el programa principal para garantizar un correcto resultado. Todo este control se lleva a cabo con nuevos lenguajes de programación que exponen APIs más o menos complejas que dificultan, en algunos casos sobremanera, el desarrollo de aplicaciones con dispositivos heterogéneos. Estas APIs incluyen una cantidad notable de nuevos conceptos muchos de los cuales serán introducidos en esta tesis. Otro punto débil de la computación heterogénea es la gestion de errores pues la inmensa mayoría de los elementos de las APIs producen códigos de error destinados a facilitar la depuración de las aplicaciones heterogéneas. La gestión de estos códigos ha de ser cuidadosa pues una implementación poco rigurosa en cuanto a gestión de errores se refiere, puede suponer horas de trabajo de depuración para los programadores inexpertos.

CUDA de NVIDIA, la cual aúna plataforma y lenguaje de programación, es una de estas soluciones. Está diseñada para trabajar exclusivamente con GPUs de NVIDIA con lo que consigue resultados de rendimiento muy buenos en dicha plataforma. Sin embargo, su dependencia del fabricante imposibilita su uso en dispositivos de otros fabricantes con lo que limita su portabilidad [1]. La comunidad de usuarios y fa-

[1]Esto ha sido así históricamente, si bien recientemente ha trascendido la posibilidad de que

bricantes, se ha implicado en la eliminación de estas barreras de portabilidad dando lugar al estándar OpenCL. Debido a su carácter independiente tanto del dispositivo como del fabricante, OpenCL se ve obligado a tratar con un mayor número de conceptos y procedimientos para mantenerse genérico. Así, por ejemplo, las aplicaciones basadas en OpenCL tienen que cargar y compilar las partes de la aplicación que van a ejecutarse en los aceleradores en tiempo de ejecución, añadiendo así más complejidad al proceso.

Mención aparte requieren dispositivos heterogéneos tan populares como los de la familia Intel Xeon Phi. La programación de estos dispositivos no es patrimonio exclusivo de herramientas propias de entornos heterogéneos como el ya mencionado OpenCL, sino que puede relizarse mediante los lenguajes tradicionales de programación de CPUs. Su arquitectura *many-core* con procesadores x86 permite la ejecución de aplicaciones secuenciales C/C++ así como aplicaciones multiproceso con tecnologías consolidadas y propias de sistemas con memoria compartida como OpenMP o memoria distribuida como MPI. Este hecho diferenciador permite evitar el engorro de introducir al usuario en un nuevo lenguaje de programación con los inconvenientes antes citados. Sin embargo, lo que *a priori* puede parecer una ventaja se torna en limitación hasta cierto punto, habida cuenta de que el código desarrollado deja de ser portable a otros sistemas heterogéneos.

En los últimos años, una buena parte de la investigación en computación heterogénea se ha centrado en mejorar las interfaces disponibles para su uso de tal forma que se oculte al usuario la mayor parte de los conceptos y tareas propios de este tipo de sistemas, enmarcándose esta tesis en esta área de trabajo. En concreto, nos propusimos facilitar la programación portable de todo tipo de sistemas heterogéneos mediante sucesivas mejoras de la librería de alto nivel *Heterogeneous Programming Library* (HPL). Esta librería, basada en OpenCL, permitía al inicio de nuestro trabajo programar sistemas basados en un único dispositivo heterogéneo expresando los códigos a ejecutar en el acelerador mediante un lenguaje embebido en C++. El uso del lenguaje embebido permitía desarrollar los programas utilizando un único fichero fuente e integrar mejor el código principal del programa con el del acelerador. No obstante, HPL presentaba muchas limitaciones que no permitían el desarrollo de

otros fabricantes implementen su propio compilador de CUDA en sus arquitecturas. Tal es el caso de AMD quien ha admitido esta posibilidad abiertamente.

aplicaciones genéricas heterogéneas y que fuimos resolviendo a lo largo de esta tesis. Así, a partir de la primera versión de HPL, y siguiendo un desarrollo iterativo, se fueron incorporando nuevas funcionalidades que resumimos a continuación:

Kernel nativos : La primera versión de HPL sólo permitía la escritura de kernels usando el lenguaje embebido de HPL. Esto impedía aprovechar los kernels ya escritos usando OpenCL, a los que denominamos en esta tesis kernels nativos. En esta primera iteración, se añadió la posibilidad de usar kernels que ya estuvieran escritos en OpenCL dentro de aplicaciones HPL. Con una interfaz muy simple, HPL permite ahora el empleo de kernels nativos.

Ejecución multi-dispositivo en sistemas de memoria compartida : Tras la realización de esta Tesis, HPL facilita el uso eficiente de todos los dispositivos OpenCL conectados a un sistema. Para ello permite por un lado realizar ejecuciones paralelas no sólo en dichos dispositivos sino también en la CPU principal del sistema, y por otro tener múltiples copias de un mismo array soportado por el tipo de datos *Array* en HPL, en varios dispositivos incorporando un sistema de gestión de memoria que mantiene automáticamente la coherencia de dichas copias. Este mecanismo, además de proporcionar al usuario una visión secuencialmente consistente de cada *Array* que oculta las distintas copias que debe crear el runtime, se adapta a las propiedades de los dispositivos para reducir el tiempo de intercambio de memoria entre ellos. Por otra parte, para facilitar la división de trabajo y datos entre los dispositivos involucrados en una ejecución, HPL define el concepto de *Subarray* como parte constituyente de un Array. Un Subarray tiene la entidad de un Array pero mantiene la coherencia de memoria con el Array del que forma parte. A partir de esta idea se diseñaron varios sistemas de distribución de trabajo en los entornos multi-dispositivo de HPL alcanzando buenas cotas de programabilidad en estos sistemas cada vez más comunes. Una de estas propuestas incluye modelos analíticos que reparten automáticamente el trabajo entre los dispositivos de forma que se maximice el rendimiento, obteniendo resultados excelentes en nuestros experimentos.

Mantenimiento automatizado de regiones solapadas : Las computaciones en plantilla (*stencil*) son operaciones que suelen basarse en los elementos vecinos de un dato dado para calcular el elemento correspondiente del array de salida. Este tipo de cálculos están muy presentes en una gran cantidad de ámbitos como resolutores de ecuaciones diferenciales, simuladores o procesadores de imágenes. Su implementación en entornos multi-dispositivo supone la existencia de regiones replicadas de los arrays utilizados en los dispositivos, para que cada uno disponga de la información necesaria para poder hacer las computaciones asociadas a los elementos ubicados en los bordes de la región que les ha sido asignada. HPL incorpora una nueva funcionalidad para el mantenimiento automático de la coherencia de estas regiones replicadas, facilitando así la implementación de este tipo de problemas.

Ejecución multi-dispositivo en sistemas de memoria distribuida : El siguiente paso lógico era facilitar la programación de clústers heterogéneos. Así, HPL ha sido integrada en una librería que mejora la programabilidad de aplicaciones en sistemas de memoria distribuida, *Hierarchically Tiled Arrays* (HTA), dando como resultado la librería *Heterogeneous Hierarchically Tiled Arrays* (H^2TA). La posibilidad de explotar la localidad así como de expresar paralelismo con mucho menos esfuerzo que soluciones de más bajo nivel, hacen de HTA una base adecuada sobre la que desarrollar una solución propia para este tipo de sistemas. Así, H^2TA permite la ejecución de aplicaciones HPL en un clúster con uno o varios dispositivos OpenCL por nodo. Manteniendo una API muy similar a HPL, H^2TA produce aplicaciones multiproceso con un esfuerzo muy inferior y rendimiento similar al de soluciones de más bajo nivel.

De esta forma, las librerías que proponemos permiten desarrollar aplicaciones que utilizan desde un único acelerador hasta todo un cluster heterogéneo ofreciendo la máxima programabilidad para ello, y teniendo una sobrecarga mínima con respecto a implementaciones realizadas a bajo nivel. Por otra parte, están desarrolladas en C++, un lenguaje muy eficiente y ampliamente utilizado, siendo de hecho uno de los lenguajes más extendidos en entornos de computación de altas prestaciones. Además las soluciones aquí propuestas proporcionan la máxima portabilidad respecto a los dispositivos computacionales a utilizar gracias a que están basadas en OpenCL, el entorno estándar para la computación heterogénea, el cual está soportado por todos los grandes fabricantes. Ello ha permitido que en esta tesis se incluyan resultados de pruebas realizadas sobre varios tipos de dispositivos de distintos fabricantes.

A su vez, para probar las capacidades multi-dispositivo, nuestras librerías fueron evaluadas sobre varios dispositivos trabajando conjuntamente, tanto a nivel de nodo como de clúster.

Metodología de trabajo

La metodología empleada es la conocida como Desarrollo Iterativo e Incremental, donde el trabajo de divide en diferentes fases o iteraciones y cada una de ellas finaliza con un producto perfectamente funcional con la funcionalidad de la iteración anterior y la incorporada en la iteración actual.

Esta tesis cuenta con 5 iteraciones:

B1.- Iteración 0 (Iteración inicial):

11. Análisis: Librería inicial que permita lanzar un kernel usando un lenguaje embebido.
12. Diseño/Codificación:
    - Tipo de datos Array.
    - Creación automática del código OpenCL.
    - Gestión transparente de elementos de OpenCL.
    - Coherencia automática de memoria.
13. Pruebas: Tests y “benchmarks” exhaustivos con comprobación de resultados usando kernels escritos con el lenguaje embebido.

B2.- Iteración 1

21. Análisis: Incorporación de kernels nativos.
22. Diseño/Codificación:
    - Establecer vínculo entre HPL y OpenCL: Cabecera de kernel y método de vínculo.
    - Plantillas para indicar la dirección de los argumentos del kernel nativo.

23. Pruebas: Tests y “benchmarks” con kernels nativos.

B3.- Iteración 2

31. Análisis: Extensión de la interfaz de HPL para facilitar el desarrollo de aplicaciones multi-dispositivo.
32. Diseño/Codificación:
    - Distintos métodos de reparto de trabajo: Arrays separados, basado en Subarrays; esto es, subregiones de Arrays, y basado en anotaciones.
    - Nuevo objeto de selección de regiones en arrays para su distribución posterior.
    - Selección de conjuntos de trabajo y dispositivos.
    - Autobalanceador de carga de trabajo entre un conjunto de dispositivos.
33. Pruebas: Tests y “benchmarks” para la ejecución multi-dispositivo y comparación con versiones implementadas con OpenCL nativo.

B4.- Iteración 3

41. Análisis: Extensión de la interfaz de HPL para el tratamiento de aplicaciones multi-dispositivo con regiones solapadas debido a computaciones en plantilla (*stencils*).
42. Diseño/Codificación:
    - Modificación interfaz multi-dispositivo basado en anotaciones.
    - Incorporación de método de sincronización automática de regiones fantasma, esto es, fragmentos de arrays replicados en varios dispositivos pero donde sólo uno tiene la responsabilidad de su actualización
43. Pruebas: Tests y “benchmarks” con computaciones con plantillas.

B5.- Iteración 4

51. Análisis: Uso de HPL en sistemas de memoria distribuida tales como clusters (H^2TA).
52. Diseño/Codificación:

  - Estudio de la librería HTA (*Hierarchically Tiled Arrays library*) para la mejora del desarrollo de aplicaciones multi-proceso.
  - Integración de HPL en HTA:
    - Nueva interfaz de usuario: Semejante a la de HPL para reducir la curva de aprendizaje.
    - Mecanismo de sincronización de memoria entre las estructuras de datos de HTA y HPL.

53. Pruebas: "Benchmarks" en H^2TA para probar su funcionamiento y en MPI+OpenCL para las comparaciones pertinentes.

Medios

Para la elaboración de la tesis se emplearon los medios detallados a continuación:

- Soporte económico proporcionados por el Grupo de Arquitectura de Computadores de la Universidade da Coruña y la propia Universidade da Coruña (bolsa predoutoral UDC Conv. 2013)

- Redes de investigación bajo las que se llevó a cabo esta tesis:

  - Red Gallega de Computación de Altas Prestaciones II.
  - High-Performance Embedded Architectures and Compilers Network of Excellence, HiPEAC2 NoE (ref. ICT-217068).
  - High-Performance Embedded Architectures and Compilers Network of Excellence, HiPEAC3 NoE (ref. ICT-287759).
  - Network for Sustainable Ultrascale Computing (NESUS). ICT COST Action IC1305.
  - Open European Network for High Performance Computing on Complex Environments (ComplexHPC). ICT COST Action IC0805
  - Red de Computación de Altas Prestaciones sobre Arquitecturas Paralelas Heterogéneas (CAPAP-H2) (ref. TIN 2009-08058-E).
  - Red de Computación de Altas Prestaciones sobre Arquitecturas Paralelas Heterogéneas (CAPAP-H3) (ref. TIN 2010-12011-E).

  - Red de Computación de Altas Prestaciones sobre Arquitecturas Paralelas Heterogéneas (CAPAP-H4) (ref. TIN 2011-15734-E).
  - Red de Computación de Altas Prestaciones sobre Arquitecturas Paralelas Heterogéneas (CAPAP-H5) (ref. TIN 2014-53522-REDT).
  - Red de Tecnologías Cloud y Big Data para HPC (Xunta de Galicia, ref. 2014/041)
  - Consolidación y Estructuración de Unidades de Investigación Competitivas: Centro de Investigación en Tecnoloxías da Información e as Comunicacións (CITIC) (ref. CN 2010/211))

- Proyectos de investigación que financiaron esta tesis:
  - Architectures, Systems and Tools for High Performance Computing (Ministerio de Economía y Competitividad, TIN2010-16735).
  - Consolidación y Estructuración de Unidades de Investigación Competitivas: Grupo de Arquitectura de Computadores de la Universidad de A Coruña (Xunta de Galicia, ref. 2010/6)
  - Consolidación y Estructuración de Unidades de Investigación Competitivas: Grupo de Arquitectura de Computadores de la Universidad de A Coruña (Xunta de Galicia, GRC2013-055).
  - Nuevos desafíos en la computación de altas prestaciones: Desde arquitecturas hasta aplicaciones. (Ministerio de Economía y Competitividad, TIN2013-42148-P).

- Clúster *pluton* del Grupo de Arquitectura de Computadores de la Universidade da Coruña.
  - 8 Nodos con CPU 2xIntel Xeon E5-2660 de 8 cores y 64 GB de RAM. Cada nodo cuenta con una GPU NVIDIA K20m con 5 GB de RAM. La red de interconexión es Infiniband FDR.
  - 4 Nodos con CPU Intel Xeon X5650 de 6 cores y 12 GB de RAM. Cada nodo cuenta con dos GPUs NVIDIA M2050 con 3 GB de RAM cada una. La red de interconexión es Infiniband QDR.

    - 1 Nodo con CPU 2xIntel Xeon E5-2660 de 8 cores y 64 GB de RAM. Cuenta con un accelerador Intel Xeon Phi 5110 de 60 cores y 8 GB de RAM.

- Máquina *Mercurio* del Grupo de Arquitectura de Computadores de la Universidade da Coruña. 1 Nodo con CPU Intel Core 2 con 2 GB de RAM. Cuenta con una GPU AMD HD6970 con 2GB de RAM.

- Máquina *Fermi* del Departamento de Computación da Kadir Has Ünuversitesi. 1 Nodo con CPU 4xIntel Xeon E5506 de 2 cores y 24 GB de RAM. Cuenta con una GPU NVIDIA C2050 con 3GB de RAM.

- Estancia de 3 meses en el grupo PDS del Prof. Henk Sips en la Delft University of Technology.

Conclusiones

Durante años, la computación de altas prestaciones, HPC por sus siglas en inglés, ha estado en manos de las CPUs tradicionales. Los clústers, entendidos históricamente como agregaciones de computadores con una o varias CPUs, han permitido la ejecución de aplicaciones paralelas por medio de entornos tan maduros actualmente como OpenMP o MPI, orientados a sistemas de memoria compartida y distribuida, respectivamente. La introducción de dispositivos heterogéneos como FPGAs, GPUs o procesadores *many-core* en HPC, ha despertado el interés en crear herramientas de programación para estas plataformas. La mayoría de las alternativas para programar estos dispositivos son fuertemente dependientes del tipo de dispositivo o fabricante en cuestión. OpenCL es el primer estándar que intenta desacoplar el código desarrollado del hardware utilizado, proporcionando una portabilidad real del código entre las plataformas. Un gran número de fabricantes ha desarrollado sus propias implementaciones del estándar OpenCL para sus dispositivos. Como consecuencia los códigos OpenCL pueden ser ejecutados en un amplio rango de dispositivos heterogéneos sin necesidad de alterar el código fuente. Las principales limitaciones de OpenCL son: (1) el esfuerzo de programación necesario para desarrollar aplicaciones OpenCL es alto, sobre todo para programadores no familiarizados con la programación paralela, (2) OpenCL no propociona portabilidad automática de rendimiento, esto es, para maximizar el rendimiento hemos de optimizar manualmente un código para cada plataforma en la que es ejecutado, y (3) OpenCL no soporta la programación de sistemas distribuidos, en cuyo caso éste ha de ser combinado con soluciones para entornos distribuidos como MPI. Muchos trabajos han abordado estas limitaciones de muchas formas. Entre otros, hemos de mencionar la librería *Heterogeneous Programming Library* (HPL) [22], la cual está basada en OpenCL y facilita notablemente el desarrollo de aplicaciones mono-dispositivo, abordando por tanto la primera de las limitaciones mencionadas. Esta Tesis profundiza en el uso de HPL como herramienta para superar las limitaciones mencionadas de OpenCL: mejorando en lo posible la programabilidad de los kernels HPL, proporcionando una portabilidad efectiva de rendimiento y un soporte para la programación de sistemas distribuidos compuestos por nodos heterogéneos. Además, inicialmente HPL no proporcionaba mecanismos para la programación de sistemas multi-dispositivo, donde varios dispositivos en el mismo nodo puedan ser usados al mismo tiempo. Esta limitación ha sido superada en esta Tesis.

Uno de los principales inconvenientes de HPL era que los kernels habían de ser escritos en un lenguaje embebido similar a C++. El uso de kernels OpenCL nativos no estaba soportado, lo cual limitaba el uso de código *legacy* y el empleo de optimizaciones de OpenCL de bajo nivel o propias del fabricante. Esta limitación ha sido superada en el Capítulo 2 con la extensión de HPL para dar soporte a kernels OpenCL nativos, además de los kernels escritos con el lenguage embebido original. Este capítulo también introduce otras nuevas funcionalidades para facilitar la escritura de kernels usando el lenguaje embebido. La evaluación de estas extensiones de HPL ha arrojado datos muy satisfactorios. Durante toda la Tesis, esta evaluación ha tenido dos vertientes, la programabilidad y el rendimiento. La programabilidad se basa en las métricas: líneas de código fuente (SLOCs), esfuerzo de programación (PE) [55] y número ciclomático (CN) [77]. El rendimiento siempre se ha obtenido comparando la ejecución de la librería con nueva funcionalidad con una ejecución de referencia. Cuando los kernels se escriben usando el lenguage embebido, la reducción de SLOCs, PE y CN de todo el programa es un 34 %, 44 % y un 30 %, respectivamente con respecto a versiones de referencia escritas usando OpenCL C++. Mientras tanto, el sobrecoste medio en términos de rendimiento de HPL se sitúa por debajo del 5 %. Conviene mencionar aquí que estas pruebas de rendimiento se llevaron a cabo sobre dispositivos de varios fabricantes. Siguiendo esta misma línea, el soporte de kernels OpenCL nativos ha cosechado también buenos resultados de rendimiento y programabilidad. De esta forma, HPL redujo las SLOCs y el PE del programa de host en un 23 % y un 42 % respectivamente, manteniendo las diferencias de rendimiento cercanas a cero. Esta mejora generalizada en las métricas también se pudo observar tras la extensa comparación realizada entre HPL y una de las aproximaciones análogas más maduras, ViennaCL [86]. Conviene recordar que todas las mejoras realizadas sobre HPL se han evaluado con benchmarks verificables incluyendo una aplicación real de simulación de fluídos [102].

La falta de soporte para sistemas con varios dispositivos era otra importante limitación de la versión inicial de HPL. Esta limitación ha ido creciendo en importancia recientemente dado lo comunes que son nodos u ordenadores compuestos de uno o más dispositivos, y mientras que OpenCL puede ser usado para programar aplicaciones que usen varios dispositivos al mismo tiempo, esto requiere de un importante esfuerzo de programación. Con vistas a tratar esta realidad, HPL fue extendida para dar soporte a sistemas heterogéneos con varios dispositivos. Dicha extensión supuso cambios tanto a nivel interno, definiendo un nuevo mecanismo de coherencia de memoria que diese soporte a las diferentes copias de un array alojadas en los distintos dispositivos implicados, como en la API, que fue también extendida para soportar el uso de varios dispositivos al mismo tiempo en una aplicación. El nuevo mecanismo de coherencia de memoria, probado con una sencilla implementación multi-dispositivo consiguió una rebaja de SLOCs de un 27 % y de un 43 % del PE, tomando como referencia implementaciones realizadas usando OpenCL C++. El rendimiento también se vio beneficiado gracias a la naturaleza adaptativa de HPL, la cual es aportada por la selección automática del método más eficiente para el volcado de datos entre los espacios de memoria de los dispositivos. Éste es un ejemplo de mecanismo de portabilidad de rendimiento introducido en HPL en esta Tesis. Este hecho se hace patente principalmente en aquellas aplicaciones con un mayor intercambio de datos entre dispositivos, alcanzando speedups medios del 28 % y máximos del 106 % para este tipo de aplicaciones y con respecto a ejecuciones de referencia escritas en OpenCL C++.

Los cambios en la interfaz son debidos principalmente a los tres mecanismos propuestos para distribuir una carga de trabajo entre varios dispositivos, desde el más manual basado en *subarrays*, donde el usuario ha de seleccionar explícitamente la porción de Array que ha de procesar en cada momento, hasta el más automático basado en anotaciones, donde el usuario sólo indica la dimensión por la que han de trocearse los Arrays, pasando por la versión intermedia basada en los planes de ejecución que otorga más libertad al usuario pero evitando la definición de los subarrays. Este último mecanismo incluye la posibilidad de permitir balancear automáticamente la carga de trabajo entre los dispositivos disponibles en el sistema de una forma muy sencilla. Particularmente, el usuario sólo ha de indicar los dispositivos que quiere utilizar y HPL calculará el reparto de trabajo más apropiado para ellos. Con estos tres esquemas se alcanzan reducciones máximas del PE del 76.7 % con respecto a códigos de referencia escritos en OpenCL C++. Por su parte y gracias al carácter adaptativo de HPL, la mejora de rendimiento llega a alcanzar el máximo de 146 % en comparación con los mismos códigos de referencia. Asimismo, el reparto automático de trabajo, soportado en el mecanismo basado en planes de ejecución, arroja también resultados óptimos en la mayoría de los experimentos, quedando, en el peor de los casos, un 6.3 % por debajo del rendimiento de la mejor distribución de trabajo lograda mediante una búsqueda exhaustiva. La última mejora del soporte de aplicaciones multi-dispositivo en HPL propuesta en esta Tesis consistió en un mecanismo para la actualización automática de filas fantasma que aparecen habitualmente en aplicaciones con computaciones en plantilla. Esta mejora, denominada *syncGhosts*, fue evaluada mediante varios experimentos incluyendo aplicaciones reales. Los resultados de éstos mejoran los buenos datos obtenidos tras la aplicación de los tres mecanismos de distribución propuestos. Así, mientras que con un enfoque basado en anotaciones HPL reduce la media del PE en un 20.5 % respecto a usar el enfoque basado en subarrays, esta reducción alcanza el 79.5 % al usar la técnica de syncGhosts conjuntamente con el esquema multi-dispositivo basado en anotaciones. En particular, para la aplicación de procesado de imágenes, CANNY, la reducción alcanza el 96.7 %. Para medir con más precisión el impacto que este mecanismo pudiera tener en el tiempo de ejecución total de la aplicación, se midió su rendimiento en sistemas con dos y tres dispositivos y en ninguno de ellos las diferencias de rendimiento superan el 1 %, asegurando la robustez de la implementación con esquemas de repartos de datos más complejos.

La última cuestión abordada en esta Tesis es la programación de sistemas distribuidos compuestos de nodos con dispositivos heterogéneos. Ésto se logró a través de la integración de la librería *Hierarchically Tiled Array* (HTA) [6] y HPL dando lugar a la librería *Heterogeneous Hierarchically Tiled Arrays* (H^2TA). Esta librería está basada en el tipo de datos abstracto HTA, que representa a un array dividido jerárquicamente en bloques o submatrices. Estos bloques pueden estar distribuidos en un clúster y ser procesados en paralelo, proporcionando al usuario una visión global de los datos distribuidos. La librería H^2TA propuesta permite el uso por parte de los programadores de los dispositivos con soporte OpenCL disponibles en un cluster, aprovechándose de las propiedades combinadas de las HTAs y de la sencillas semántica y API de HPL. Los resultados obtenidos son muy positivos. Por ejemplo, si comparamos la programabilidad de las H^2TAs con la de una aproximación formada por la combinación de HPL y la librería MPI para las comunicaciones, la cual ya disfruta de una mejora significativa de programabilidad respecto a soluciones de más bajo nivel, H^2TAs reducen las SLOCs, el PE y en CN en un 20.5 %, 31.8 % y un 26.9 %, respectivamente. Estos resultados también son mejores que los obtenidos usando ambas librerías de forma separada, tal y como se demuestra en el Capítulo 4 comparando aplicaciones basadas en H^2TA con versiones escritas combinando HTAs y HPL. En cuanto al rendimiento, H^2TA mantiene una pérdida de rendimiento media con respecto a soluciones basadas en MPI por debajo del 1 %, lo cual, y teniendo en cuenta los resultados de programabilidad, no hace sino justificar este desarrollo.

Principales contribuciones

- Estudio y análisis de múltiples soluciones para computación heterogénea.
- Estudio y prueba de diferentes arquitecturas y entornos.
- Implementación de soluciones híbridas resultantes de la fusión de varios paradigmas.
- Diseño, implementación y prueba de una solución para el problema de computación heterogénea:
    - un proceso con uno o varios dispositivos en un nodo,
    - varios procesos con uno o varios dispositivos por nodo en un clúster heterogéneo.

- Estudio exhaustivo de programabilidad y rendimiento de esta solución comparándolos con los de otras alternativas notorias existentes.

Publications from the thesis

- Viñas, M.; Lobeiras, J.; Fraguela, B.B.; Arenaz, M., Amor, M. Doallo, R. Simulation of Pollutant Transport in Shallow Water on a CUDA Architecture, Proceedings of the 2011 International Conference on High Performance Computing and Simulation (HPCS 2011), pp. 664-670, 2011.

- Viñas, M.; Lobeiras, J.; Fraguela, B.B.; Arenaz, M.; Amor, M.; García, J.A.; Castro M.J.; Doallo, R. A Multi-GPU shallow water simulation with transport of contaminants. Concurrency and Computation, 25(8), pp. 1153-1169, 2013.

- Lobeiras, J.; Viñas, M.; Amor, M.; Fraguela, B.B.; Arenaz, M.; García, J.A.; Castro, M.J. Parallelization of Shallow Water Simulations on Current Multi-threaded Systems. International Journal of High Performance Computing Applications, 27(4), pp. 493-512, 2013.

- Viñas, M.; Bozkus, Z.; Fraguela B.B. Exploiting Heterogeneous Parallelism with the Heterogeneous Programming Library. Journal of Parallel and Distributed Computing, 73(12), pp. 1626-1638, 2013.

- Viñas, M.; Bozkus, Z.; Fraguela B.B. Heterogeneous Programming Library: A Framework for Quick Development of Heterogeneous Applications, Proceedings of the ACACES 2013, pp. 69-72, 2013.

- Viñas, M.; Bozkus, Z.; Fraguela B.B. Heterogeneous Programming Library: A Framework for Facilitating the Exploitation of Heterogeneous Applications, Proceedings of the 17th Workshop on Compilers for Parallel Computing, (CPC'13), 2013.

- Viñas, M.; Bozkus, Z.; Fraguela B.B.; Andrade, D.; Doallo, R. Exploiting multi-GPU systems using the Heterogeneous Programming Library, Proceedings of the 14th Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE 2014), 2014.
- Viñas, M.; Fraguela B.B.; Bozkus, Z.; Andrade, D. Improving OpenCL Programmability with the Heterogeneous Programming Library, Proceedings of the International Conference on Computational Science (ICCS 2015), 2015.
- Viñas, M.; Bozkus, Z.; Fraguela B.B.; Andrade, D.; Doallo, R. Developing adaptive multi-device applications with the Heterogeneous Programming Library. The Journal of Supercomputing, 71(6), pp. 2204-2220, 2015.

Contents

1. Introduction **1**

1.1. Heterogeneous Computing 2

1.2. OpenCL 4

1.2.1. The Platform Model 5

1.2.2. The Execution Model 6

1.2.3. Memory Model 8

1.2.4. Example: vectorAddition 10

1.2.5. OpenCL C++ bindings 14

1.3. High Level Proposals in Single Device Environments 16

1.3.1. Support for common and skeletal operations 16

1.3.2. Compiler directives 16

1.3.3. Libraries that improve the usability of low level APIs 17

1.4. High level proposals that target multi-device systems 18

1.5. Heterogeneous Clusters 19

1.6. Thesis approach and contributions 20

2. The Heterogeneous Programming Library **25**

2.1. Programming model . . . 26
2.2. Library frontend . . . 28
2.2.1. The `Array` data type . . . 29
2.2.2. HPL embedded language . . . 32
2.2.3. Host Interface . . . 38
2.3. Evaluation . . . 44
2.3.1. Embedded language kernels . . . 45
2.3.2. Native OpenCL C kernels . . . 53
2.4. Related work . . . 56
2.5. Conclusions . . . 59

3. Multi-device computing **61**
3.1. Multi-device support in HPL . . . 62
3.1.1. Implementation details . . . 67
3.2. Improving multi-device support . . . 70
3.2.1. Subarrays . . . 70
3.2.2. Subkernels based on annotations . . . 72
3.2.3. Subkernels based on execution plans . . . 73
3.2.4. Unified memory exploitation . . . 77
3.3. Improving stencil applications . . . 78
3.3.1. Automatic update of the shadow regions: syncGhosts . . . 82
3.4. Evaluation . . . 84
3.4.1. Naïve multi-device support . . . 86
3.4.2. Improved multi-device support . . . 92

3.4.3. Improved stencil applications 97
3.5. Related work 104
3.6. Conclusions 107

4. Heterogeneous clusters support **111**

4.1. Hierarchically Tiled Arrays 112
4.2. Heterogeneous Hierarchically Tiled Arrays 115
4.2.1. Separate use of HTA and HPL 116
4.2.2. Heterogeneous cluster programming with H^2TAs 119
4.3. Evaluation 122
4.3.1. Programmability 124
4.3.2. Performance 125
4.4. Related work 127
4.5. Conclusions 130

5. Conclusions **133**

5.1. Future Work 137

References **139**

Appendices **151**

A. OpenCL Programming **153**

List of Figures

1.1. Platform Model . . . 5

1.2. Global local index spaces . . . 7

1.3. OpenCL Memory Model . . . 9

1.4. Example of an 1-D problem in OpenCL C . . . 13

1.5. Example of an 1-D problem in OpenCL C++ . . . 15

2.1. Heterogeneous Programming Library hardware model . . . 26

2.2. HPL running example . . . 29

2.3. Declaring a `struct` type to HPL in order to use it in `Arrays` . . . 30

2.4. Usage of `Arrays` in host code . . . 31

2.5. SAXPY kernel in HPL . . . 32

2.6. Generic HPL kernel to add bidimensional arrays of any type . . . 33

2.7. Dot product kernels in HPL . . . 34

2.8. Using regular C++ in a kernel to generate an unrolled matrix product 36

2.9. Array creation and SAXPY kernel usage . . . 39

2.10. Kernel invocation algorithm. . . . 41

2.11. Saxpy using native OpenCL C kernel with HPL . . . 44

2.12. Productivity metrics reduction in HPL with respect to two OpenCL baseline implementations 48

2.13. Performance of the OpenCL and the HPL versions of the codes, normalized to the runtime of the OpenCL version 50

2.14. Simulation of evolution of a pollutant in Ría de Arousa 51

2.15. Runtime of the OpenCL and the HPL versions of the shallow water simulator for different problem sizes 53

2.16. Productivity improvement in ViennaCL and HPL with respect to the baseline 55

2.17. Overhead of ViennaCL and HPL in ShWa with respect to the baseline 55

3.1. Matrix product on a single device using HPL 65

3.2. Matrix product on multiple GPUs using HPL 65

3.3. Data exchange to implement a pipeline between devices 67

3.4. Bandwidth of copies between devices. T stands for two transfers in sequence and C for clEnqueueCopyBuffer usage 69

3.5. Matrix product on multiple GPUs using subarrays 71

3.6. Matrix product on multiple GPUs using an execution plan 75

3.7. `SINGLE_STEP_MODEL` load balancing algorithm 76

3.8. `ITERATIVE_MODEL` adaptive load balancing algorithm 78

3.9. HPL stencil example in single-device 80

3.10. HPL stencil example in multi-device using subarrays 81

3.11. Examples of ghost regions for (a) 1-D, (b) 2-D and (c) 3-D problems in a problem divided by two devices 83

3.12. 1-D `syncGhosts` algorithm 83

3.13. Example on multiple GPUs using subarrays with `syncGhosts` 85

3.14. Example on multiple GPUs using annotations with `syncGhosts` . . . 85

3.15. Reduction in the number of SLOCs and programming effort of the host side of the application when using HPL with respect to the OpenCL C++ baseline. 88

3.16. Speedups with S2050 90

3.17. Speedups with K20 90

3.18. Speedups with Xeon Phi 90

3.19. Speedup of HPL over OpenCL for different problem sizes of FT . . . 91

3.20. Speedup of HPL over OpenCL for different problem sizes of ShWa . . 91

3.21. Speedup of HPL over OpenCL for different problem sizes of N-Body . 91

3.22. Performance in the S2050 system using both GPUs 95

3.23. Performance in the K20 system using two GPUs 95

3.24. Relative performance of the distributions found by the analytical models with respect to the best one found when using one CPU and two K20 GPUs. 97

3.25. Performance in the S2050 system using both GPUs 101

3.26. Performance in the K20 system using two GPUs 101

3.27. Performance improvement using 3 devices 101

3.28. Performance of *syncGhosts* versions varying the ghost region sizes and using two GPUs in K20 system 103

4.1. HTA creation 113

4.2. HTA indexing 113

4.3. HTA complex assignment example 114

4.4. Parallel application of a user-defined function to the tiles of HTAs . . 115

4.5. Joint usage of HTAs and HPL 117

4.6. Example of the memory coherency mechanism implemented when HTA and HPL are separately used. 118

4.7. H^2TA example code 120

4.8. HPL example code 121

4.9. Reduction of programming complexity metrics of HTA+HPL and H^2TA programs with respect to versions based on MPI+HPL. 125

4.10. Speedup of the executions in Fermi using multiple devices with respect to an execution using a single device 127

4.11. Speedup of the executions in K20 using multiple devices with respect to an execution using a single device 128

List of Tables

1.1. Summary of the high level proposals. 21

2.1. Predefined HPL variables. 32

2.2. OpenCL generating process. 39

2.3. Benchmarks characteristics. 45

2.4. NAS Parallel Benchmark EP runtimes for class C (in seconds) 52

2.5. Benchmarks characteristics. 54

3.1. Benchmarks characteristics. 87

3.2. The Hardware Platform Details 89

3.3. MG benchmark characteristics. 93

3.4. Programmability improvement for several strategies. 93

3.5. Slowdown of the worst distribution with respect to the best one in the two configurations tested. 98

3.6. Benchmark details 99

3.7. Programmability improvement without and with the syncGhosts mechanism of HPL programs based on annotations with respect to versions based on explicit subarrays. 100

4.1. Benchmarks characteristics. 124

1. OpenCL device types. 154

2. OpenCL device types. 161

Chapter 1

Introduction

Heterogeneous: composed of diverse elements or constituents; consisting of parts of different kinds; not homogeneous. (Oxford dict.)

The relevance of the usage of computing devices with very different characteristics that cooperate in a computation has increased exponentially in the past few years. The reason for this has been the appearance of accelerators that can be programmed to perform general-purpose computations achieving larger speedups and/or power savings than traditional single-core and even multi-core CPUs.

Unfortunately this hardware heterogeneity is also reflected in the software required to program these systems since, unlike with regular CPUs, with these types of accelerators programmers are typically exposed to a number of characteristics and limitations that must be handled. This way, heterogeneous systems require much more effort to be programmed than the traditional computers because of the appearance of new concepts and tools with different restrictions. Additionally, many of the approaches to exploit heterogeneous systems are specific to one vendor or device, resulting in little portability or rapid obsolescence for the applications built on them. Open standards for programming heterogeneous systems such as OpenCL contribute to improve this situation, but the requirement of portability has led to a programming interface more complex than that of other approaches.

The purpose of this PhD Thesis is to propose tools that facilitate the programming of heterogeneous computing systems while providing the maximum portability and performance. This first chapter introduces the reader to heterogeneous computing by means of a short description of its history and current state of the subject matter. This is followed by an introduction to the only portable alternative nowadays, the OpenCL standard, which is the backend used by the tools developed in this Thesis. Next, the family of solutions for the improvement of the programmability of heterogeneous systems considering a single device, multiple devices in the same node, and heterogeneous clusters is briefly reviewed in Sections 1.3, 1.4, and 1.5, respectively. The chapter finishes with the motivation and the scope of this PhD Thesis and its contributions.

1.1. Heterogeneous Computing

Heterogeneous computing arises when different types of devices with compute capability are used by an application. These devices usually include one or more general purpose CPUs that collaborate with a number of so-called accelerators, which are computing systems that cannot operate on their own and to which the CPUs can offload computations. These devices, characterized by their high performance, first appeared in the market as specialized hardware that sought to satisfy the strong demand for high resolution 3D graphics in real-time. In those days, the users were able to implement their own applications for graphics visualization. At the beginning these rendering processes had to be done on the CPU using the graphics hardware only in order to display the pixels in the screen. Unfortunately, the CPU was too slow to produce attractive 3D effects. To solve this problem, the graphics hardware evolved, gradually gaining computing power, and as a result the CPU was increasingly freed from the graphics tasks. Therefore, at some point 3D applications no longer implemented their own 3D rendering algorithms on the CPU; but rather they began to rely on either OpenGL [105] or Direct3D [21], the two new standard 3D programming interfaces to communicate rendering commands the new generation of devices, called GPUs (*Graphics Processing Units*). GPUs are very well suited for these tasks because they offer several pipelines to execute different "shaders" or programs to compute operations on the visualization matrix. Languages, like OpenGL Shading Language [85] or High Level Shader Language [91] were the most widely used for OpenGL and Direct3D environments respectively.

Cg [76] is a proprietary language of Nvidia that allows the creation of shaders for both OpenGL and Direct3D.

The GPU tasks are executed on many data elements in parallel. Many algorithms outside the field of image rendering and processing can be accelerated applying data-parallel processing, also known as “stream processing”. The first attempts at exploiting GPUs for general purpose computations, also called GPGPU, relied on the existing languages oriented to graphic tasks. However, the approach was cumbersome and there were many limitations. This, together with the large advantages observed when exploiting GPGPU, lead to the development of new programming languages as well as improved hardware for the GPUs that made them more amenable for GPGPU. This way, version 11.0 of Direct3D was equipped with a new graphic pipeline to perform tasks not related with graphics, such as stream processing, giving birth to the compute shaders, also called kernels. These are computing unities not linked to a graphical task, having on the contrary hardware suitable for general purpose computations. As a result, they demand the management of new concepts and procedures not related with graphics. Programming frameworks such as CUDA [80], Close To Metal [10], BrookGPU [25] or Brook+ [8] are the answer of manufacturers and organizations to this demand. The current situation is that, despite being supported by a single vendor, the NVIDIA CUDA platform accounts nowadays for most of the GPGPU market.

There are also accelerators that played a very important role in the raise of heterogeneous computing but have now disappeared. This is the case of the Cell processor [60] developed in the early 2000’s. The Cell processor was jointly developed by Sony, Toshiba and IBM and while its main commercial application was the Sony’s PlayStation 3 game console, it was also used in a number of computers as accelerator. It consisted in nine processors: one acting as a controller and the rest acting as mere processing units, all of them interconnected with a bus. Its programming relied on C/C++ with language extensions [59] to support vector types and functions to operate with those vectors that are executed in parallel distributing the work on the processing units. Despite its good performance for scientific computing, the large number of details that needed to be managed coupled with the low level of the tools available for the development of Cell applications strongly hampered the adoption of this platform beyond very specific niches, leading to its cancellation in 2009.

In contrast, there are other co-processors with a longer history that enjoy nowadays the support of an important community of developers and users. This is the case of the Field-Programmable Gate Arrays (FPGAs), a co-processor family with a long and successful history . The main difference between these devices and the GPUs is related with their hardware arquitecture. FPGAs are arrays of programmable logic blocks, memory cells and physical connections, which the programmer is in charge of programming at hardware level, instead of software level which is the case of GPUs. The compilers reconfigure the FPGA so that it becomes a custom processor designed for computing a specific kernel. Programming FPGAs has traditionally been difficult and requires expertise in specialized languages like VHDL [12] or Verilog [97]. Another successful case is the Intel Xeon Phi, which has became pretty popular as a co-processor, an important reason being its very good programmability. In fact, it can be programmed using the same tools as standard multicore systems, such as OpenMP, MPI, or pthreads because it is made up of a collection of x86 processors. This accelerator is based on a previously cancelled project of Intel called *Larrabee* that seeked to create a new family of GPUs based on x86 cores.

As we have seen, there are several families of accelerators, and worse, a large number of incompatible alternatives to program them, which makes applications based on them inherently non-portable. As a result of this situation, several organisms and manufacturers grouped under the Khronos Group brand, focused their efforts on developing a standard for heterogeneous computing, the Open Computing Language (OpenCL) [62]. Nowadays, OpenCL is supported by the large majority of the vendors of heterogeneous systems (IBM, AMD, Intel, NVIDIA, . . .) and it already replaced several APIs now deprecated such as Close To Metal, BrookGPU or Brook+. Because of this portability we chose OpenCL as the backend for our efforts to improve the programmability of heterogeneous systems.

1.2. OpenCL

The Open Computing Language (OpenCL [62]) is an open API designed to allow the use of GPUs and other co-processors to work jointly with the CPU, in order to take advantage of the additional computing power. As a standard, OpenCL 1.0 was released in December 2008, by an independent standards consortium called The

Khronos Group. The standard has evolved from that initial version to the most recent one, the OpenCL 2.1 released in November 2015. This introduction only includes basic concepts of OpenCL and therefore, they have endured through the successive versions of the standard.

The two main features of OpenCL are the exploitation of all the OpenCL computing resources available in the system, such as multi-core CPUs and GPUs among others, and the total portability of the OpenCL codes among different manufacturers, unless vendor-specific extensions are used. The standard separates the software layer, which is a responsibility of the programmers, from the hardware layer, which is a responsibility of the manufacturers. All the hardware implementation details, such as drivers and runtime are transparent from the point of view of the software programmers.

1.2.1. The Platform Model

The platform model of OpenCL, illustrated in Figure 1.1, is defined as a host connected to one or more OpenCL devices. A host is any computer with a CPU. The OpenCL devices can be GPUs, FPGAs, many-core processors, . . . A device consists

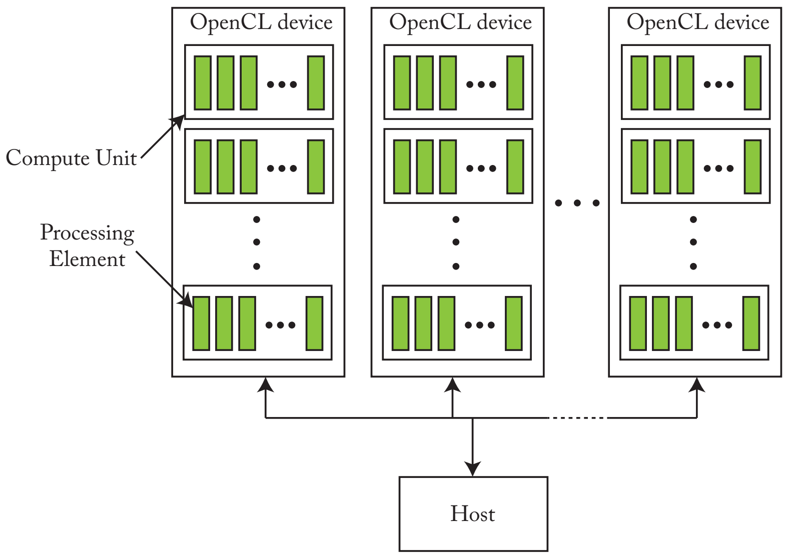


Figure 1.1: Platform Model

of a collection of one or more *compute units*. A compute unit is at the same time composed by one or several *processing elements*. The processing elements execute SIMD (Single Instruction, Multiple Data) instructions so that only one instruction is executed simultaneously in several processing elements.

1.2.2. The Execution Model

The OpenCL execution model mainly consists of two different elements: the kernels and the host program. A kernel is the basic unit of executable code that runs on an OpenCL device. Kernels are basically C-like functions that are executed in parallel by the *processing elements* of a device. The host program is the main program, which is executed in the CPU and defines a context for the OpenCL devices and enqueues the kernel executions using command queues. The queuing is in-order but the execution of the command can be out-of-order.

Kernels

OpenCL defines for each kernel a N-dimensional index space of work. This workspace can have up to three dimensions. The kernel will be enqueued to be executed in the device. The OpenCL runtime creates one instance of this kernel, called *work-item*, per point of the defined index space. While each work-item executes the same kernel function, it does it using different data that can be identified using the global position of the work-item in the index space, which is known as *global ID*.

In OpenCL the work-items can be grouped in teams called *work-groups*. The size of each work-group is defined by its own local index space. As in the global index space, work-items can also know their position into the local index space (*local ID*). All the work-items that belong to a work-group share several properties. First, their execution takes place in the same compute unit because the execution of a work-group can not be split in more than one compute unit. Second, the work-items belonging to the same group share a local *on-chip* memory. As a result, the communication of data into a work-group can be done in a very fast way using this memory. Finally, work-items can be synchronized at a work-group level, this is, the work-group execution can be stopped at a synchronization point, concretely a barrier, set by the programmer.

Figure 1.2 illustrates the example of a two-dimensional global index space of 16×16 work-items. This index space is divided in 16 work-groups. The work-group containing the highlighted work-item has a global ID (2,3) and a local size 4×4. The local ID of the highlighted work-item (in green) is (2,1), although it can be also identified globally with the global ID (10,13).

Host Program

The host program is in charge of setting up and managing the execution of the kernels on the chosen OpenCL devices of the OpenCL platforms installed in the system. By means of the OpenCL API, the host can create and manipulate the following elements belonging to an OpenCL context:

- Devices: Set of OpenCL devices used by the host for kernel execution.
- Program objects, that implements a kernel or a collection of them.
- Kernels: OpenCL functions that will be executed on the device.
- Memory objects: Memory buffers used in both the host program and the OpenCL devices.

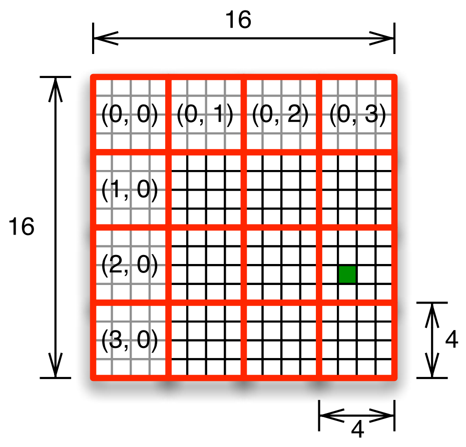


Figure 1.2: Global local index spaces

- Command queues: Objects in charge of submitting commands on the device.

When the context is created, the command queues are created and managed by the user to perform the execution of the kernels in the OpenCL devices associated to that context. Overall, the command queues accept three kind of commands:

- Kernel execution command, which runs the kernel in the device.
- Memory commands, which transfer data between the host program memory and the device memory.
- Synchronization commands, which allow the user to manipulate the order of the command execution with respect to other commands or the host program.

The user can specify two types of execution for the commands enqueued: blocking or non-blocking. If a command is blocking, the command blocks the execution of the host program until its completion. Otherwise, the host program continues its execution without waiting for the result of the command.

1.2.3. Memory Model

Figure 1.3 shows the scheme of the OpenCL memory model, which reflects four kinds of memory that a work-item can access during the execution of a kernel: Global, Local, Private and Constant memory. We describe them now in turn.

Global Memory

The global memory is the main and largest memory space of the device and it can span several GB. Work-items can read and write random memory positions in this memory, which is also used to communicate the host with the devices. If the kernel needs to send or receive data to/from the host application, they have to be stored in global memory. Nowadays, accelerators have several levels of cache for this memory, which help to mitigate the time penalty suffered in the global memory accesses because of the lack of locality and/or coalescence. Coalescence here refers to the ability of the hardware to combine several memory accesses into a reduced number

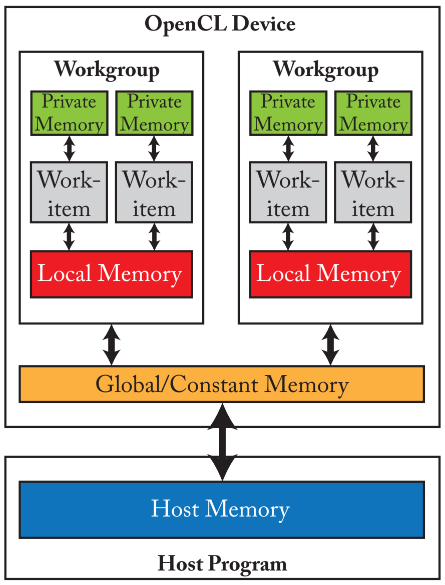


Figure 1.3: OpenCL Memory Model

of memory transactions. Briefly, a global memory access is coalesced (combined) if and only if consecutive work-items access consecutive global memory positions and this memory is aligned (i.e. its address is a multiple of the data type size). Before the introduction of global memory caches, uncoalesced accesses played a much larger role in the performance of a kernel.

Local Memory

The local memory only serves to the work-items of the same work-group, who have the same view of this memory space. It is a very fast *on-chip* memory so it can be seen as a scratchpad that can be managed by the user.

Private Memory

By default, all the variables defined inside a kernel are stored in this memory. Private memory has two natures. On the one hand, if a scalar variable is defined and there are registers available, that variable will be stored in a register. If an array is defined, or there are no free registers for store a scalar variable, that variable will be stored in global memory. *Register spilling* is the problem that appears when the registers are overused, causing the copy of data from global memory to registers and vice-versa in order to keep in the registers the working set of the kernel. This problem can cause important performance penalties.

Constant Memory

The constant memory is a memory space that can be defined statically inside the kernel code, or dynamically before the kernel execution. As its name implies, it is constant for the kernels, all the changes on it being performed by the host application.

1.2.4. Example: vectorAddition

This section shows an example (Figure 1.4) of a host program and its corresponding kernel. It serves as an example of the typical steps of an OpenCL application. The kernel of the example chosen uses two buffers, *src* and *dst*, and computes $dst = dst + src$. This code is somewhat simplified with respect to a realistic one because it does not include error checks and it assumes the existence of only one GPU. In addition, the buffers used are not initialized in order to center only on the requirements of an OpenCL application.

Firstly, Lines 6-9 define the string that contains the kernel code that will be executed on the selected device. It is a vector addition that uses two vectors `src` and `dst` defined in global memory. The memory spaces in OpenCL can be distinguished through the keywords `__global`, `__local` and `__constant`. Note that it is not needed to specify a private memory space because it is the default choice inside the kernel when none is specified. The code of this kernel specifies that each work-item adds an element of the `src` array with an element of the same position of `dst` array and stores the result in the latter one. The functions `get_global_id(int d)` and `get_local_id(int d)` provide the positions in the `d` dimension of the calling thread in the global and local index spaces respectively.

The first step in the application is to obtain the platform found in the system (Line 14). The most relevant functions of the OpenCL API are detailed in Appendix

A. In Line 17, we obtain the identifier of the first GPU device of the chosen platform. If the system has more than one platform or device, these same calls will return pointers to the platforms and devices found.

In Line 19, a context is created for the platform and device chosen. Taking into account the context created and the device associated to it, the `commandQueue` object is created in Line 21.

In Line 23, the program object is created using the kernel code and it is compiled in Line 24 for the chosen device. In Line 26, the kernel object, which will be used for passing arguments and its later execution, is obtained from the program.

Lines 29-30 define the buffers that contain the data of the vectors in the device memory. Their creation includes the context they belong to and, among other parameters, the access type in the kernel. The most important types are read-only (`CL_MEM_READ_ONLY`), write-only (`CL_MEM_WRITE_ONLY`) and read-write (`CL_MEM_READ_WRITE`).

In Lines 35-36 the content of the buffers is copied from the host to the device by means of the `clEnqueueWriteBuffer` function. In Line 35, this command is enqueued in the `commandQueue` passed as argument. This command performs the copy of the data pointed by `src_host` in the device buffer `src_buffer`. In Line 36, the same task is done for the target buffer.

Lines 38-41 specify the execution parameters of the kernel. In the first place, Line 38 defines the size of the global index space of the problem. In this case it is one-dimensional with NWORKITEMS work-items. Lines 39 and 40 specify the arguments of the kernel, in this case only the buffers `src_buffer` and `dst_buffer`. Finally, in Line 41, the execution of the kernel `kernel` is launched by means of the call `clEnqueueNDRangeKernel` specifying among others, the `commandQueue` that will perform the execution and the global space. It deserves to be mentioned, that in this case there is no local domain size specified (6th argument). Instead of a local domain, there is a NULL pointer so that the system is in charge of choosing the most suitable local domain for the hardware of the selected device.

Kernel invocations are not blocking. For this reason, after this point the kernel is executed in the GPU in parallel with the rest of the host code. Since the result of `dst_buffer` is used in the host application after Line 43, it has to be downloaded from the GPU to the host. The transfer of the data of `dst_buffer` to the host memory pointed by `dst_host` is achieved by means of the call `clEnqueueReadBuffer` of Line 43. Some commands like `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` allow to specify their behavior in terms of synchronization by means of the BLOCKING flag (3rd argument). This is, if the BLOCKING flag is TRUE, the host application will stop until the command specified as BLOCKING has finished. Otherwise, the control will return to the host immediately after enqueueing the command. In this example, the host will be waiting for the finish of the `enqueueReadBuffer` command.

Finally, the elements of the resulting vector, recently copied from the device to the host, are printed on the standard output.

```
#include <CL/cl.h>
#include <stdio.h>

#define NWORKITEMS 1024

const char *kernel_code =
``__kernel void vectorAddition(__global int *src, __global int *dst) { \n \
   dst[get_global_id(0)] += src[get_global_id(0)]; \n \
 }'';

int main(int argc, char** argv)
{
  cl_platform_id platform;
  clGetPlatformIDs(1, &platform, NULL);

  cl_device_id device;
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);

  cl_context context = clCreateContext(NULL, 1, &device, 0, NULL);

  cl_command_queue queue = clCreateCommandQueue(context, device, 0, NULL);

  cl_program program = clCreateProgramWithSource(context, 1, &kernel_code, NULL, NULL);
  clBuildProgram(program, 1, &device, NULL, NULL, NULL);

  cl_kernel kernel = clCreateKernel(program, "vectorAddition", NULL);

  int size = NWORKITEMS * sizeof(int);
  cl_mem src_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, size, NULL, NULL);
  cl_mem dst_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, size, NULL, NULL);

  int *src_host = (int*)malloc(size);
  int *dst_host = (int*)malloc(size);

  clEnqueueWriteBuffer(queue, src_buffer, CL_TRUE, 0, size, src_host, 0, NULL, NULL);
  clEnqueueWriteBuffer(queue, dst_buffer, CL_TRUE, 0, size, dst_host, 0, NULL, NULL);

  size_t global_work_size = NWORKITEMS;
  clSetKernelArg(kernel, 0, sizeof(buffer), (*void)&src_buffer);
  clSetKernelArg(kernel, 1, sizeof(buffer), (*void)&dst_buffer);
  clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_work_size, NULL, 0, NULL, NULL);

  clEnqueueReadBuffer(queue, dst_buffer, CL_TRUE, size, dst_host, 0, NULL, NULL);

  for(int i = 0; i < NWORKITEMS; i++)
    printf(``%d %d\n'', i, dst_host[i]);

  return 0;
}
```

Figure 1.4: Example of an 1-D problem in OpenCL C

1.2.5. OpenCL C++ bindings

The Khronos Group has also provided a host API for C++ that is called the OpenCL C++ bindings or wrapper API. This version works as a wrapper of OpenCL C, so that it provides exactly the same performance, but better programmability. These bindings, which are often called for short OpenCL C++ in the rest of this document, are one of the first attempts in order to reduce the development costs of the heterogeneous computing using OpenCL. Figure 1.5 illustrates the same code of Figure 1.4 using the object-oriented approach of OpenCL C++.

The code that relies on this C++ wrapper is more concise and intuitive than the one based on the C bindings. As we will see in the following chapters, since OpenCL is the standard of the heterogeneous computing, each new feature added to HPL has been compared with OpenCL both in terms of performance and programmability to make strong enough each iteration of the development. This OpenCL C++ wrapper has been used in all the comparisons because it has proven to improve the programmability of OpenCL C codes with no performance penalties. Additionally, HPL has been developed following the object-oriented paradigm in the same way as OpenCL C++, both of them thus enjoying the programmability advantages of that paradigm.

```
#include <CL/cl.hpp>
#include <iostream>

#define NWORKITEMS 1024

const char *kernel_code =
‘‘__kernel void vectorAddition(__global int *src, __global int *dst) { \n \
   dst[get_global_id(0)] += src[get_global_id(0)]; \n \
 }’’;

int main(int argc, char** argv)
{
  std::vector<Platform> platforms;
  cl::Platform::get(&platforms);

  std::vector<Device> devices;
  platform[0].getDevices(CL_DEVICE_TYPE_GPU, &devices);

  Context *context = new Context(devices[0]);

  CommandQueue *queue = new CommandQueue(*context, devices[0]);

  Program *program = new Program(*context, &kernel_code);
  program->build(devices);

  Kernel *kernel = new Kernel(*program, ”vectorAddition”);

  int size = NWORKITEMS * sizeof(int);
  Buffer *src_buffer = new Buffer(*context, CL_MEM_READ_ONLY, size);
  Buffer *dst_buffer = new Buffer(*context, CL_MEM_READ_WRITE, size);

  int *src_host = new int[size];
  int *dst_host = new int[size];

  queue->enqueueWriteBuffer(*src_buffer, CL_TRUE, 0, size, src_host);
  queue->enqueueWriteBuffer(*dst_buffer, CL_TRUE, 0, size, dst_host);

  size_t global_work_size = NWORKITEMS;
  kernel->setArg(0, sizeof(Buffer), src_buffer));
  kernel->setArg(1, sizeof(buffer), dst_buffer));
  queue->enqueueNDRangeKernel(*kernel, cl::NullRange, global_work_size, cl::NullRange);

  queue->enqueueReadBuffer(*dst_buffer, CL_TRUE, 0, size, dst_host, 0);

  for(int i = 0; i < NWORKITEMS; i++)
    std::cout << i << ‘‘ ’’ << dst_host[i] << std::endl;

  return 0;
}
```

Figure 1.5: Example of an 1-D problem in OpenCL C++

1.3. High Level Proposals in Single Device Environments

The previous section showed that OpenCL applications require programmers to manage a vast amount of new concepts and procedures, even to perform simple computations. For the past few years, the research community has come up with several proposals to improve the programmability of the codes developed for heterogeneous systems. This research can be classified in three large groups: proposals that support common and skeletal operations, compiler directives and libraries that improve the usability of the most common APIs.

1.3.1. Support for common and skeletal operations

Some proposals [86][9] identify important functions algorithms and operations, such as the Fourier transform or the common linear algebra operations [34], which appear in a large number of applications, and provide solutions restricted to them. Other researchers have focused on facilitating the expression of typical patterns of computation in heterogeneous systems, sometimes focusing more strongly on data-parallelism [28] and in other cases [38][93] on skeletons [30][50]. These solutions both avoid the boilerplate code associated to the lower level solutions and automatically implement the synchronization and communication tasks associated to these common patterns. Some approaches combine several of these features. For example, [19, 64] provide both predefined functions and tools for the easy execution of custom kernels under strong restrictions, as they only support one-to-one computations and reductions. The problem with these proposals is that, while they are very convenient, they are only suitable for the computations that adjust to the functionality they provide.

1.3.2. Compiler directives

Other works provide a more widely applicable solution by means of compiler directives [47][20][69][56][81][84]. Given its large popularity in shared memory systems, a particularly important member of this group is OpenMP [82]. The version 4 or OpenMP allows the development of heterogeneous applications. Therefore, it deserves to be compared with the other main standard of this group, OpenACC [81]. In many ways both have a similar structure, but while OpenMP is more prescriptive, setting explicitly the moment to do an operation, OpenACC is more descriptive. This way, using OpenACC programmers only specify the operation to do, leaving more freedom to the compiler to perform the considered optimizations. While, OpenACC is currently more mature for accelerators, OpenMP continues gradually increasing its support for the heterogeneous devices. This way, version 4.5 of the OpenMP standard was approved in 2015, bringing new mechanisms for data mapping and asynchronous execution as well as routines for allocating, copying and freeing device memory.

Compiler directives require specific compilers and usually provides users little or no control on the result, which strongly depends on the capabilities of the compiler. Relatedly, these tools usually lack a clear performance model [79]. These problems are even more important when we consider accelerators. The reasons are the large number of characteristics that can be managed, which leads to a much wider variety of potential implementations for a given algorithm than regular CPUs, and the high sensitivity of the performance of these devices with respect to the implementation decisions taken.

1.3.3. Libraries that improve the usability of low level APIs

The other family of proposals that enjoy the widest scope of applications are libraries that improve the usability of the most common APIs, OpenCL in particular. These libraries, among others, [68][104] require the kernels to be written using the native API, focusing on the automation of the tasks performed in the host code. In this category we can find also [86][64][19], which were already seen in Section 1.3.1, since they also allows the users to write their own kernels. While these tools largely facilitate the programming of heterogeneous systems with respect to OpenCL, they have strong limitations. For example, none of them provides arrays that can be seen as a single coherent object across the system because they rely on a host-side representation of each array together with per-device buffers. However, in some approaches such as [86][68] this single object could be not strictly necessary because buffers have defined convenient element-wise access operators. Now, each access involves a memory transfer from the device to the host, incurring in a relevant overhead. In addition, the user has to maintain the coherency of memory of the buffers used among devices and host image. Moreover, these solutions do not expose a suitable labelling mechanism to specify the directions of the arguments of each kernel, being impossible their automatic transfer. This also implies that the data dependencies both between kernels run in different devices or between kernel executions and host accesses cannot be respected without explicit synchronizations. Some of the studied proposals have additional minor limitations. For example, [68] does not support device side functions, while [86] and [104] only support scalars and arrays in global memory in their arguments.

1.4. High level proposals that target multi-device systems

Nowadays, almost every computer with an accelerator is a multi-device system from the point of view of OpenCL because regular CPUs can also be exploited using this standard. In addition, in HPC environments many systems have more than one accelerator. As a result, there have been several high-level proposals to improve the simultaneous use of several devices in heterogeneous applications. Some of them have already been covered in the previous section, as many of the proposals discussed there either already covered the use of multiple devices or naturally evolved to manage multi-device systems, suffering the limitations discussed in Section 1.3. This is for example the case of SkelCL [93], extended to these environments in [92] and in [24]. Another solution based on skeletons that is suitable for multi-device environments is [38], which supports several backends including CUDA and OpenCL. There are also skeletons that have been specifically designed for environments with several accelerators such as [2]. Other frameworks offer a support for multiple devices that has some restrictions or is simply not as convenient as it should ideally be. For example, [86] and [104] are based on the idea of selecting a device, and then operating on it, including the explicit addition of each program to use to each device.

In this category we can also classify PARTANS [73], which is specifically oriented to support stencil computations in heterogeneous environments with several accelerators, as stencils in distributed memory environments require ghost regions that must be kept coherent, which considerably complicates their implementation. This tool automatically optimizes the distribution of data taking into account both the problem and the device characteristics. It presents an interface consisting of two main classes, *Volume* and *Stencil*, which define the elements of a grid and the operation to do with each one of them, respectively.

1.5. Heterogeneous Clusters

The development of applications for distributed memory systems continues to be dominated by MPI, the standard message passing library. Although it is massively used by the scientific community due to its good performance, its programming model is not specially user-friendly, the most important reasons being the local view of the distributed data in its applications and the SPMD style it leads to. There has been much research aimed to hide the cost of this communications layer in order to enjoy the benefits of the heterogeneous computing in clusters enhanced with accelerators. Many proposals [94][35][63][7][61] expand the CUDA and OpenCL paradigms, which are well suited for the management of multiple accelerators in a node, enabling the access to accelerators located in other nodes. As a result, the level of abstraction of these tools is analogous to that of CUDA or OpenCL, which force programmers to manage numerous low level details. Some approaches like [52] avoid some of these tasks with their related boilerplate code, but they still keep many others such as the explicit kernel creation, allocation of buffers associated to devices, event-based synchronizations, etc. A common problem of these approaches is that since they are based on the extension of the CUDA/OpenCL model to access heterogeneous resources across a cluster, they do not provide a solution to the efficient exploitation of the CPUs in the cluster and the communication of the data to be used only in the CPUs. In the OpenCL-based approaches, this could be addressed by writing the CPU codes as OpenCL kernels that are run in those CPUs considering them as OpenCL devices. This clearly requires much more effort than directly using those resources using more abstract solutions instead of mere extensions of the CUDA/OpenCL model.

OmpSs deserves a separate mention, as it targeted traditional clusters [26] before being extended to support heterogeneous ones [27], which enables it to exploit all the parallelism across these systems. However, OmpSs requires users to indicate which are the input and the output parameters of each parallel task. It also lacks distributed structures, which forces programmers to manually partition in chunks the program arrays and to individually specify the computation to perform with each chunk.

1.6. Thesis approach and contributions

As discussed above, the most widely utilized approach to take advantage of heterogeneous systems is the usage of extended versions of well-known languages [80, 62] that reflect and allow for the management of the particular semantics, characteristics and limitations that these accelerators pose for programmers. It deserves to be mentioned, the existence of other approaches such as [12], which are specifically designed for a subset of heterogeneous systems. Portability problems arise from the fact that the vast majority of these programming environments, in fact all of them with the exception of OpenCL [62], are vendor-specific, and sometimes even accelerator-specific. This situation has led to extensive research on ways to improve the programmability of heterogeneous systems. In light of this, as we have seen in the preceding sections, summarized in Table 1.1, researchers have proposed a rich set of libraries, each with different strengths and weaknesses, and compiler directives, whose performance strongly depends on compiler technology.

This Thesis contribute to this research effort. Namely, we focused on facilitating the portable programming of all kinds of heterogeneous systems by means of successive improvements applied on the high level library *Heterogeneous Programming Library* (HPL). At the beginning, this library based on OpenCL, allowed the programming of heterogeneous systems enhanced with only one device by expressing the codes to be accelerated through an embedded C++ language. Using that language, programming only could be single-source, this is, only one source file which integrates the main program code and the one of the accelerator. However, HPL had more limiting factors that hampered the development of generic heterogeneous applications, such as the lack of support for device functions and vector types. Thus,

Proposal	Type	Main drawbacks
ViennaCL [86]	Common Operations / Low level	▪ Maintenance of several coherent views of arrays ▪ Lack of labelling ▪ Overhead of memory accesses
clMath [9]	Common Operations	▪ Only fixed functions and operations allowed
Copperhead [28]	Common Operations	▪ Only fixed functions and complex synchronization
SkePU[38], SkelCL [93]	Skeleton	▪ Computations that only adjust to fixed functionality
PyCUDA/PyOpenCL[64], Thrust [19]	Common Operations	▪ Only one-to-one computations and reductions
OpenACC [81], OpenMP [82]	Compiler Directives	▪ Compiler dependent ▪ Lack of clear performance model
O. Lawlor [68], clUtil [104]	Low level	▪ Maintenance of several coherent views of arrays ▪ Lack of labelling

Table 1.1: Summary of the high level proposals.

starting with that version of the library, we have been solving its limitations along this Thesis following an iterative development by providing new functionalities as the ones listed below:

Native kernels : The initial version of HPL only supported kernels written using its embedded language. That prevented the use of kernels already written using OpenCL, which we call native kernels. In a first iteration, HPL was extended to support them with a very simple and intuitive interface while maximizing the automation of their management [99][101].

Multi-device execution on a single host : HPL allows the efficient use of all the OpenCL devices connected to a host. This way, it allows having multiple copies of the same array, which is represented by the HPL data type *Array*, on several devices by means of a memory management system that maintains automatically the coherency of the copies that can be distributed in several devices simultaneously [100]. This mechanism adapts the data transfers required to maintain this coherency to the device properties in order to reduce the time required to exchange data among devices. In addition, in order to facilitate the work and data division among the devices involved in an execution, HPL defines the *subarray* concept as a constituent part of an Array. A subarray has the entity of an HPL Array that is always kept coherent with the image of the Array it belongs to. Taking advantage of this kind of objects, several work distribution systems were designed in HPL for multi-device environments achieving high levels of programmability in those systems. One of these proposals includes analytical models that automatically divide the work among the devices maximizing the performance and obtaining excellent results in our experiments.

Automatic management of overlapped regions : Stencil computations are operations in which each element of the result is computed based on some of its neighbors in the input array(s). This type of computation is very common in PDE solvers, scientific simulations and image processing applications. The efficient implementation of stencils in multi-device environments requires the existence of replicated regions of the arrays used in the devices, in order to support the computations related with the frontier elements of the region assigned to each device. HPL incorporates a new feature to automatically maintain the coherency of these replicated regions, thus facilitating the implementation of this kind of problems.

Multi-device execution in distributed-memory systems : The next step consisted in facilitating the programming of heterogeneous clusters. This way, HPL has been integrated in a library that improves the programmability of applications on distributed memory systems, namely the *Hierarchically Tiled Arrays* (HTA), resulting in the second main contribution of this Thesis, the *Heterogeneous Hierarchically Tiled Arrays* (H^2TA) library. HTAs allow to exploit the locality as well as to express parallelism with much less effort than lower level solutions, which makes them a suitable base that can serve as a starting point to the development of a solution for heterogeneous clusters. This way, H^2TA allows the execution of HPL applications in a cluster with one or more OpenCL devices per node. With a similar API to HPL, H^2TA simplifies the development of distributed-memory applications with a similar performance and much lower effort than lower level solutions.

This way, the libraries proposed, HPL and H^2TA , allow the development of applications using from a single accelerator to several accelerators in a heterogeneous cluster. In all the cases the resulting programs offer both a noticeable increase of programmability and a negligible overhead with respect to versions based on low level solutions. Moreover, they have been developed in C++, which is a very efficient and widely used language, being in fact one of the most spread languages in high performance computing environments. In addition, the solutions proposed provide the maximum level of portability with respect to the computing devices because of the use of OpenCL, the standard environment for heterogeneous computing supported by most of well-known vendors. In fact, in this Thesis we have included results of tests performed on several types of devices of different vendors. Finally, it deserves to be mentioned that other researchers have also reported positive conclusions of the programmability and performance provided by HPL [45].

Chapter 2

The Heterogeneous Programming Library

In this chapter we present the Heterogeneous Programming Library (HPL), a novel library-based approach to programming heterogeneous systems that couples portability with ease of usage. Our library, which is publicly available under GPL license at `http://hpl.des.udc.es`, allows to express the kernels that exploit heterogeneous parallelism in two different ways: a language embedded in C++, which provides single-source heterogeneous programming, and native OpenCL C kernels. The kernels written using the C++ embedded language allow the library to capture at run-time the computations and variables required for the execution of those kernels. With this information HPL performs run-time code generation (RTCG) in order to run those kernels on the requested device. This feature has important advantages, as it allows to express kernels that adapt to the underlying architecture and problem at hand at runtime, a possibility that has been successfully tested in [43] and [42]. Nevertheless, users may prefer or even require to write their kernels in native OpenCL C for many reasons. For example, they may want to develop or prototype their kernels in OpenCL C so they can later integrate them in another project without adding HPL as another requirement for the project. Programmers may also want to take advantage of OpenCL C kernels provided by several projects [9][86]. Also, users may need to use native OpenCL C kernels because they want to use some of the automatic tuning tools available for them [41][44]. For these reasons, HPL also includes a very convenient mechanism that allows it to use native OpenCL C kernels.

This chapter is organized as follows: we begin with an overview of the hardware and programming model supported by HPL. Then, the chapter continues with a description of the interface and implementation of our library in Section 2.2. An evaluation in terms of programmability and performance for applications using the mechanisms provided by HPL to support kernels is shown in Section 2.3. After this, Section 2.4 discusses on related work and the last section is devoted to our conclusions.

2.1. Programming model

The Heterogeneous Programming Library (HPL) hardware and programming models are similar to those provided by CUDA [80] and OpenCL [62] and they are so general that they can be applied to any computing system and application.

The HPL hardware model, depicted in Fig. 2.1, is comprised of a host with a standard CPU and memory, to which is attached a number of computing devices. The sequential portions of the application run in this host and can only access its memory. The parallel parts, run in the attached devices at the request of the host program. Each device has one or more processors, which can only operate on data found within the memory of the associated device, and which must all execute the same code in SPMD. Processors in different devices, however, can execute different pieces of code. Also, in some devices the processors are organized in groups with

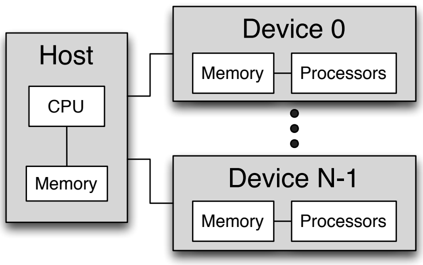


Figure 2.1: Heterogeneous Programming Library hardware model

two properties. First, the processors in a group can synchronize using barriers, while processors in different groups cannot be synchronized. Second, each group of processors may share a small and fast scratchpad memory.

As regards the memory model of the HPL, while no special distinction is made in the host, four kinds of memory can be identified in the devices. First, we have the global memory of the device, which is the largest one, and which can be both read and written by any processor in the device. Second, the scratchpad memory which is local and restricted to a single group of processors is called local memory. Third, a device may have a constant memory, which is read-only memory for its processors, but which can be set up by the host. Finally, each processor can access to a private memory that is exclusive of each one.

As this description of the hardware indicates, HPL applications run their serial portions in the host while their parallel regions run in SPMD mode in the attached devices. While the processors in the same device must all run the same code at a given time, different devices can run different codes. Thus, both data and task parallelism are supported. The parallel tasks are called kernels and they are expressed as functions written in the HPL embedded language. Since the device and host memories are separate, the inputs of a kernel are transferred to its device by the host, and they are provided to the kernel by means of some of its arguments. Similarly, kernels output their results through some of their arguments, which will be transferred to the host when required.

Since multiple threads in a device run the same kernel in SPMD style, an identifier is needed to univocally distinguish each thread. For this purpose, when a kernel is launched to execution in a device, it is associated to a domain of non-negative integers with between one and three dimensions called global domain. An instance of the kernel is run for each point in this domain. In this way, this point is the unique identifier (global id) of the thread, and the domain size gives the number of threads used.

Kernel executions can also be optionally associated to another domain, called local domain, whose purpose is to define groups of threads that run together in a group of device processors able to synchronize and share local memory. The local domain must have the same dimensionality as the global domain, and its size in every dimension must be a divisor of the size of that dimension in the global domain. The global domain can thus be evenly divided in regions of the size of the local domain, so that each region corresponds to a separate thread group whose threads can cooperate thanks to the barriers and the exploitation of the local memory. Each group has a unique identifier based on its position in the global domain (group id). Each thread also has a local id that corresponds to the relative position of its global id within the group's local domain.

As we will see as we develop the description of HPL, in comparison with OpenCL, its backend, HPL avoids the concepts of the context, command queues and commands submitted to the devices. There is no correspondence either for the OpenCL program and memory objects and thus for their management (explicit load and compilation, data transfers, buffer allocation, etc.). Kernel objects are not needed to refer to kernels, just their function name, as in C or C++. There are also issues that OpenCL forces to manage, while HPL can either totally hide or let the user just provide hints for optimization purposes, such as the synchronization between the devices and the host. HPL also brings generic programming capabilities to portable heterogeneous programming, as its kernels and data types support templates. Another interesting feature is that HPL supports multidimensional arrays in the kernel arguments even if their sizes are determined at runtime, giving place to a much more natural notation than the array linearization forced by the usage of raw pointers in OpenCL. Finally, HPL provides run-time code generation (RTCG) that can simplify the generation and selection of code versions at runtime.

2.2. Library frontend

Our library supports the model described in the preceding section, providing three main components to users. First, it provides a template class `Array` that allows for the definition of both the variables that need to be communicated between the host and the devices, and the variables that are local to the kernels. Second, these kernels, as mentioned in the previous section, are functions that can be written using one of the two different approaches accepted in HPL: kernels written using the HPL embedded language (HPL kernels) or kernels can also be written using OpenCL C (native kernels). Finally HPL provides an API for the host code in order to inspect

```
 1
 2 #include "HPL.h"
 3 using namespace HPL;
 4
 5 void mxProduct(Array<float,2> c, Array<float,2> a, Array<float,2> b, Int P)
 6 {
 7   Size_t k;
 8   c[idy][idx] = 0.f;
 9   for_(k =0, k < P, k++)
10     c[idy][idx] += a[idy][k] * b[k][idx];
11 }
12
13 int main()
14 {
15   Array<float,2> c(M,N),a(M,P),b(P,N);
16   ...
17   eval(mxProduct)(c, a, b, P);
18 }
```

Figure 2.2: HPL running example

the available devices, request the execution of kernels and define the properties of native OpenCL kernels. The entire HPL interface is made available by the inclusion of the single header file `HPL.h` and it is encapsulated inside the `HPL` namespace in order to avoid collisions with other program objects. The different components of an HPL application can be seen in the running example shown in Figure 2.2. This example, which is a matrix product, will be used along the HPL introduction. At this point, we will turn to a discussion of the library components.

2.2.1. The `Array` data type

Like any function, kernels in HPL have parameters and private variables (see Line 5 in Figure 2.2). Both kinds of variables must have type `Array<`*type, ndim [, memoryFlag]*`>`, which represents an *ndim*-dimensional array of elements of the C++ type *type*, or a scalar for *ndim=0*. The optional *memoryFlag* either specifies one of the kinds of memory supported (`Global`, `Local` and `Constant`, in the order used in Section 2.1) or is `Private` to a HPL kernel, which specifies that the variable is private to the kernel and which is the default for variables defined inside kernels. The *type* of the elements can be any of the usual C++ arithmetic types or a `struct`. In this latter case, the `struct` definition must be made known to HPL using the syntax shown in Fig. 2.3, where `mystruct_t` is the name we want to give to the `struct`.

When the host code invokes a kernel, it provides the arguments for its execution, which must also be `Array`s. In this way `Array`s must be declared in the host space as global variables or inside functions that run in the host, in order to specify the inputs and outputs of the kernels. These variables, which we call host `Array`s, are initially stored only in the host memory. When they are used as kernel arguments, our library transparently builds a buffer for each one of them in the required device if no such buffer exists yet. The library also automatically performs the appropriate transfers between host and device memory, again only if needed. When a host array or kernel argument declaration specifies no *memoryFlag*, `Global` is assumed. Variables defined inside HPL kernels do not allow the `Global` and `Constant` flags. By default they follow the standard behavior of data items defined inside functions, being thus private to each thread in its kernel instantiation. The exceptions are `Array`s with the `Local` flag, which are shared by all the threads in a group even if they are defined inside a kernel. In Line 15 of Figure 2.2 there are three two-dimensional `Array`s.

While scalars can be defined using the `Array` template class with *ndim=0*, there are convenience types (`Int`, `Uint`, `Float`, `Size_t` ...) that simplify the definition of scalars of the obvious corresponding C++ type (see Line 7 of Figure 2.2). As in native kernels, vector types are also supported both in the HPL kernels (e.g. `Int2`, `Float4`, ...) and the host code (correspondingly `int2`, `float4`, ...). These vectors can be indexed to access their components and manipulated with several functions, including the standard operators. Computations can be performed between vectors as well as between vectors and scalars.

```
HPL_DEFINE_STRUCT( mystruct_t,
                   { int i;
                     float f;
                   } );

Array<mystruct_t, 2> matrix(100, 100);
```

Figure 2.3: Declaring a `struct` type to HPL in order to use it in `Array`s

An important characteristic both of `Arrays` and HPL vector types is that while they are indexed with the usual square brackets in HPL kernels, their indexing in host code is made with parenthesis. This difference visually emphasizes the fact that while `Array` accesses in the host code experience the usual overheads found in the indexing of user-defined data types [46], this is not the case in the kernels. The reason is that HPL kernels are dynamically captured and compiled into native binary code by our library, so that the array accesses have no added overheads.

One reason for the extra cost of the `Array` accesses in the host code is that they track the status of the array in order to maintain a consistent state for the computations. In this way, an array that has been modified by a kernel in a device is refreshed in the host when an access detects the host copy is non-consistent. If the array is written, it is marked as obsolete in the devices. The other crucial point for the maintenance of the consistency is at kernel launch. Input arrays are updated in the device only if there have been most recent writes to them in the host or another device. Also, output arrays are marked as modified by the device, but they are not actively refreshed in the host after the execution. Rather, as explained above, an access in the host will trigger this process. Overall this leads to a lazy copying policy that minimizes the number of transfers.

While this automated management is the default, it can be avoided in order to improve performance. For example, the user may get the raw pointer to the array data in the host through the `Array` method `data` and perform the accesses through the pointer. This method has as an optional argument a flag to indicate whether the array will be read, written or both through the pointer; if not provided, both kinds of accesses are assumed. With this information the host data is updated if

```
Array<float, 1> a(N), b(N);
...
for(int i = 0; i < N; i++)
  a(i) = b(i);
```

(a) automated management

```
Array<float, 1> a(N), b(N);
...
float *pa = a.data(HPL_WRITE);
float *pb = b.data(HPL_READ);

for(int i = 0; i < N; i++)
  pa[i] = pb[i];
```

(b) manual management

Figure 2.4: Usage of `Arrays` in host code

necessary, and the status of the array is correctly tracked. Figure 2.4 illustrates both possibilities. In the case of Fig. 2.4(a) HPL automatically tracks the state of the arrays and makes the required updates, but the check is performed in every access. In Fig. 2.4(b), however, the user explicitly indicates in lines 3 and 4 that `Array a` will be overwritten in the host, while `b` should be brought from the device with the newest version, unless such version is of course the one in the host. Data are then accessed through pointers in line 7, incurring no overhead.

2.2.2. HPL embedded language

The second requirement for writing kernels using its embedded language, after the usage of the HPL datatypes, is to express control flow structures using HPL keywords. The constructs are the same as in C++, but the difference is that an underscore finishes their name (`if_`, `for_`, ...) and that the arguments to `for_` are separated by commas instead of semicolons (see Line 9 in Figure 2.2).

Given the SPMD nature of the execution of kernels, an API to obtain the global, local and group ids as well as the sizes of the domains and numbers of groups described in Section 2.1 is critical. This is achieved by means of the predefined variables displayed in Table 2.1.

The HPL kernels are written as regular C++ functions that use these elements

Meaning	First dimension	Second dimension	Third dimension
Global id	`idx`	`idy`	`idz`
Local id	`lidx`	`lidy`	`lidz`
Group id	`gidx`	`gidy`	`gidz`
Global domain size	`szx`	`szy`	`szz`
Local domain size	`lszx`	`lszy`	`lszz`
Number of groups	`ngroupsx`	`ngroupsy`	`ngroupsz`

Table 2.1: Predefined HPL variables.

```
void saxpy(Array<float,1> y, Array<float,1> x, Float a) {
    y[idx] = a * x[idx] + y[idx];
}
```

Figure 2.5: SAXPY kernel in HPL

and whose parameters are passed by value if they are scalars, and by reference otherwise. For example, the SAXPY (Single-precision real Alpha X Plus Y) vector BLAS routine, which computes $Y = aX + Y$, can be parallelized with a kernel in which each thread `idx` computes `y[idx]`. This results in the code in Fig. 2.5.

The kernel functions can be instantiations of function templates, i.e., C++ functions that depend on template parameters. This is a very useful feature, as it facilitates generic programming and code reuse with the corresponding boost in productivity. In fact, templates are one of the most missed features by OpenCL developers, who can finally exploit them on top of OpenCL, the current backend for our library, thanks to HPL. A small kernel to add two 2-D arrays `a` and `b` into a destination array `c`, all of them with elements of a generic type `T`, is shown in Fig. 2.6. The kernel will be executed with a global domain of the size of the arrays, and the thread with the global id given by the combination of `idx` and `idy` takes care of the addition of the corresponding elements of the arrays.

HPL functions: HPL provides several functions very useful for the development of kernels. For example, `barrier` performs a barrier synchronization among all the threads in a group. It accepts an argument to specify whether the local memory (argument `LOCAL`), the global memory (argument `GLOBAL`) or both (`LOCAL|GLOBAL`) must provide a coherent view for all those threads after the barrier. Fig. 2.7(a) illustrates its usage in a kernel used in the computation of the dot product between two vectors `v1` and `v2`. An instance of the kernel, which is run using groups (local domain size) of `M` threads, is executed for each one of the elements of the vectors so that thread `idx` multiplies `v1[idx]` by `v2[idx]`. The reduction of these values is achieved in two stages. First, a shared vector `vec` of `M` elements located in the local memory stores the partial result computed by each thread in the group. Once the barrier ensures all the values have been stored, the thread with the local id 0

```
template<typename T>
void addmatrices(Array<T,2> c, Array<T,2> a, Array<T,2> b) {
   c[idx][idy] = a[idx][idy] + b[idx][idy];
}
```

Figure 2.6: Generic HPL kernel to add bidimensional arrays of any type

```
#define M 64

void dotp(Array<float,1> v1,
          Array<float,1> v2,
          Array<float, 1> pSum) {
  Int i;
  Array<float, 1, Local> vec(M);

  vec[lidx] = v1[idx] * v2[idx];

  barrier(LOCAL);

  if_( lidx == 0 ) {
     for_( i = 0, i < M, i++ ) {
        pSum[gidx] += vec[i];
     }
  }
}
```

(a) basic manual reduction

```
#define M 64

void dotp(Array<float,1> v1,
          Array<float,1> v2,
          Array<float, 1> pSum) {

  reduce(pSum[gidx],
         v1[idx] * v2[idx],
         "+").groupSize(M).inTree();
}
```

(b) using `reduce` and binary tree reduction

Figure 2.7: Dot product kernels in HPL

reduces them. There are more efficient algorithms to perform this reduction, but our priority here is clarity. The result is stored in the element of the output vector `pSum` associated to this group, which is selected with the group id `gidx`. In a second stage, when the kernel finishes, the host reduces the contents of `pSum` into a single value.

Another example of useful HPL function is `call`, used for invoking functions within kernels. For example, `call(f)(a,b)` calls function `f` with the arguments `a` and `b`. Of course the routine must also be written using the HPL data types and syntax. HPL will internally generate code for a routine and compile it only the first time it is used; subsequent `call`s will simply invoke it. It should be mentioned that routines that do not include a `return_` statement can also be called with the usual `f(a,b)` syntax. The difference is that they will be completely inlined inside the code of the calling function.

This behavior of `call` raises the issue of how HPL kernels are transformed into a binary suitable to run on a given device. This is a two-step process that is hidden from the user. In the first stage, called instantiation, the kernel is run as a regular

C++ code compiled in the host application. The fact that this code is written using the embedded language provided by HPL allows the library to capture all the data definitions, computations, control flow structures, etc. involved in the code, and build a suitable internal representation (IR) that can be compiled, as a second step, into a binary for the desired device. Our current implementation relies on OpenCL C [62] as IR because, as the open standard for the programming of heterogeneous systems, it provides the HPL programs with portability across the wide range of platforms that already support it. There are not, however, any restrictions that preclude the usage of other IRs and platforms as backend. In fact efforts were made in the development of the library to facilitate this possibility, for example by placing most OpenCL-dependent code in a separate module. The aim is for heterogeneous applications written in HPL to have the potential both to preserve the effort spent in their development even in environments where OpenCL is not available and to exploit more efficient backends where possible.

C++ code in HPL kernels: Since the kernel is run as a regular C++ routine during the instantiation, variables of standard C++ types can appear in the kernel. These variables will not appear in the kernel IR; rather, they will be replaced by a constant with their value at the points of the kernel in which they interact with the HPL embedded language elements. By taking advantage of this property, the macro `M` used in lines 7 and 14 of Fig. 2.7(a) and defined as a constant in line 1, could have been instead defined as an external integer variable. The best value for the group size could have been chosen at runtime and stored in this variable before the kernel was instantiated, which happens when it is invoked for the first time. At that point, any reference to `M` in the kernel would be replaced by its actual value in the IR.

For the reasons explained above, standard C++ code, such as computations and control flow keywords, can also appear in kernels. Just as the variables of a type other than `Array`, they will not appear in the IR. In their case, they will simply be executed during the instantiation. In this way, they can be used to compute at runtime values that can become constants in the kernel, to choose among different HPL code versions to include in the kernel or to simplify the generation of repetitive codes. This is illustrated in Fig. 2.8, where `r`, `a` and `b` are 2-D `Array`s of `m`$\times$`n`, `m`$\times$`m` and m×n elements, respectively, and in which m and n are C++ integers chose value is only known at runtime, but remains fixed once computed, and the matrix a is known to be upper triangular. The code computes r=a×b avoiding computations on the zeros of the lower triangle of a. HPL first helps by allowing the direct usage of m and n in the kernel without having to pass them as arguments. If the number of iterations of the innermost loop is above some threshold C, the matrix product is computed using HPL loops whose optimization is left to the backend compiler. Otherwise the code runs the loops in C++ so that they get completely unrolled during the instantiation, which should enhance the performance in GPUs given the properties of these devices. This gives place to $(m \times (m + 1) \times n)/2$ lines of code with the shape of line 11 in the figure, each one with a combination of the values of i, j and k. In CUDA or OpenCL the compiler may have trouble applying this optimization due to the triangular loop, the variable nature of m and n or both, so the programmer would have to perform this tedious and error-prone process by hand. Nevertheless, the HPL user can write the code in Fig. 2.8, knowing that the loops will only run during the instantiation, generating the required versions of line 11 with the appropriate frozen values of i, j and k. These lines will be part of the kernel IR due to the use of the variables of type Array.

As can be seen, regular C++ embedded inside HPL kernels acts as a metaprogramming language that controls the code generated for the kernels. This provides HPL with advanced run-time code generation (RTCG) abilities that simplify the creation of versions of a kernel optimized for different situations as well as the ap-

```
 1  if( ( (m * (m + 1) ) / 2 ) * n > C ) {
 2      Int i, j, k;
 3      for_( i = 0, i < m, i++ )
 4          for_( j = 0, j < n, j++ )
 5              for_( k = i, k < m, k++ )
 6                  r[i][j] += a[i][k] * b[k][j];
 7  } else {
 8      for( int i = 0; i < m; i++ )
 9          for( int j = 0; j < n; j++ )
10              for( int k = i; k < m; k++ )
11                r[i][j] += a[i][k] * b[k][j];
12  }
```

Figure 2.8: Using regular C++ in a kernel to generate an unrolled matrix product

plication of several optimizations. This property is particularly valuable for HPL given the diversity of heterogeneous devices on which the kernels could be run and the high dependence of their performance on the exact codification chosen. In fact, optimized codes for different platforms using the adaptive properties of the HPL kernels were tested and measured in [43] and [42], obtaining very good results. The metaprogramming approach provided by HPL, also called generative metaprogramming [31, 57], is much more powerful than other well-known metaprogramming techniques such as those based on C++ templates [98, 1]. For example, templates are very restricted by the requirement to perform their transformations only with the information available at compile-time. Another problem is their somewhat cumbersome notation (specializations of functions or classes are used to choose between different versions of code, recursion rather than iteration are used for repetitive structures, etc.), which complicates their application. This contrasts with our approach, which takes place at run-time and uses the familiar control structs of C++. Template metaprogramming has been widely used though in the internal implementation of HPL in order to optimize the HPL code capture and the backend code generation, moving computations to compile time whenever possible. Still, most of the process is performed at runtime, although its cost is negligible, as we will see in Section 2.3.

The advantages of RTCG are not only provided by HPL as a feature to be manually exploited by the programmer. Rather, the interface includes facilities to express common patterns of computation whose codification can be built at runtime in order to tailor it to the specific requirements needed. An example is `reduce`, which accepts as inputs a destination, an input value and a string representing an operator and which performs the reduction of the input value provided by each thread in a group using the specified operator into the destination. This routine actually builds an object that generates at run-time the code for the reduction. This object accepts, by means of methods, a number of optional hints to control or optimize the code generated. As an example, the dot product kernel in Fig. 2.7(a) is simplified using this feature in Fig. 2.7(b). In this case, we provide the optional hint that the kernel will be run using groups of `M` threads to help generate a more optimized code. We also request the reduction to be performed using a binary tree algorithm, which often yields better performance than the alternative used in Fig. 2.7(a), at the cost of a more complex codification. As of now `reduce` supports nine code generation modifiers. Other examples of modifiers are requesting a maximum amount of local memory to be used in the reduction process, indicating a minimum group size rather than the exact group size, or specifying whether only one thread needs to write the result in the destination, which is the default, or whether all of them must do it. The object builds an optimized code that tries to fulfill all the requests performed while using the minimum number of resources, computations and thread synchronizations, and inserts it in the kernel. This mechanism is thus equivalent to having a library of an infinite number of functions to perform reductions in a thread group, each one optimized for a specific situation.

HPL kernel analysis: Finally, it should be pointed out that our library does not merely translate the HPL embedded language into an IR in a passive way. On the contrary, during this process the code can be analyzed by the library, which enables it to act as a compiler, gathering information and performing optimizations in the generation of the IR. As of now, HPL does not yet automatically optimize the IR. Nevertheless, kernels are analyzed during the instantiation in order to learn which arrays are only read, only written or both, and in this case, in what order. This information is used by the runtime to minimize the number of transfers required for the kernel and host accesses between the host and the device memories in use without any user intervention, as discussed in the previous section. It also allows to learn the dependences of each kernel submitted to execution, so that HPL automatically ensures they are satisfied before it runs, which results in an automatic and effortless synchronization system. Table 2.2 summarizes the generating process of OpenCL codes.

2.2.3. Host Interface

The most important part of the host interface is function `eval`, which requests the execution of a kernel with the syntax `eval(`*f*`)(`*arg1*, *arg2*, `...)` where *f* is the routine that implements the kernel. As mentioned before, scalars are passed by value and arrays are passed by reference, and thus allow the returning of results.

Specifications in the form of methods to parameterize the execution of the kernel can be inserted between `eval` and the argument list. Two key properties are

Sentence in HPL kernel	OpenCL code generated	Observations
`idx`,`idy`,. . . , reduce	get_global_id([0|1]),. . . , parameterized reductions	
call(g)	Function `g` in OpenCL	Takes into account the Array accesses happened in `g` to modify the direction of those of the caller function `f` passed as arguments to `g`
HPL control flow	OpenCL control flow	
C++ variables	Scalar value is captured in OpenCL code	
C++ control flow sentences		Change the execution flow of the generating process
HPL Array assignments	Assignments in OpenCL buffers	Store/Change the directions of the HPL Arrays used as arguments

Table 2.2: OpenCL generating process.

the global and local domains associated with the kernel run explained in Section 2.1, which can be specified using methods `global` and `local`, respectively. For example, if kernel `f` is to be run on arguments `a` and `b` on a global domain of 100×200 threads with a local domain of size 5×2, the programmer should write `eval(f).global(100, 200).local(5, 2)(a, b)`.

By default the size of each dimension of the global domain corresponds to the size of each dimension of the first argument of the kernel, while the local domain is

```
void saxpy(Array<float,1> y, Array<float,1> x, Float a) {
   y[idx] = a * x[idx] + y[idx];
}

int main(int argc, char **argv) {
   float myvector[1000];
   Float a;
   Array<float, 1> x(1000), y(1000, myvector);

   //the vectors and a are filled in with data (not shown)

   eval(saxpy)(y, x, a);
}
```

Figure 2.9: Array creation and SAXPY kernel usage

chosen by the library. Fig. 2.9 illustrates a simple invocation of the SAXPY kernel of Fig. 2.5, also included in this figure for completeness, by means of its pointer in line 12. The global domain requires one point per element of `y`, which is the first argument, while the local domain needs no specification. The example also shows that host arrays can be created from scratch (`x`), making the library responsible for the allocation and deallocation of its storage in the host, or they can use already allocated host memory by providing the pointer to this memory as last argument to the constructor (`y`). In this latter case the user is responsible for the deallocation too.

Also, our library provides a simple interface to identify and inspect the devices in the system and their attributes (number of threads supported, amount of memory of each kind, etc.) and to obtain a handle of type `Device` to make reference to each one of them. A final method to control the execution of a kernel is `device`, which takes as argument one of these handles in order to choose the associated device for the execution. If none is specified, the kernel is run in the first device found in the system that is not a standard CPU. If no such device is found, the kernel is run in the CPU. The management of the device interface is explained in detail in Chapter 3 as a natural way to introduce the multi-device support in HPL.

The sequence of steps performed by HPL when a kernel is invoked is described in Fig. 2.10. In the first place, an IR of the kernel suitable for the chosen device is sought in an internal cache. If such IR is not found, the kernel is instantiated following the process described in the previous section. Once the required IR is available, it could have been already compiled to generate a binary for the chosen device or not. This is checked in a second cache, which is updated with that binary after the corresponding compilation if it is not found. At this point, HPL transfers to the device those and only those data needed for the execution. This is possible thanks to the information that is automatically tracked on the status of the HPL arrays, and the knowledge of which of the kernel arguments are inputs, which is obtained during the kernel instantiation. As a final step, the kernel is launched for execution.

As we can see in Fig. 2.10, the kernel evaluation request finishes in the host side when the device is asked to run the kernel, without further synchronizations with the device. In this way, HPL kernel runs are asynchronous, i.e., the host

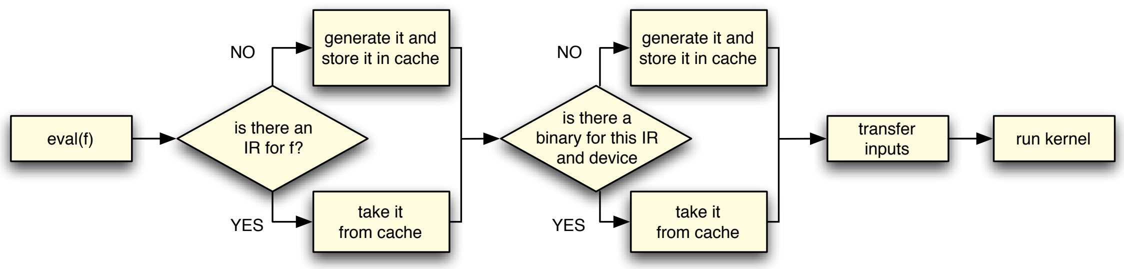

eval(f)
is there an IR for f?
NO
generate it and store it in cache
YES
take it from cache
is there a binary for this IR and device
NO
generate it and store it in cache
YES
take it from cache
transfer inputs
run kernel

Figure 2.10: Kernel invocation algorithm.

does not wait for their completion before proceeding to the next statement. This enables overlapping computations among the host and the device, as well as among several devices in a straightforward way. There are several ways to synchronize with the kernel evaluations. As discussed in Section 2.2.1, whenever the host code accesses an array or submits for execution a kernel that uses it, our runtime analyzes the dependences with preceding uses of the array, enforces them and performs the required transfers. There are also explicit synchronization mechanisms such as the `data` method of `Arrays` or the `sync` method of `Devices`, which waits for the completion of all the kernels sent to the associated device and then updates those that have been modified in the host memory.

Another conclusion from our description of Fig. 2.10 is that kernel instantiations and compilations are minimized, because each kernel is only instantiated the first time it is used, and an IR is only compiled when an associated binary does not exist yet for the chosen device. However, a user might want to reinstantiate a kernel in some situations. For example, as we mentioned in Section 2.2.2, the instantiation could depend on the values of external C++ variables, and the user could be interested in generating several possible instantiations and comparing their performance in order to choose the best one. For this reason, there is a function `reeval` with the same syntax as `eval`, but which forces the instantiation of a kernel even if there were already a version in the HPL caches. Also, our library allows the user to retrieve the string with the IR generated for any kernel, so that it can be inspected and/or directly used on top of the corresponding backend.

Support for native OpenCL C kernels

While the semantics of the HPL embedded language are identical to those of C and its syntax is analogous, users may prefer or need to use native kernels written in OpenCL C for several reasons, the most important one being that this favors code reuse. For this reason HPL provides a convenient interface that requires minimum effort while providing much flexibility. Our proposal requires defining a kernel handle that takes the form of a regular C++ function, and associating it to the native kernel code. After that point, the native kernel can be invoked using regular `eval` invocations on the kernel handle function. These invocations have exactly the same structure and arguments as those of the kernels written in the HPL embedded language, and they also fully automate the buffer creation, data transfer, kernel compilation, etc. that largely complicate OpenCL host codes. The possibility of using the native OpenCL C kernels is the most important contribution to HPL performed in this chapter.

A kernel handle is a regular C++ function with return type `void` (just as all kernels must be), and only its list of arguments matters. In fact its body will never be executed, so it is sensible to leave it empty. The arguments of the handle are associated one by one to the arguments of the kernel that will be associated to it. Namely, each kernel handle function argument must have the HPL type associated to the corresponding OpenCL C native type. This way, OpenCL C pointers of type `T *` will be associated to an `Array<T, n>` where `n` should be the number of dimensions of the underlying array for documentation purposes, although for correct execution it suffices that its value is 1 or greater. By default HPL arrays are allocated in the global memory of the device, so this suffices for OpenCL C pointers with the modifier `__global`. If an input is expected from `__local` or `__constant` memory, then `Array<T, n, Local>` or `Array<T, n, Constant>` must be used, respectively. As for scalars of type `T`, we can use an `Array<T, 0>` or the corresponding convenience type provided by HPL (`Int`, `Double`, . . .).

While following these rules suffices for a correct execution, a kernel function handle defined with these arguments may incur in large overheads. The reason is that by default HPL assumes that the non-scalar arguments are both inputs and outputs of the associated kernel. This guarantees a correct execution, but it results in transfers between the host and the device that are unnecessary if some of those arguments are only inputs or only outputs. Our extension allows to label whether an array is an input, an output or both, so that HPL can minimize the number of transfers and follow exactly the same policies as with the kernels defined with its embedded language. The labeling consists in using the data types `In<Array<...>>`, `Out<Array<...>>` and `InOut<Array<...>>` in the list of arguments of the kernel handle function, respectively.

Once the kernel handle function has been defined, it must be associated to the native OpenCL C kernel code. This is achieved by means of a single invocation to the function `nativeHandle(handle, kernelName, kernelCode)`, whose arguments are the handle, a string with the name of the kernel it is associated to, and finally a string with the kernel OpenCL C code. The string may also contain other code such as helper functions, macros, etc. It helps programmability that HPL stores these strings in a common container, so that if subsequent kernels need to reuse previously defined items, they need not, and in fact should not, be repeated in the string of these new kernels. Also, it is very common that OpenCL kernels are stored in separate files, as it is easier to work on them there than in strings inserted in the host application and it allows to use them in different programs. The price to pay for this is that the application must include code to open these files and load the kernels from them, thus increasing the programmer effort. Our `nativeHandle` function further improves the programmability of OpenCL by allowing its third argument to be a file name. This situation is automatically detected by `nativeHandle`, which then reads the code from the given file. All the information related to the function is stored in a HPL internal structure that is indexed by the handle. The code is only compiled on demand, the first time the user requests its execution. The generated binary is stored in an internal cache from which it can be reused, so that compilation only takes place once. Altogether, `nativeHandle` replaces the IR generation stage explained above, being the compilation stage identical to that of the HPL language kernels. Finally, HPL also offers a helper macro called `TOSTRING` that turns its argument into a C-style string, avoiding both the quotes and per-line string continuation characters otherwise required.

The same saxpy benchmark developed using the HPL embedded language shown in Fig. 2.9 has been transformed to use a native OpenCL C kernel in Fig. 2.11.

The OpenCL kernel, called `saxpy_simple` is stored in a regular C-style string called `kernel_code`, and it is associated to the handle function `saxpy_native`. Notice that since `eval` requires its arguments to be `Array`s, the kernel arguments are defined with this type in the host. Let us remember that it is possible to define them so that they use the data of a preexisting data structure, which facilitates the interface with external code. This strategy has been followed in this example with the `Array` `y`, which uses in the host the storage of the regular C-style matrix `myvector`.

2.3. Evaluation

This section evaluates separately the programmability benefits and the performance achieved by HPL using the C++ embedded language and native OpenCL C kernels. In both cases, the baseline of our studies is OpenCL, since this is the only tool that provides the same degree of portability. Also, as it is the current backend for HPL, the comparison allows for the measurement of the overhead that HPL incurs.

```
const char * const kernel_code = TOSTRING(
  __kernel void saxpy_simple(__global float *y, const __global float *x, const float a)
  {
    y[get_global_id(0)] = a * x[get_global_id(0)] + y[get_global_id(0)];
  } );

void saxpy_native(Array<float, 1> y, In<Array<float, 1>> x, Float a) { }
...
float myvector[1000];
Float a;
Array<float,1> x(1000), y(1000,myvector);
...
nativeHandle(saxpy_native, "saxpy_simple", kernel_code);
eval(saxpy_native)(y, x, a);
```

Figure 2.11: Saxpy using native OpenCL C kernel with HPL

2.3.1. Embedded language kernels

This evaluation is based on six codes, namely the sparse matrix vector multiplication (spmv) and reduction benchmarks of the SHOC Benchmark suite [32], the matrix transpose and Floyd-Warshall codes from the AMD APP SDK, the EP benchmark of the NPB suite [5], and the shallow water simulator with pollutant transport (ShWa) first presented in [102], whose OpenCL version is thoroughly described and evaluated in [71]. The first five codes were already used in a preliminary evaluation in [22]. This study relied on the original OpenCL implementations from the corresponding suites, which include several non-basic routines and use the C interface of the OpenCL framework. Although EP had not been taken from any distribution, the baseline code suffered similar problems. The HPL versions of spmv and reduction also had some unneeded routines inherited from the original OpenCL implementation.

All the codes have been streamlined and cleared for the evaluation performed in this Thesis. The OpenCL baselines have also been translated to C++ in order to use the much more succinct OpenCL C++ interface, so that by avoiding the C++ versus C expressivity difference in the host interface, the programmability comparison is much fairer. The same policies were followed in the translation of the shallow water code from the original CUDA implementation [102]. The result is that now the number of source lines of code excluding comments (SLOC) of our OpenCL baselines is up to 3.3 times smaller than in [22], as Table 2.3 indicates. The HPL codes were also improved with features implemented after the publication of [22], such as the customized `reduce` mechanism described in Section 2.2.2.

Table 2.3 further characterizes the benchmarks indicating whether their kernels

Benchmark	SLOCs OpenCL	Routines	Repetitive invocation	Cooperation	Arithmetic intensity
Spmv	500			medium	low
Reduction	399		1 kernel	high	low
Matrix transpose	373			low	low
Floyd-Warshall	407		1 kernel	no	low
EP	605	✓		low	high
Shallow water	965	✓	3 kernels	low	high

Table 2.3: Benchmarks characteristics.

use subroutines, whether there is a single kernel invocation or repetitive invocations (and in this case of how many kernels), the degree of cooperation between the threads in the kernels and the arithmetic intensity. The repetitive invocation of kernels is interesting for the analysis of the cost of the kernel executions and synchronizations with the host, including the effectiveness of the mechanisms to avoid unneeded transfers between host and device memory. Reduction and Floyd-Warshall repetitively invoke in a loop a single kernel, while the shallow water simulator performs a simulation through time in a sequential loop in which in each iteration three different kernels are run one after another, there being also computations in the host in each time iteration.

The cooperation column qualitatively represents the weight of synchronizations and data exchanges between threads in the kernels. For example, in spmv each thread first performs part of the product of the compressed row of a matrix by a vector, and then performs a binary tree reduction with other threads in its group to compute the final value for the row. The reduction benchmark focuses intensively in reductions that imply continuous data sharing and synchronization among threads. In matrix transpose, each thread group loads the local memory with a sub-block of the matrix to transpose, then synchronizes once with a barrier, and finally copies the data from the local memory to the transposed matrix. In Floyd-Warshall, each thread performs its own computations without the use of local memory or barriers. In EP, each thread runs the vast majority of the time working on its own data, there being a final reduction of the results of each thread. The situation is similar in the shallow water simulator, in which threads only need to cooperate in a reduction in the most lightweight kernel.

Finally, the arithmetic intensity, which measures the ratio of computations per memory word transferred, is a usual indicator for characterizing applications run in GPUs. We consider low arithmetic intensity if the number of computing instructions is similar to the number of memory instructions. On the contrary, the arithmetic intensity is high if the ratio of computing instructions per each memory instructions is high. Due to the much higher cost of memory accesses compared to computations in these devices, high arithmetic intensity is a very desirable property for GPGPU computing. As can be seen in Table 2.3, our evaluation relies on codes with a wide range of different characteristics.

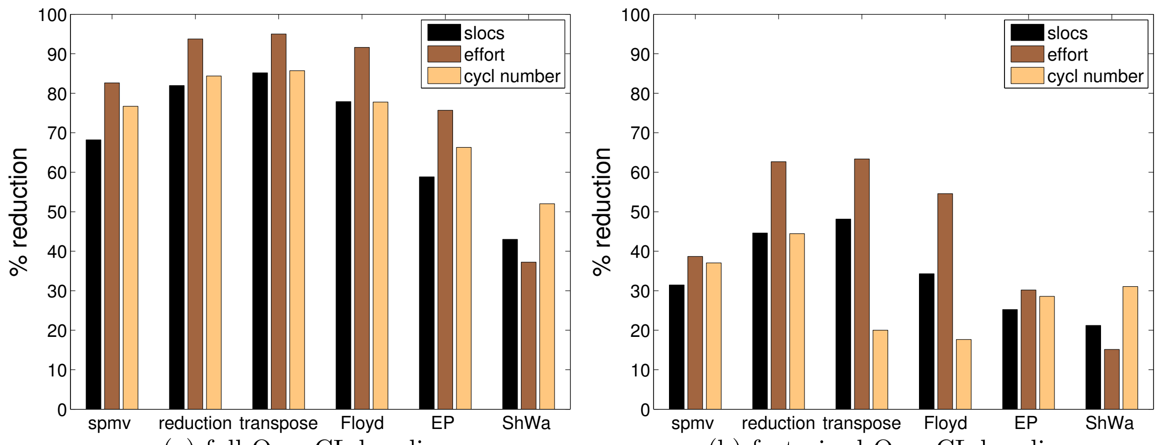
Programmability analysis

Productivity is difficult to measure, thus most studies try to approximate it from metrics obtained from the source code such as the SLOCs. Lines of code, however, can vary to a large extent in terms of complexity. Therefore, other objective metrics have been proposed to more accurately characterize productivity. For example, the programming effort [55] tries to estimate in a reasoned way the cost of developing a code by means of a formula that takes into account the number of unique operands, unique operators, total operands and total operators found in the code. For this, it regards as operands the constants and identifiers, while symbols or combinations of symbols that affect the value or ordering of operands constitute the operators. Another indicator of the complexity of a program is the number of conditions and branches it contains. Based on this idea, [77] proposed as a measure of complexity the cyclomatic number $V = P + 1$, where P is the number of decision points or predicates.

Figure 2.12 shows the reduction of SLOCs, programming effort [55] and the cyclomatic number [77] of HPL with respect to an OpenCL implementation of the considered benchmarks. Figure 2.12(a) takes as the baseline an OpenCL program including the initialization code to choose a suitable platform and device, build the context and command queue used by this framework, and load and compile the kernels. The initialization code is written in a very generic way, so that it maximizes portability by supporting environments with multiple OpenCL platforms installed and/or several devices, and it controls all the errors that can appear during the process. The code is in fact taken with minor adaptations from the internals of HPL in order to provide exactly the same high degree of portability and error control. This is the OpenCL version whose SLOCs appear in Table 2.3. Nevertheless, the initialization of the OpenCL environment as well as the loading and compilation of the kernels can be easily placed in routines that can be reused across most applications, thus avoiding much programming cost. For this reason Fig. 2.12(b) takes as the baseline for the improvement in productivity metrics provided by HPL a factorized OpenCL code that replaces with a few generic routine calls this heavy initialization of ∼270 SLOCs. The two baselines considered thus represent a reasonable maximal and minimal programming cost of the OpenCL version of each application, even if the minimal one is somewhat unfair to HPL, as the removed code has still to be written at some point.

Even compared to the more efficiently programmed OpenCL version, HPL reduces the SLOCs between 21% and 48%, the programming effort between 15% and 63% and the cyclomatic number between 18% and 44%. While these numbers are very positive, complexity measurements on the code do not tell the whole story. Our experience when programming with the HPL embedded language is that it speeds up the development process in two additional ways not reflected in the code. The first way is by moving the detection of errors to an earlier point. Concretely, since OpenCL kernels are compiled at runtime, the application needs to be recompiled (if there are changes in the host code), sent to execution, and reach the point where the kernel is compiled to find the usual lexical and syntactical errors, fix them and repeat the process. With HPL the detection of the most common problems of this kind (missing semicolons, unbalanced parenthesis, mistyped variables, . . .) happens right at the compilation stage, as in any C++ program. Besides in many integrated development environments (IDEs) the integration of the compiler with the editor allows quickly going through all the errors found by the compiler and fix them. We have seen a productivity improvement thanks to the faster response time enabled by HPL.

The second way how the HPL embedded language further improves productivity is by providing better error messages. This way, sometimes the error messages



(a) full OpenCL baseline (b) factorized OpenCL baseline

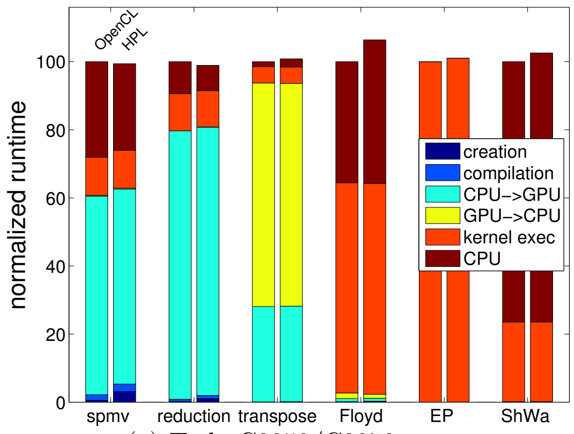
Figure 2.12: Productivity metrics reduction in HPL with respect to two OpenCL baseline implementations

obtained from some OpenCL compilers were not helpful. For example, some detectable at compile or link time errors, such as invoking a nonexistent function due to a typo, were reported using a line of a PTX assembly file, without any mention of the identifier or line of OpenCL where the error had been made. Obviously, this is an obstacle to the productivity of the average user who has to track the source of this problem and fix it. With HPL, the C++ compiler always clearly complains about the unknown identifier in the point of the source code where it is referenced or, in the worst case, when the error is detected during linking, at least it indicates the object file and name of the missing function, largely simplifying the programmer's work.

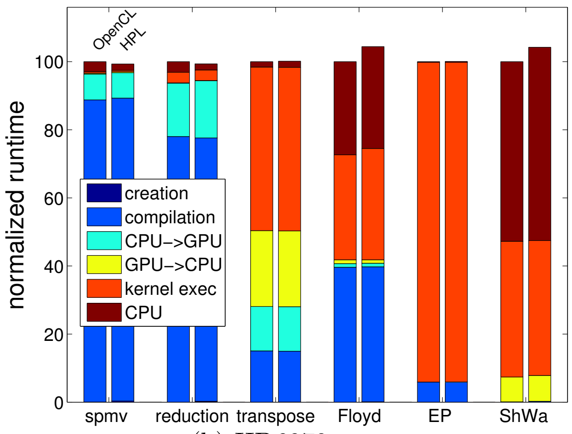
Performance analysis

This section compares the performance of the baseline OpenCL applications with those developed in HPL in two systems. The first is a host with 4xDual-Core Intel 2.13 GHz Xeon processors that are connected to a Tesla C2050/C2070 GPU, a device with 448 thread processors operating at 1.15 GHz and 3GB of DRAM. This GPU operates under CUDA 4.2.1 with an OpenCL 1.1 driver. In order to evaluate the very different environments and test the portability of the applications, the second machine selected was an Intel Core 2 at 2.4GHz with an AMD HD6970 GPU with 2 GB of DRAM and 1536 processing elements at 880 MHz operating under OpenCL 1.2 AMD-APP. The applications were compiled with g++ 4.7.1 using optimization level O3 on both platforms.

The performance of OpenCL and HPL applications is compared for the NVIDIA and AMD GPU based systems in Fig. 2.13(a) and Fig. 2.13(b), respectively. The runtime of both versions was normalized to that which was achieved by the OpenCL version and it was decomposed in six components: kernel creation, kernel compilation, time spent in CPU to GPU transfers, time required by GPU to CPU transfers, kernel execution time, and finally host CPU runtime. The kernel creation time accounts in the OpenCL version for the loading of the kernel source code from a file, as this is the most usual approach followed, particularly for medium and large kernels. In the HPL columns, it corresponds to the time our library required to build the kernel IR from the C++ embedded language representation. The other portions of the runtime correspond to the same basic steps in both versions. The measurements



(a) Tesla C2050/C2070 system



(b) HD6970 system

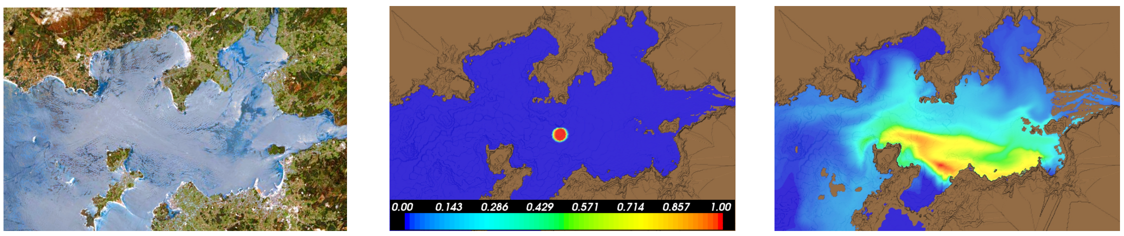
Figure 2.13: Performance of the OpenCL and the HPL versions of the codes, normalized to the runtime of the OpenCL version

were made using synchronous executions, as most profilers do for accuracy reasons, thus there was no overlapping between host computations and GPU computations or transfers. It should be pointed out that it is particularly easy to obtain this detailed profiling for the HPL codes because when our library and the application are compiled with the flag `HPL_PROFILE`, HPL automatically gathers these statistics for each individual kernel invocation as well as for the global execution of the program. The user can retrieve these measurements by means of a convenient API.

The experiments consisted of multiplying a 16K×16K sparse matrix with a 1% of nonzeros by a vector in spmv, adding 16M values in reduction, transposing a 8K×8K matrix, applying the Floyd-Warshall algorithm on 1024 nodes, running EP with class C, and finally simulating the evolution of a contaminant during one week in a mesh of 400 × 400 cells that represents an actual estuary (*Ría de Arousa*, in Galicia, Spain) using real terrain and bathymetry data. The input and the visual representation of the results of this real-world application are illustrated in Fig. 2.14(a), with the Google Maps satellite image of the region where the simulation takes place, the illustration of the initial setup in Fig. 2.14(b), in which the contaminant is concentrated in a small circle with a radius of 400m., and Fig. 2.14(c) where we see how it evolved after eight days via a color scale which indicates the normalized concentration of the pollutant. All the benchmarks operate on single-precision values, the exceptions being Floyd-Warshall, which works with integers, and EP, which is based on double-precision floating point computations. It should also be mentioned that the shallow water simulation kernels largely rely on vector types both in the OpenCL and HPL versions.

The most relevant conclusion that can be drawn from Fig. 2.13 is that the performance of HPL applications is very similar to that of the corresponding native OpenCL code. The average slowdown of HPL with respect to OpenCL across these tests was just 1.5% and 1.3% in the NVIDIA and AMD GPU based systems, respectively. The maximum overhead measured has been 6.4% for Floyd-Warshall in the NVIDIA system, followed by a 4.4% for this same algorithm in the AMD system, and it is mostly concentrated in the CPU runtime in both cases. The reason is that this application launches 1024 consecutive kernel executions of very short length (0.1 ms) to the GPU, without any array transfer (only a scalar is sent), and unsurprisingly the HPL runtime incurs in additional costs in the kernel launches with respect to the OpenCL implementation. This was also the main overhead found in the HPL versions of the shallow water simulator, as it is the other application that launches many kernel executions. At this point it is relevant to remember that the measurements were taken using synchronous executions for the benefit of the detailed analysis of all the execution stages. However, HPL runs by default in asynchronous mode, which enables partially overlapping this overhead with GPU computations and transfers. This way, the overhead, in a non-profiled run of HPL with respect to an OpenCL implementation of Floyd-Warshall that also exploits asynchronous execution, is 5% and just 0.44% in the NVIDIA and AMD systems, respectively.

It is interesting to note in Fig. 2.13 that the same code spends its runtime in quite different activities in the two platforms tested. For example, compilation consumes much more resources in the AMD than in the NVIDIA system. Also, the kernel creation time is always negligible.



(a) satellite image (b) pollutant drop (c) situation after 8 days

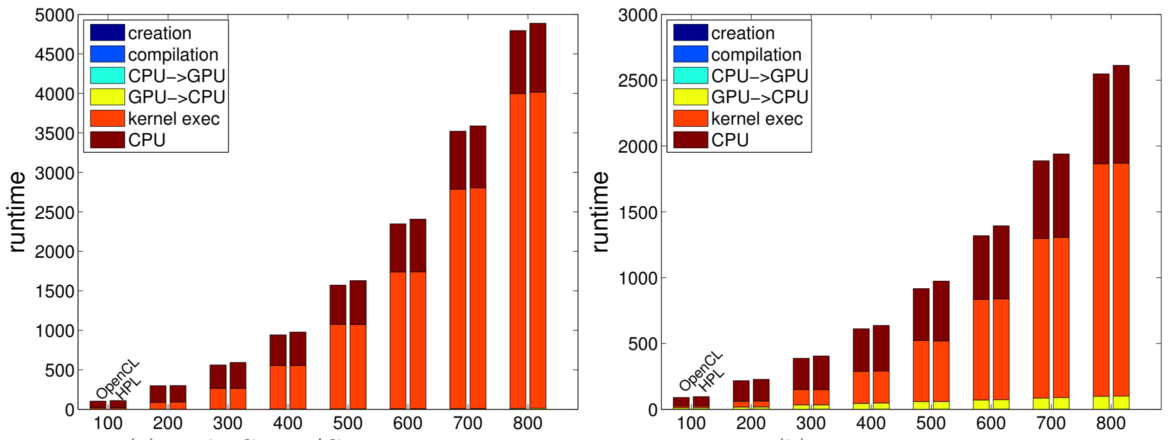
Figure 2.14: Simulation of evolution of a pollutant in Ría de Arousa

While our sparse matrix vector product, reduction, matrix transpose and Floyd-Warshall algorithm baseline OpenCL codes are existing works taken from well-known external sources, this is not the case for NAS Parallel Benchmark EP and the shallow water simulator, which we have developed ourselves. Thus it can be interesting to compare these baselines with other works in order to evaluate their quality. Although it is not feasible to find another shallow water simulator with exactly the same characteristics, the quality of our OpenCL implementation can be assessed in our recent publication [71]. As for EP, Table 2.4 shows the total runtime for problem size C of the SNU NPB suite [89] EP and the EP we developed in the two platforms tested, both when written in OpenCL and in HPL. We can see that HPL has a minimal overhead of around 0.85-1% for both EP versions in the NVIDIA system, which drops to 0.5% in the AMD GPU. Regarding the performance of our implementation, it is competitive with respect to the SNU NPB implementation, and in fact it outperforms it by a small margin of 5.7% and 2.4% in the NVIDIA and AMD systems, respectively. As a result, our HPL version slightly outperforms the SNU OpenCL native implementation in both platforms.

The runtime of the OpenCL and HPL versions of the shallow water simulator is shown for varying problem sizes in the two platforms considered in Fig. 2.15. This code was chosen because it is the largest and unlike the others has several kernels, which are invoked repetitively during the simulation, and also because it is an actual complete application. The figure shows the runtimes for mesh sizes from 100×100 to 800×800 in steps of 100 cells. The runtimes of the OpenCL version went from 103.5 and 90 seconds for the smallest mesh, to 4794 and 2548 seconds for the largest in the NVIDIA and AMD systems, respectively. In all of the cases the runtime was mostly dominated by the execution times of the kernels, followed by the operations in the host CPU. The periodic transfers of data from the GPU to CPU are only noticeable in the AMD system. The runtimes were very similar for both versions

Benchmark	Tesla C2050/C2070		HD6970	
	OpenCL	HPL	OpenCL	HPL
SNU NPB EP	2.905	2.930	4.513	4.536
locally developed EP	2.745	2.772	4.408	4.428

Table 2.4: NAS Parallel Benchmark EP runtimes for class C (in seconds)



(a) Tesla C2050/C2070 system (b) HD6970 system

Figure 2.15: Runtime of the OpenCL and the HPL versions of the shallow water simulator for different problem sizes

for all the problem sizes, the average slowdown of HPL with respect to OpenCL being 3.4% and 4.6% in the NVIDIA an AMD GPU based systems, respectively. The HPL overhead is concentrated in the host CPU usage implied by its runtime. As we previously explained, this is a maximal bound of the actual overhead found in a non-profiled run, in which the asynchronous execution between host and device hides part of it.

2.3.2. Native OpenCL C kernels

The evaluation of the native OpenCL feature of HPL is based on three codes of the SNU NPB suite [89] (FT, IS and EP) and the same shallow simulator used in Section 2.3.1, which was developed in [102]. In these tests, we also ported the codes from C to C++, so that our baselines use the more succinct C++ OpenCL host API, which exploits all the advantages of this language such as its object orientation. This way the language characteristics play a neutral role in the comparison. As in the evaluation of the embedded language kernels illustrated in Figure 2.12(b), for this evaluation we have also encapsulated the initialization of OpenCL (platform and device selection, creation of context and command queue, loading and compilation of kernels) in routines that can be used across most applications and replaced these tasks with invocations to these common routines, so that they are not part of the evaluation. As a result our baseline corresponds to the bare minimum amount of code that a user has to write for these applications when using the OpenCL host C++ API.

Table 2.5 summarizes the most relevant characteristics of the baseline benchmarks with respect to the productivity evaluation. For each benchmark the number of source lines of code (SLOCs) excluding comments and empty lines for the host side code, the programming effort [55] of the host side code, the SLOCs of its kernels, the number of kernels and the number of arrays found in the arguments of the invocations of those kernels are listed. The SLOCs of the kernels are only given for informative purposes, as the changes only affect the host side of the applications. Finally, the number of kernels and related host-side arrays are relevant to interpret the productivity results, since once the usual initialization tasks required by OpenCL have been reduced to the minimum expression in the host code, the OpenCL related activities left basically focus on the creation of buffers, parameterization and execution of kernels, and the transfers between the device and the host.

In order to better gauge the advantages of HPL our evaluation includes ViennaCL [86] because it is one of the best alternatives for improving the usability of OpenCL in C++ and it is a live and well supported project. We had to make some adjustments in applications with kernels that required local memory arrays and OpenCL vector types in their arguments to adapt them to ViennaCL because it does not support these possibilities.

Figure 2.16 shows the reduction of the SLOCs and the programming effort of the host side of our baseline applications when they are written with ViennaCL and HPL. The last group of columns represents the average reduction. Even when our baseline enjoys the C++ OpenCL API and the common boilerplate code required by OpenCL has been factored out, ViennaCL and HPL still provide average noticeable

Table 2.5: Benchmarks characteristics.

Benchmark	SLOCs host	Effort host	SLOCs kernels	Number of kernels	Number of arrays
FT	641	6118988	567	8	8
IS	394	2705245	571	11	12
EP	163	469038	238	1	2
ShWa	186	893085	343	3	6

reductions of the programming cost metrics between 20% and 42%. Interestingly, the effort, which is more proportional to the actual programming cost than SLOCs, is the metric that obtains the largest reductions in all the benchmarks. Finally, HPL reduces SLOCs and particularly effort stronger than ViennaCL. It must be mentioned that the kernels were included in the host as a string. If they had been loaded from files, the automatic file loading feature of HPL would have allowed it to further improve programmability with respect to the other alternatives.

We also measured the overhead of ViennaCL and HPL with respect to the OpenCL C++ API in an NVIDIA Tesla Fermi 2050 with 3GB whose host has a Intel Xeon X5650 (6 cores) at 2,67GHz and 12GB RAM, and an Intel Xeon Phi with 60 cores at 1.056 GHz and 8GB with a host with 2 Intel Xeon CPU E5-2660 (8 cores per CPU) at 2.20GHz with 64GB RAM. The compiler was g++ 4.7.2 with optimization level `-O3`. The ViennaCL and HPL runtimes use the same strategies and functions as our optimized OpenCL baselines for data and kernel management, thus any time difference is related to the overheads of these libraries. We run very small tests, with class S for FT, IS and EP, and a 100×100 mesh for ShWa to measure the overhead in the worst conditions, i.e., when the portion of the runtime associated to the OpenCL management (not the kernel runs or host computations) is maximal. We also run more representative tests using FT class B, IS class C, EP class C and ShWa on a mesh of 500×500 cells. The three versions achieved the

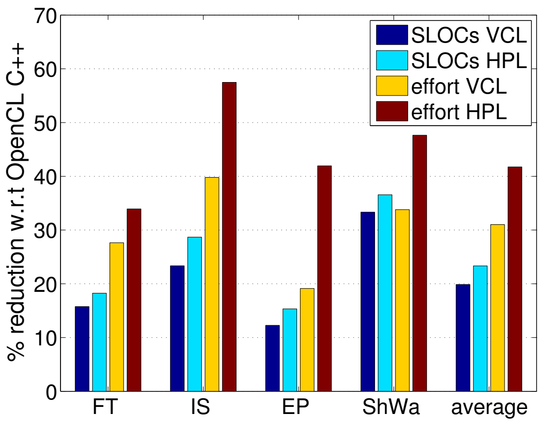


Figure 2.16: Productivity improvement in ViennaCL and HPL with respect to the baseline

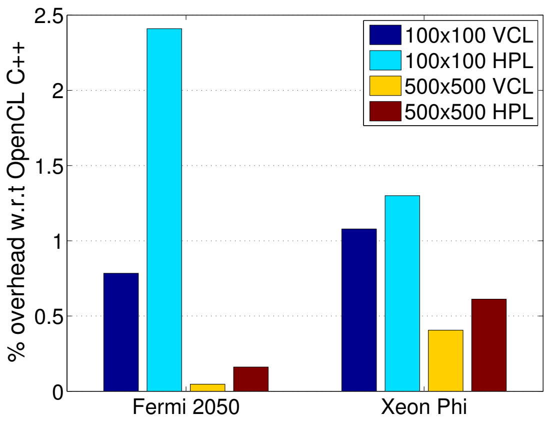


Figure 2.17: Overhead of ViennaCL and HPL in ShWa with respect to the baseline

same performance in both platforms for every benchmark except ShWa. The reason is the very large number of kernel runs of this code, 492729 for the small test and 2517711 for the medium one, which allows to accumulate some overhead, shown in Fig. 2.17. As expected the larger automation of HPL generates more overhead than ViennaCL. However this is still very small, and it only reaches 2.5% on a baseline execution of just 17.86 s. in the GPU. For the more representative 500×500 runs this overhead falls to a negligible 0.16% and 0.61% in the GPU and the Xeon Phi, respectively.

2.4. Related work

Much research has been devoted to improve the programmability of heterogeneous systems, particularly since the rise of modern hardware accelerators. This way, CuPP [23] and EPGPU [68] facilitate the usage in C++ programs of CUDA [80] and OpenCL [62], respectively, by providing a better interface and a runtime that takes care of low level tasks such as memory management and kernel invocation. A higher degree of abstraction is provided by CUDPP [88], a library of data-parallel algorithm primitives that can only run a predefined set of operations and only in CUDA-supported devices. ViennaCL [86] mainly focuses on a convenient C++ API for running linear algebra operations on top of OpenCL, although it also simplifies the execution of custom kernels provided as strings in OpenCL C.

Thrust [19] provides an interface inspired in STL to perform operations on 1D vectors in either CPUs of CUDA-supported GPUs. Its user-defined operations are restricted to being one-to-one, that is, each element of the output vector is computed using a single value from each input vector and the user cannot control basic execution parameters such as numbers of threads or kinds of memory to use.

SkePU [38] and SkelCL [93] further explore the idea of using skeletons to express computations in heterogeneous systems. They can run on top of OpenCL (SkePU also supports CUDA and OpenMP) and they support up to 1D (OpenCL) or 2D (SkePU) arrays. Their skeletons accept user functions in the form of strings for OpenCL, or class member functions for the CUDA and OpenMP backends. However, since these latter functions must be representable as strings, they have in practice the same restrictions as strings. In this way, contrary to HPL kernels, which can capture external variables and perform RTCG even under the control of the programmer, their code must be constant at compile time and include all the definitions of the values they use. An additional restriction in the case of SkePU is that since all its backends use the same user function code, only the common denominator of the language supported by all the backends can appear in the user code, which can preclude many important optimizations. These libraries, which also support the usage of multiple GPUs in a straightforward way, are excellent option to run in heterogeneous devices for those computations whose structure naturally conforms to one of their skeletons.

The PyCUDA and PyOpenCL [64] toolkits simplify the usage of hardware accelerators in the high-level scripting language Python to perform many predefined computations. Custom user functions in the form of strings are also supported, although they are restricted to element-to-element computations and reductions. These projects also emphasize RTCG, although in their case it is based on string processing in the form of keyword replacement, textual templating and syntax tree building. These approaches require learning a new, and sometimes quite complex, interface to perform the corresponding transformations. This contrasts with the natural integration of HPL kernels in C++ and the direct and simple use of this language to control RTCG.

The kind of RTCG provided by HPL is supported by TaskGraph [18] and Intel Array Building Blocks (ArBB) [78] because they also build at runtime their kernels from a representation using a language embedded in C++. TaskGraph combines code specialization with runtime dependence analysis and restructuring optimizations. It has been used to build active libraries that can compose and optimize sequences of kernels [87], and while it exposes no parallel programming model, its authors have explored parallel schemes using it. Regarding ArrBB, it only targets multicore CPUs, however, and it has a very different programming model, with special instructions to copy data in and out of the space where the kernels are run and does not offer the possibility of controlling the task granularity, optimizing the memory hierarchy usage or cooperating between parallel threads.

Copperhead [28] is an embedded language that allows the exploitation of heterogeneous devices, although only NVIDIA GPUs, in order to run computations expressed with data-parallel primitives and a restricted subset of Python, which is its host language. It is a powerful tool for expressing computations that adjust to the usual data-parallel abstractions and in which all the code generation and execution parameters are transparently controlled by the Copperhead runtime. This results in a high level of abstraction that benefits programmability, but which provides little or no programmer control on the result, which largely depends on the ability of the compiler. These characteristics are typical of compiler directives, which have been also explored in the area of heterogeneous programming [20, 69, 56, 81]. The number of directives and clauses that some of these approaches require to generate competitive code is sometimes on par with or even exceeds the programming costs of library-based approaches. More importantly, the lack of a clear performance model [79] and the suboptimal code generated by compilers in many situations have already led to the demise of promising approaches of this kind such as HPF [58]. The state of affairs is even worse in the case of heterogeneous systems because they allow for more possible implementations for the same algorithm, they have a large number of parameters that can be chosen, and their performance is very sensitive to small changes in these parameters.

The native OpenCL C kernels support in HPL proposed in this section has several advantages and provides a higher-level view with respect to the related proposals we know of. This way, it is the only one that provides arrays that are seen as a single coherent object across the system, as the other solutions rely on a host-side representation of the array together with per-device buffers. While it is possible to avoid the host side representation for the buffers in [86][68] because they provide random element-wise accesses, each one of such accesses involves a transfer between the host and the device, and due to the enormous overhead, this is only very seldom a reasonable solution. In addition, these buffers are not kept automatically coherent with their host image or with the buffers that represent the same data structure in other devices. Rather, they must be explicitly read or written. This makes sense because these proposals do not provide a mechanism to label which are the inputs and the outputs of each kernel, so their runtime cannot automate the transfers. For similar reasons, it is impossible for them to automatically enforce data dependences between kernels run in different devices, or between kernel executions and arrays accesses in the host, unless by considering the most conservative, and therefore suboptimal, assumptions. Regarding devices, [68] only supports a single device, while [86][104] are based on the idea of selecting a current device, and then operating on it, including the explicit addition of each program to use to each device. Also, [68] does not allow to define auxiliary functions, but only kernels, while [86][104] do not support local or constant memory arrays in the arguments.

2.5. Conclusions

Heterogeneous systems are becoming increasingly important in the computing landscape as a result of the absolute performance and performance per watt advantage that devices such as GPUs achieve with respect to the standard general-purpose CPUs for many problems. Nevertheless, an improvable aspect of these systems is their programmability and the portability of the codes that exploit them. This chapter addresses these issues proposing the Heterogeneous Programming Library (HPL), which provides large portability thanks to being based on the OpenCL standard and whose most characteristic component is a language embedded inside C++ to express the computations (kernels) to run in heterogeneous environments. This language allows our library to capture the kernels so that it can translate them into a suitable IR that is then compiled for the device where they will be run. The host C++ language can be naturally interleaved with our embedded language in the kernels, acting as a metaprogramming language that controls the code generated using a syntax that is much more convenient and intuitive than other metaprogramming approaches such as C++ templates. This results in a very powerful run-time code generation (RTCG) environment that is particularly useful for heterogeneous systems, in which users often need to tune the kernels to the specific characteristics of each device to achieve good performance. HPL also provides very convenient interfaces to exploit RTCG to generate highly specialized code for common patterns of computation such as reductions.

During the generation of the IR for a kernel, our library has the opportunity of analyzing and potentially optimizing it, as a compiler would do. While our current implementation performs no code transformations, it does analyze the kernels in order to determine their inputs and outputs. This information allows HPL to track the data dependences between the tasks that the user requests to run in the devices exploiting the asynchronous execution model of the library, as well as between these tasks and the host. This way HPL provides automatic task synchronization while minimizing the number of data transfers. In a related manner, HPL provides rather handy classes to represent data for use in the heterogeneous kernels whose management (creation and deallocation of buffers, tracking of the state and synchronization as required among physical buffers associated with the same logical array in different memories, etc.) is completely automated by our library. All these advantages of high automation of the management and ease of use are also enjoyed by HPL for kernels written in OpenCL C, which our library supports by means of a simple notation.

The HPL embedded language has however a number of programmability advantages that are not available using native OpenCL kernels. The ability to exploit C++ templates in kernels, the detection of errors at compile time, at times with clearer messages, the natural embedding in the kernels of runtime constants, and the transparent indexing of multidimensional arrays when using this language further boost HPL programmer productivity.

An evaluation using codes with quite different natures and taken from different sources indicates that the HPL embedded language provides significant programmability improvements with respect to OpenCL while achieving nearly the same performance in different platforms. In fact, even if we take as the baseline a streamlined version of OpenCL codes in which the initialization and program compilation stages typical of this platform have been removed, the average reduction in terms of SLOCs, programming effort and cyclomatic number achieved by the HPL embedded language are 34%, 44% and 30%, respectively. Nevertheless, the typical performance overhead is below 5%.

The usage of HPL to execute standard OpenCL C kernels has also important programmability benefits. This way, even when in our tests the baseline used the minimum amount of code and the OpenCL C++ bindings, our proposal reduced the number of lines and the programmer effort of the host code by a notable 23% and 42%, respectively, while imposing a totally negligible overhead on performance. Also, its productivity metrics, and particularly the programming effort, were consistently better than those of a comparable powerful approach such as ViennaCL [86].

Chapter 3

Multi-device computing

In the previous chapter, HPL was presented as a way to simplify the programming of heterogeneous systems in single-device environments. In this chapter we extend this tool to exploit multiple devices while keeping its characteristics of minimum user effort and maximum performance. In a first step we extend HPL with a totally general data coherency scheme for the data structures it manages as well as a mechanism to make assignments between these structures so that they can be easily copied. The implementation is efficient, as it not only requires the minimum number of transfers, but it also applies the most efficient mechanisms to perform these transfers. This latter characteristic implies a dynamic adaption capability of our library, as different transfer mechanisms suit better different systems. In a second stage the host API is improved with three mechanisms to reduce the programming effort of applications that exploit several heterogeneous devices. A first idea is the ability to use subarrays, that is, regions of arrays, that can be used in one or several devices without becoming separate data structures, so that it is always possible to keep a view of the underlying full array, while automatically keeping the consistency of the data. The second idea are mechanisms to split kernels for their execution in several devices, giving place to what we call subkernels. Our experiments show that these mechanisms can largely improve programmability, their overhead being minimal. The last idea is an alternative to our initial subkernel proposal that is more flexible and allows to easily explore and choose the best workload distribution when a computation is accelerated using devices with different capabilities. This proposal, which includes analytical models capable of choosing optimal or near-optimal work distributions, not only helps programmability but also performance. Finally, we propose a mechanism to address stencil computations in multi-device environments in a very convenient way in HPL that is focused on the management of the required ghost regions. Thanks to it, HPL can automatically detect and update ghost regions, thus freeing the user from the tedious and error-prone tasks involved by the manual update so that she only needs to specify when to perform the update. The addition of this automatic system noticeably improves the programmability without involving any penalty with respect to the HPL implementations based on manual updates.

The rest of this chapter is organized as follows. Section 3.1 presents the novel data coherency scheme and a naïve host API to support multi-device computing. Section 3.2 explains the improvements done in both the host API and the runtime to increase the programmability and performance, respectively. The mechanism to automatically detect and update the ghost regions is explained in depth in Section 3.3. This is followed by the evaluation in Section 3.4 and the conclusions are closing the chapter.

3.1. Multi-device support in HPL

The exploitation of multiple devices requires several features of the HPL library: support in its API for multiple devices, the improvement of its coherency and synchronization scheme, and an easy syntax to copy data between `Arrays`. These features are first presented in turn, while implementation details are discussed in Section 3.1.1.

Multi-device support in the HPL API: The HPL API allows to identify the devices that can be used and their characteristics. This characteristic is necessary to enable multi-device support. HPL currently classifies the devices as either CPUs, GPUs or generic accelerators (this is for example the case of the Xeon Phi). The user can obtain the number of devices of each kind (e.g. `getDeviceNumber(GPU)` provides the number of GPUs) and refer to a specific device using an object of type `Device` that can be built providing a device type and a number of device. For example, `Device(ACCELERATOR, 2)` would be the third accelerator in the system, as the numbering is zero-based. An object `d` of this type can be used to specify where to run a kernel using the syntax `eval(f).device(d)`, that is followed optionally by other optional execution modifiers, and finally, the kernel arguments. The method `getProperties` of this class fills a structure of type `DeviceProperties` that has a field for each property of the associated device, thus allowing their inspection.

Advanced coherency and synchronization scheme: When an HPL kernel execution is requested, the host copies to the memory of the selected device the kernel inputs that were not available in it, then launches the kernel, and it continues executing the main program without waiting for the kernel to finish. This allows parallelizing computations in the host and the devices as well as requesting parallel executions of kernels in different devices.

The synchronization mechanism that allowed to wait for a kernel execution to finish and retrieve its results in Chapter 2 was only based on the host accesses to the `Array`s used as arguments to the kernel executions. This way, when the host code tried to read an array that was written by a previously launched kernel, the HPL runtime waited for the kernel to finish and copied the resulting array to the host memory, after which the execution of the main thread in the host would be allowed to continue. Subsequent host accesses to the array would be immediately satisfied from the host-side copy until new kernel executions that wrote to the array were requested. Similarly, an array used as input in a kernel execution would be copied to the device only in the first usage of the array in the device or if the host had written to the array in its memory after the most recent usage of the array in the device. These mechanisms sufficed for efficient single-device executions as the evaluation in the previous chapter shows.

However, in order to successfully exploit with a reasonable performance and consistent semantics several accelerators, HPL had to be extended in several ways. First, since the user can request to use the same array in multiple devices, and they do not share memory, the HPL runtime was improved to support multiple simultaneous copies of the same array, one per device where it is used, in addition to the host-side copy. The copies of each `Array` are hidden from the user, who only sees its current logical value. The underlying copies are managed following a multiple-readers/single-writer policy (MRSW) policy [95] with an invalidation protocol on writes [70] in order to keep a single coherent image. Let us notice that a general implementation of the data replications implied by the MRSW strategy requires the copy of data between devices, which are automatically performed by our runtime. Finally, since the host code considers in its turn each one of the kernel execution requests as well as the host accesses to the arrays, the main thread of the application provides sequential consistency [66] to all these accesses to the `Array`s, which is the simplest and most convenient model to reason about parallel programs.

Since the `Array` is the unit of consistency, the usage of the same `Array` in several kernel executions serializes them, even if each kernel operates on disjoint parts of its data, unless of course if the `Array` is only a read-only input to all these kernels. This way, in order to successfully parallelize executions of kernels that update different portions of the same array in several devices, a different HPL `Array`, associated to the specific portion updated in the device, must be defined for each device. This policy also makes sense for read-only arrays when each device only needs to read a portion of the array. The reason in this case is not to avoid the serialization of the tasks, but to minimize the data transferred, as the `Array` is also the unit of allocation and transfer. The construction of HPL `Array`s associated to different portions of the same host array is facilitated by the fact that their constructor supports an optional argument to specify the location in host memory of the data managed by the `Array`, as commented in Section 2.2. This way, different `Array`s can start in different positions within the same C array.

Example **3.1.1.** *Figure 3.1 illustrates the implementation of a matrix product in HPL and Figure 3.2 shows a multi-device implementation of this same matrix product where the work is splitted among the available GPUs by rows. This example uses the features that provide basic multi-device support in the HPL API and takes advantage of the extended coherency and synchronization algorithm by working with* `Array`*s associated to different parts of the original underlying matrices. For simplicity the code assumes that the number of rows* `M` *is a multiple of the number* `ndevices` *of GPUs, obtained in line 4. The underlying matrices are declared in line 1 as regular arrays. Since the whole matrix* `bx` *is used in each one of the parallel kernel executions, a single HPL* `Array b` *is declared in line 2 that contains it.*

```
void mxProduct(Array<float,2> c, Array<float,2> a, Array<float,2> b, Int P)
{
  Size_t k;
  c[idy][idx] = 0.f;
  for_(k =0, k < P, k++)
    c[idy][idx] += a[idy][k] * b[k][idx];
}

...
Array<float,2> c(M,N),a(M,P),b(P,N);
...
eval(mxProduct)(c, a, b, P);
```

Figure 3.1: Matrix product on a single device using HPL

```
 1 float cx[M][N], ax[M][P], bx[P][N];
 2 Array<float,2> **c, **a, b(P, N, bx);
 3
 4 const int ndevices = getDeviceNumber(GPU);
 5 c = new Array<float, 2> * [ndevices];
 6 a = new Array<float, 2> * [ndevices];
 7
 8 for(i = 0; i < ndevices; i++) {
 9   c[i] = new Array<float, 2>(M/ndevices, N, cx+i*(M/ndevices*N));
10   a[i] = new Array<float, 2>(M/ndevices, P, cx+i*(M/ndevices*P));
11 }
12 ...
13 for(i=0; i< ndevices; i++)
14   eval(mxProduct).device(Device(GPU,i))(*c[i], *a[i], b, P);
```

Figure 3.2: Matrix product on multiple GPUs using HPL

As explained before, the kernel executions in different devices will only take place in parallel if separate `Array`*s for* `cx` *are used in each one of them, thus an array of* `ndevices` *pointers to* `Array`*s is built in line 5. Then each* `Array` *of the appropriate size is created, associating its storage to the corresponding portion of matrix* `cx` *in line 9. The same approach is followed with respect to matrix* `ax`*, as this minimizes the amount of data transferred to each device. Finally, the kernel executions in lines 13-14 use the i-th* `Array`*s of* `c` *and* `a` *for the run in the i-th GPU. After the* *kernels are launched, the host continues executing the code after line 14. It will only stop and wait for a kernel execution to finish when either the host code tries to read the associated output* `Array c[i]` *or a kernel execution in a different device requires* `c[i]` *as input. In the latter case, the host will wait for* `c[i]` *to be computed, and then it will transfer it to the other device* ■

The totally general coherency support implemented in HPL together with its automated movement of data enables to program algorithms that require transfers between devices in a very natural way. For example, stencil codes are usually parallelized by means of ghost regions [54] that replicate a portion of an array that is updated by another processor or accelerator. These structures need to be refreshed with the most recent version of the data they replicate after each update and before the next round of computations begins. When accelerators are used, this gives place to a data exchange between them [103]. Another example are pipelines, in which data proceed through a series of tasks that transform them and handle the results to the next task in the sequence. The parallelism comes from the fact that different tasks work in parallel in different processors or devices on different sets of data.

***Example* 3.1.2.** *Figure 3.3 shows a multi-device pipeline implemented with HPL. In this code we assume that the first argument of each task is only read, and the second one is only written. The pipeline iterates while there are new inputs to read in the initial* `Array input0` *(line 2). Device* `d0` *runs task* `f0` *on this input to generate the intermediate result* `output0`*. If this is not the first iteration of the pipeline, the boolean* `f1_was_run` *is true, so in line 4 we write to a file the final result of the pipeline, contained in* `Array output1`*. When the host accesses* `output1` *in function* `write` *through its API, HPL checks this* `Array` *status, so that if there are pending writes to it (from the execution of* `f1` *in line 5 in the previous iteration), HPL waits for them, updates the host copy with the current value, and finally provides the data to the user code. When line 5 requests to run* `f1` *on device* `d1` *taking as input* `output0`*, HPL coherency system waits for the most recent execution of* `f0` *to finish in order to generate the most up-to-date value, which is then transferred to* `d1`*. Both operations are blocking for the host, so lines 5 and 6 are only executed once* `output0` *has been safely copied to a buffer in* `d1`*. Line 9 ensures that the last result generated is written. Notice that since* `f1` *only reads the copy of* `output0` *in the device* `d1`*, its execution does not delay the next execution of* `f0`*, which just writes to its local copy*

```
bool f1_was_run = false;
while(read(input0)) {
  eval(f0).device(d0)(input0, output0);
  if(f1_was_run) write(output1);
  eval(f1).device(d1)(output0, output1);
  f1_was_run = true;
}

if(f1_was_run) write(output1);
```

Figure 3.3: Data exchange to implement a pipeline between devices

of `output0` *in device* `d0`*, located thus in a separate buffer, as HPL knows there is no dependency between both tasks* ■

Copy data between arrays: Another programmability improvement has been the implementation of an intelligent assignment operator (see Section 3.1.1) that allows to easily copy data between `Array`s using the natural notation `a=b` in the host code.

3.1.1. Implementation details

HPL `Array`s were extended to support multiple simultaneous copies of the same `Array`, one per device where it is used, each copy being supported by an underlying OpenCL buffer, in addition to the host-side copy, which is located in plain host memory. HPL builds the buffer images of `Array`s on demand, so that an `Array` is only allocated in a device if it is used in a kernel execution in the device. The potential existence of copies in several devices implied the need for a coherency strategy, which is the multiple-readers/single-writer policy (MRSW) policy [95] with copy invalidations on writes [70]. Our implementation takes into account the new situation that an array in a device could become outdated not only by host-side modifications as in Chapter 2, but also by executions of kernels in other devices that wrote to the array. This required in turn a new update mechanism that implied device-to-device transfers of `Array`s, in addition to the transfers between host and device considered in the previous chapter.

A very important issue that we have not seen discussed in the bibliography, is how to transfer data between OpenCL buffers in different devices, which corresponds to the copies of `Array`s between devices. There are two possibilities to perform this transfer in OpenCL, which is our backend. One is to use the OpenCL function `clEnqueueCopyBuffer`, which performs a copy between two buffers. The other possibility is to first transfer the data from the source device to a host location, and once it has finished, transfer the data from the host to the destination buffer. Common sense suggests that the first option should be the best one, since it uses a specific runtime function defined for this purpose, which enables it to exploit better possibilities when they are available, and fall back on the second option when that is not possible. In fact, the families of OpenCL benchmarks that support multiple devices that we know of, such as the SNU NPB suite [89], use this approach to exchange data between devices. Also, the benchmarks to characterize OpenCL [96] have never compared these two possibilities as far as we know. We have found however that `clEnqueueCopyBuffer` can be in fact much slower than the two sequenced transfers possibility in some systems. Figure 3.4 shows the bandwidth observed in transfers between two devices of the same type in the S2050, K20 and Xeon Phi systems that will be described in Section 3.4 using the two copy mechanisms described, two transfers (T) and `clEnqueueCopyBuffer` (C). We can see that there is a substantial difference between both approaches, and while the Xeon Phi systematically favors `clEnqueueCopyBuffer`, the situation is the opposite in the Nvidia GPUs.

In order to cope with this variability, HPL follows an adaptive approach. The library runs tests making a few transfers using both copy mechanisms to choose the best one for each kind of device when it is installed in a system. The chosen copy strategy is stored in a configuration file that is read whenever a HPL application begins its execution. The HPL runtime then uses the best copy mechanism as a function of the device involved in the communication.

The implementation of the assignment operator `a=b` to copy data between arrays has many optimizations. Its data transfer is performed in a smart way, so that the data from `b` is copied to the image of `a` that holds its most recent version (no matter it is in a device or in the host), as it is expected that subsequent uses of `a` will take place in the same place. If there are multiple updated copies of `a`, the host copy is

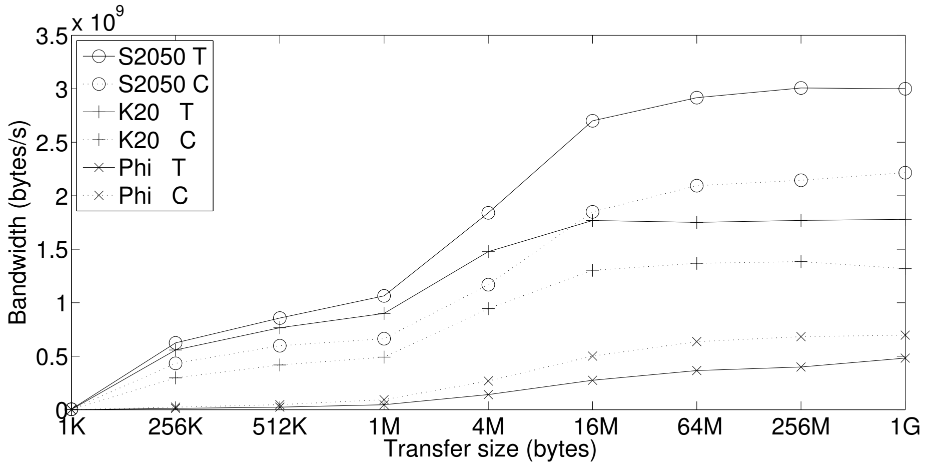


Figure 3.4: Bandwidth of copies between devices. T stands for two transfers in sequence and C for clEnqueueCopyBuffer usage

updated, and it is later transferred to the devices under demand when needed. Also, the copy is automatically performed by means of a kernel when the source and the destination are in the same device.

The synchronization mechanism was also updated, as if a kernel execution B in a device requires an array written by another kernel execution A in another device, B must be delayed until A finishes to gather the correct results. This is in contrast with Chapter 2, which never had to delay a kernel execution, as they were all run sequentially in the only device available.

Finally, we must stress that the complexity of the extended environment is totally hidden by our runtime, so that users are not concerned by the existence of the multiple copies and they do not even need to specify when to perform any transfers or updates, all the analysis of dependencies and other details being automatically managed by HPL. This way programmers are just given the simple and intuitive semantics that an `Array` data are (sequentially) consistent across all their usages in the host and the multiple devices available.

3.2. Improving multi-device support

While the programming style explained in the previous section is much better than that of alternatives such as OpenCL, it had some limitations. This section explains how these limitations were removed. In addition to the limitation related with the larger programmability cost, another important one is that since the `Array` is the unit of coherency, using the same `Array` in several kernel executions serializes them, even if each kernel operates on disjoint parts of its data, unless it is a read-only input to all these kernels. This way, the parallel execution of kernels that update different portions of an array in several devices requires defining a different `Array` per device, associated to the specific portion updated in that device. This policy also makes sense for read-only arrays when each device only needs to read a portion of them. The objective in this case is to minimize the data transferred, as the `Array` is also the unit of allocation and transfer.

In this section we propose three features to remove the aforementioned limitations of multi-device support, which are the usage of subarrays and two approaches to easily split kernels on multiple devices. The last one of these mechanisms also incorporates facilities to optimally use resources with different computing properties that participate in the computation of a kernel. The three proposals are now described in turn, followed by the description of unified memory exploitation, a general optimization that has been recently incorporated in HPL whose usage requires no effort from the user. This incorporation is an example of the steadily evolution of HPL in order to provide high programmability without sacrificing the performance.

3.2.1. Subarrays

Making the `Array` the unit of allocation, transfer and consistency forces to build separate `Arrays` to enable parallel executions and to reduce the amount of data allocated and copied. Allowing the independent use of regions of an `Array`, while providing a coherent view of it, is much more convenient. We thus enabled the selection of regions of `Arrays` using `Ranges`, `Range`(a, b) corresponding to the inclusive range of integers $[a, b]$. HPL subarrays are not copies by value, but references to the data of the underlying `Array`, so that updating them changes the corresponding

```
1 float cm[M][N], am[M][P], bm[P][N];
2 Array<float,2> c(M, N, cm), a(M, P, am), b(P, N, bm);
3
4 const int ndevices = getDeviceNumber(GPU);
5
6 for(i=0; i< ndevices; i++) {
7   Range rows(i * (M/ndevices), (i+1) * (M/ndevices) − 1);
8   eval(mxProduct).device(GPU, i)(c(rows, Range(0, N−1)), a(rows, Range(0, P−1)), b, P);
9 }
```

Figure 3.5: Matrix product on multiple GPUs using subarrays

portion of their parent `Array`. Using this feature, the matrix product example using multiple GPUs shown in Figure 3.2 can be written as Figure 3.5 shows. The main `Arrays` are defined using as storage the original C-style arrays defined in line 1, following an approach similar to that of Figure 3.2, but they could have been defined on their own, as in Figure 3.1. `Ranges` are objects of their own, so for example line 7 builds the `Range rows` that identifies the range of `M/ndevices` consecutive rows of the matrices `c` and `a` that are used in the `i`-th device. Line 8 uses this range together with appropriate ranges that span the `N` columns of `c` and the `P` columns of `a` to select the subarrays used in the execution in the `i`-th GPU.

Supporting subarrays, with the need to keep them consistent with their parent arrays, required deeply redesigning the HPL internal coherency mechanisms. The runtime keeps track of the relations between the different `Arrays` that need to be kept consistent, enforcing waits for pending writes and performing the required transfers as necessary in a process that is totally transparent to the user. The management is highly optimized in several ways, as HPL caches the subarrays structures in order to avoid costly creation processes in each reuse, performs the minimum possible number of transfers, and whenever a subarray is used in a memory that already holds an `Array` that contains the new subarray, this new subarray does not allocate new memory, but just maps in the memory of the existing `Array`.

It is important to mention that subarrays can be also used to copy portions of an `Array` using the assignment operator. Assignments are allowed between `Arrays` of the same size and from scalars to arbitrary `Arrays`, which replicates the scalar in all the elements of the destination. Finally, subarrays also facilitate the implementation of algorithms in single device environments when they need to work on different portions of arrays in different kernel invocations (e.g. matrix factorizations).

3.2.2. Subkernels based on annotations

Launching kernels in multiple devices can be simplified in several ways. For example, the selection of device and the portion of each array to process in each kernel can be hidden to the user by means of the mechanism explained below. We propose here a simple mechanism to split a kernel in multiple executions, called subkernels, ensuring that each subkernel uses the appropriate arguments. Our approach consists in annotating the arguments to specify how they must be partitioned among the devices. For example the notation `PART1(a)` indicates that `Array a` must be evenly distributed among the devices involved in the kernel execution by its first (most significant) dimension, so that each chunk is a consecutive region of data. Non annotated arrays are not partitioned, but replicated in each device.

The annotations support the syntax ANNOTATION(a,n) where a is the `Array` to split and n is the number of elements of overlapping in each direction in the selected dimension. If the user needs to specify a different number of overlapped elements per dimension, she can specify them separately with ANNOTATION(a,x,y). Namely, x is the number of elements of the current chunk of data overlapped with the previous chunk (lower memory positions) and y defines the overlapped region with the following chunk (upper memory positions). This extended syntax of the annotations interface allows the user to define stencil applications concisely and reducing the possibility of incurring in common errors in this kind of codes.

When the global domain is not specified in an `eval` of this kind, the consistent behavior of adopting as global domain of each subkernel the one associated to its first argument is followed. Similarly, unspecified local domains are chosen by HPL. The modifiers that specify these domains have been extended allowing that any or all their arguments are vectors of integers, so that the i-th component indicates the domain for the i-th subkernel. Regarding the devices to use, when no device is specified in an `eval` of this kind, HPL distributes the subkernels among the accelerators in the system. Also, the `device` modifier supports as argument a vector of HPL device handles whose length indicates the number of devices to use, and the i-th subkernel runs in the i-th device.

This approach allows to write `eval(mxProduct)(PART1(c), PART1(a), b, P)`, which partitions `c` and `a` by rows and replicates `b` in each device, as the most compact representation of our matrix product parallelized on the available accelerators described in Figure 3.1. That is, this proposal allows to distribute the computation expressed in Figure 3.1 among several devices just by simply applying the notation `PART1` to the arguments `c` and `a` in line 12.

3.2.3. Subkernels based on execution plans

While the proposal in Section 3.2.2 is very convenient, it has some restrictions. The most important one is that when several argument arrays are partitioned, it necessarily uses the n-th subarray generated of each one of them in the n-th subkernel. Although this perfectly fits most kernels, some require more flexible patterns. Also, some kernels may require a more ad-hoc partitioning. This led us to design a more complex subkernel generation strategy. This proposal is based on an object, which we call execution plan (`ExecutionPlan` class), that encapsulates the information needed to partition the work expressed by the global domain of a kernel. This information is the list of devices to use, the percentage of the work that each device will perform, and how to partition the arguments for each subkernel responsible for a portion of the computation. This last item is expressed by means of a user-provided function, which we call partitioner, that given a range of work and the kernel arguments, performs the subkernel invocation partitioning accordingly the arguments.

When an `eval` is provided with an execution plan, it uses this object to partition the global domain provided for the kernel in chunks that are proportional to the amount of work to be performed in each device. Our current implementation partitions the most significant dimension of the global domain of the kernel in chunks that are proportional to the ratios requested for the user. The reason for focusing the partitioning on this dimension is that the less significant dimensions are usually the ones with more locality and where coalesced accesses (in devices such as GPUs) are exploited. The resulting global domain partitions are also automatically rounded to ensure that they are multiples of the local domain if the user has specified one. Then, it invokes the partitioner with a functor object called `FRunner` that encapsulates the information needed to run a subkernel, an array of ranges that provide the portion of the global domain assigned to that subkernel in each dimension, and the kernel arguments. The partitioner must invoke the `FRunner` as a function whose arguments are the subarrays of the global argument arrays that are associated to the range(s) of the global domain computed by the execution plan.

Figure 3.6 implements our matrix product example using this approach. The `ExecutionPlan` is built in line 7 providing the partitioner defined in lines 1-5. A partitioner always has as first argument the `FRunner` that runs the kernel.The second argument is always an array of up to three ranges that describes the portion of the global domain associated to a subkernel, each `Range` being associated to one dimension of the problem. These two arguments are followed by the actual list of arguments of the kernel. While its body can contain more things, a partitioner only needs to invoke the `FRunner` with the arguments associated to the portion of the global domain associated to the input ranges. Lines 8 and 9 add GPUs 0 and 1 to the `ExecutionPlan ep`, respectively, each one being assigned 50% of the work. Finally, the parallel multi-device kernel execution using `ep` is requested in line 10. As we can see, this strategy allows to easily and flexibly split the work between devices assigning arbitrary portions of work to each one of them while supporting any custom required subarrays.

Automatic load balancing

Execution plans can be either completely specified by the user, as seen above, or programmed to search for a distribution of work that balances the load among the devices that participate in the kernel execution. This second possibility is very appealing when heterogeneous devices with different communication latencies and bandwidths and/or computational capabilities can be used to execute a portion of the considered kernel. In order to use an execution plan in this second way the user has to specify in its constructor the search algorithm to use and whether only the computation or also the transfer times of the kernel must be taken into account in the balancing. Also, there is no need to provide the ratios of work for each device that is added to the plan, as they will be automatically derived by our library.

```
void partitioner(FRunner& fr, Range rg[2],
                 Array<float,2>& c, Array<float,2>& a,
                 Array<float,2>& b, int P) {
  fr( c(rg[0], rg[1]), a(rg[0], Range(0, P−1)), b(Range(0, P−1), rg[1]), P );
}

ExecutionPlan ep(partitioner);
ep.add(GPU, 0, 50);
ep.add(GPU, 1, 50);
eval(mxProduct).executionPlan(ep)(c, a, b, P);
```

Figure 3.6: Matrix product on multiple GPUs using an execution plan

The search will be performed the first time that the execution plan is used in the execution of the kernel. Subsequent usages of the object will reuse the distribution found unless the user resets it.

Currently HPL execution plans provide three algorithms to search for the best work distribution. The simplest and most expensive one is the `EXHAUSTIVE` search, which tries all the legal combinations of distributions that differ in a given minimum step and chooses the best one. The default step variation is 5% of the global domain, but users can choose a different one. Just as in the other search algorithms, the library will only time the kernel execution or it will also perform and time the transfers for the inputs and the outputs to choose the best option depending on what the user specified in the construction of the execution plan. Notice that the two possibilities make sense in different scenarios, as sometimes the user may know that the data will have to be transferred for each execution of the kernel, while in other situations, such as iterative algorithms, the vast majority of the kernel executions do not require transfers. It also deserves to be mentioned that in order to avoid noise measurement problems, each distribution considered both in this scheme and in the ones described below is timed a number of times that the user can configure, the default being twice.

The other two possibilities are based on profiling and a simple model that relates the portion of the global domain assigned to a device with its runtime. Both models start with an execution in which each one of the $NDevices$ devices involved is assigned $(100/NDevices)\%$ of the global domain, and the times gathered are stored

```
1  Algorithm balance($\overrightarrow{t}$)
       input  : Vector of times measured in each device $\overrightarrow{t} = t_i,\ 0 \leq i < NDevices$
       output: Vector of percentage of work to be performed in each device
               $\overrightarrow{w} = w_i,\ 0 \leq i < NDevices$
2      $MaxTime = \max\{t_i,\ 0 \leq i < NDevices\}$
3      $u_i = MaxTime/t_i,\ 0 \leq i < NDevices$
4      $U = \sum_{i=0}^{Ndevices-1} u_i$
5      $w_i = u_i \times 100/U, 0 \leq i < NDevices$
6      $Adjust = 0$
7      for $i = 0$ to $NDevices - 1$ do
8          if $\frac{t_i \times w_i}{100/Ndevices} < Threshold$ then
9              $Adjust = Adjust + w_i$
10             $w_i = 0$
11         end
12     end
13     $w_i = w_i \times 100/(100 - Adjust), 0 \leq i < NDevices$
14     return $\overrightarrow{w}$
```

Figure 3.7: SINGLE_STEP_MODEL load balancing algorithm

in a vector of times $\overrightarrow{t}$. Figure 3.7 shows the load balancing algorithm of the first model, called SINGLE_STEP_MODEL because it is not iterative. Starting from the measured times $\overrightarrow{t}$, this algorithm assigns to each device a ratio of work that is proportional to the speedup it achieved with respect to the slowest device in the set in this initial execution (line 3). The rationale is that this speedup is proportional to the amount of work that the corresponding device can receive in a balanced distribution of work, as this number should be proportional to its computing (and transfer, if included) performance. Lines 4 and 5 scale this initial ratio to a percentage making sure the addition for all the devices covers the whole global domain. Lines 6-12 compute whether with this distribution the estimated runtime for some device falls below a measured threshold. If this is the case, the device receives no work (line 10), and its portion of the problem is added to *Adjust* (line 9), so that it is redistributed among the remaining devices in line 13. This last stage of the algorithm avoids sending very small amounts of work to devices that do not compensate the reduced fixed overhead associated to this process. Notice that since this model requires a single execution, and it is made at runtime on correct inputs, thus generating correct outputs, this model can make just as many kernels executions as an untuned code.

The second model is iterative, so it can adjust its distribution with more precision in situations where the workload is not necessarily directly proportional to the ratio of the global domain assigned to each device. This model, called `ITERATIVE_MODEL`, starts applying the `SINGLE_STEP_MODEL` and measures the runtimes of the distribution it chooses. Then, as long as the longest runtime for a device is $\delta\%$ or more longer than the shortest time, it recomputes the percentage of the global domain assigned to each device using the algorithm in Figure 3.8 and makes a new execution to measure the new times. The value of δ defaults to 5, but it can be chosen by the user. The algorithm estimates which would have been the runtime for each device in an execution in which all of them received the same amount of work based on the actual distribution of work made and the time measured, and then applies the `SINGLE_STEP_MODEL` to these times.

A problem that appears when users let the library choose the work distribution is that they lose the information on which portion of each array has been loaded and updated in each device. This never endangers the correctness of the HPL applications because whenever the host code or a kernel invocation tries to use an `Array` or a portion of it, our library knows where is the most up-to-date version of this data. Nevertheless performance will suffer if these automatic mechanisms introduce unnecessary transfers. HPL provides two mechanisms to avoid these problems. First, an `ExecutionPlan` can be built as a copy of an existing one so that it retains the same devices and work distribution, only changing the partitioner. Second, these objects provide methods to retrieve the distribution performed so that the user can know which portion of each array was used in each device and act accordingly.

3.2.4. Unified memory exploitation

Some devices have a memory that is separated from that of the host, while others work on the same physical memory. This is the case of the CPU when it is used under OpenCL, or the integrated GPUs that ship with many current CPU models. When the memory of a device is unified with that of the host transfers between both memories can be avoided or optimized. The exploitation of this property in OpenCL requires programmers to follow a series of steps that are different from

```
1 Algorithm adaptiveBalance($\overrightarrow{v}$, $\overrightarrow{t}$)
      input  : Vector of times measured in each device $\overrightarrow{t} = t_i$, $0 \leq i < NDevices$
      input  : Vector of percentage of work that was performed in each device
               $\overrightarrow{v} = v_i$, $0 \leq i < NDevices$
      output : Adapted vector of percentage of work to be performed in each device
               $\overrightarrow{w} = w_i$, $0 \leq i < NDevices$
2     $nt_i = \frac{t_i}{v_i} \times \frac{100}{NDevices}$, $0 \leq i < NDevices$
3     $\overrightarrow{w}$ = balance($\overrightarrow{nt}$)
4     return $\overrightarrow{w}$
```

Figure 3.8: `ITERATIVE_MODEL` adaptive load balancing algorithm

the usual ones, an example being working based on mapping and unmapping of OpenCL buffers instead of usual read and write operations. This way, performing this optimization further adds to the complexity of OpenCL programs.

HPL automatically detects when the device used for a kernel execution has its memory unified with that of the host, and internally applies the suitable OpenCL protocol to minimize the communication cost between this device and the host, making the optimization totally transparent and effortless to users.

3.3. Improving stencil applications

Stencil computations can be commonly found in many scientific applications with computations that follow a fixed pattern. In this pattern, the value of an element of the output array depends on the values of elements of the input array that belong to a given neighborhood. In single-device environments this property does not cause any trouble because all the threads can access the whole device memory. Nevertheless, in multi-device systems, where both data and the computations are divided among several devices, stencil computations require that each device also accesses regions of data that are updated by other device. Thus, these replicated regions of data, which have to be coherent with the original data located in other device, are called ghost regions. The management of these ghost regions is the main challenge of stencil computations in multi-device systems. Because of their importance, HPL has been extended to tackle them in a convenient way.

Figure 3.9 illustrates a minimal HPL code using stencil computations. In each iteration, this code computes `input[idx] = input[idx]+input[idx-1]+input[idx+1]` by means of two kernels: `simpleKernel`, which computes the accumulation in the temporary array `accum` and `update`, which accumulates it into `input`. This code verifies the condition of stencil because it is a 1-D problem where the value of `input[idx]` for the next iteration depends on the value of `input[idx]` and its neighborhood at the beginning of the iteration. This is a single-device version, where only one device does all the work and therefore there are no ghost regions. Multi-device executions of stencil applications require a non trivial mechanism of memory management. To understand the details of this mechanism, a multi-device version of the example of Figure 3.9 is presented in Figure 3.10. The goal of this example is to divide the workload of each kernel among several devices, so that in each iteration, each device performs its corresponding part of the work.

HPL allows the definition of overlapped subarrays, that is, subarrays with shared parts of their memory spaces. When the subarrays are updated in different devices, their shared parts are called ghost regions and they must be synchronized to guarantee that the most up-to-date values are available in each device when they are needed. In our example, this involves manually updating the ghost regions of `input` in each iteration as in Figure 3.10. Function `obtain_subarrays_size()` (Lines [3-23]) illustrates the work required to compute the limits of the subarrays needed in any multi-device problem of one dimension. The bounds of each subarray are represented by the variables `r_begin` and `r_end` in the code. These bounds are computed taking into account the size of the problem and the number of devices involved. Two data structures, `s_s` and `s_e`, are filled with starting and ending points for each subarray. In case of having two devices in the system, lines 33 and 34 launch half of the work load on two GPUs in parallel. Lines 36 and 37 launch the kernel that updates the input array with the data of the array `accum`. After this, the ghost regions of input array are still outdated. The update of these ghost regions is performed in lines [39-49]. To cope with this update, HPL supports subarray assignments and nested subarrays, this is, subarrays defined inside another subarray. For example, in line 40 the 2nd element of subarray `input(Range(N/2-1, N-1))` is copied in the last element of subarray `input(Range(0,N/2))`. This is the update of the ghost region of

```
void simpleKernel(Array<float,1> input, Array<float,1> accum)
{
  accum[idx] = input[idx]+input[idx−1]+input[idx+1];
}
void update(Array<float,1> accum, Array<float,1> array)
{
  array[idx] += accum[idx];
}
...
Array<float,1> input(N), accum(N);


for(i = 0; i < 10; i++)
{
  eval(simpleKernel)(input, accum);
  eval(update)(accum, input);
}
```

Figure 3.9: HPL stencil example in single-device

the GPU 0. In following line, the update of the ghost region of GPU 1 takes place. Once the ghost regions have been updated, a new iteration can start. Notice that this update process becomes even more complex and error-prone as the number of dimensions of the problem involved increases.

```

int obtain_subarrays_sizes(std::vector<int>& s_s, std::vector<int>& s_e, const int ndevices)
{
  float work;
  int accum_work, last, r_begin, r_end, current_work;
  accum_work = 0;
  last = ndevices−1;
  for(int j = 0; j < ndevices; j++)
  {
    work = 1.0f / ndevices;
    if(j == last)
      current_work = N − accum_work;
    else
      current_work = N ∗ work;

    r_begin = (accum_work−(j!=0));
    r_end = (accum_work + (current_work+(j!=(last)))) −1;
    s_s.push_back(r_begin);
    s_e.push_back(r_end);
    accum_work += current_work;
  }
}
...
Array<float,1> input(N), accum(N);
vector<int> s_s;        // Subarray start points
vector<int> s_e;        // Subarray end points
const int ndevices = getDeviceNumber(GPU);
obtain_subarrays_sizes(s_s, s_e, ndevices);
...
for(i = 0; i < 10; i++)
{
  for(int j = 0; j < ndevices; j++)
    eval(simpleKernel).device(GPU, j)(input(Range(s_s[j],s_e[j])), accum(Range(s_s[j],s_e[j])));

  for(int j = 0; j < ndevices; j++)
    eval(update).device(GPU, j)(accum(Range(s_s[j],s_e[j])), input(Range(s_s[j],s_e[j])));

  if(ndevices > 1)
  {
    input(Range(s_s[0],s_e[0]))(Range(s_e[0],s_e[0])) = input(Range(s_s[1],s_e[1]))(Range(1,1));
    input(Range(s_s[1],s_e[1]))(Range(0,0)) = input(Range(s_s[0],s_e[0]))(Range(s_e[0]−1,s_e[0]−1));

    for(int j = 2; j < ndevices; j++)
    {
      input(Range(s_s[j],s_e[j]))(Range(0,0)) = input(Range(s_s[j−1],s_e[j−1]))(Range(s_e[j−1]−s_s[j−1]−1,s_e
            [j−1]−s_s[j−1]−1));
      input(Range(s_s[j−1],s_e[j−1]))(Range(s_e[j−1]−s_s[j−1],s_e[j−1]−s_s[j−1])) = input(Range(s_s[j],s_e[j])
            )(Range(1,1));
    }
  }
}
```

Figure 3.10: HPL stencil example in multi-device using subarrays

3.3.1. Automatic update of the shadow regions: syncGhosts

Ghost regions always appear in the frontiers of the subarrays. Figure 3.11 illustrates the most meaningful examples of ghost regions, which are the shadowed cells, for three spaces (1-D, 2-D and 3-D). For example, in the case 3.11(a), where the vector `A` of eight elements is distributed, the device in charge of processing the elements [0-3] has one extra element that replicates A(4), which is updated in parallel by the other device. The same happens in the second device with A(3). When the parallel executions of both devices have finished, these ghost regions must be updated because they are outdated. We now explain the details of the novel automatic mechanism to update ghost regions in HPL.

A first thing to take into account is that ghost regions are naturally generated in HPL by selecting subarrays that overlap in their borders and which are used in computations in different devices. No matter the subarrays are generated by means of explicit indexing, as in the example in Figure 3.10 or by means of an automatic distribution of an `Array` using annotations, the HPL runtime keeps track of their size and relative position within the global `Array` where they are defined. In addition, as explained in Section 3.2 the HPL coherency system knows where the current and the outdated versions of every `Array` are, no matter it is a subarray of another `Array` or not. With this information it is possible to infer which are the ghost regions and which device owns each portion on them. For example, in Figure 3.11(a) since one device would work on A(Range(0,4)) and the other one on A(Range(3,7)), HPL would realize that A(Range(3,4)) is shared between both devices, A(3) being actually owned by the first device and A(4) by the second one. In this case the overlapped area has just two cells, but it could be more complex containing multiple elements for 2-D or 3-D problems. In addition, the ghost regions can have a width of more than one element, which is achieved simply by making bigger the region shared between the subarrays or by using a larger value for the overlapping extent when using annotations. The HPL runtime has been enhanced to correctly identify all these situations and determine which part of each shared region is owned by each device based on its relative position within the region. With this information, HPL can apply its automatic update algorithm, called syncGhosts, whose pseudo-code for the one-dimensional version is shown in Figure 3.12.

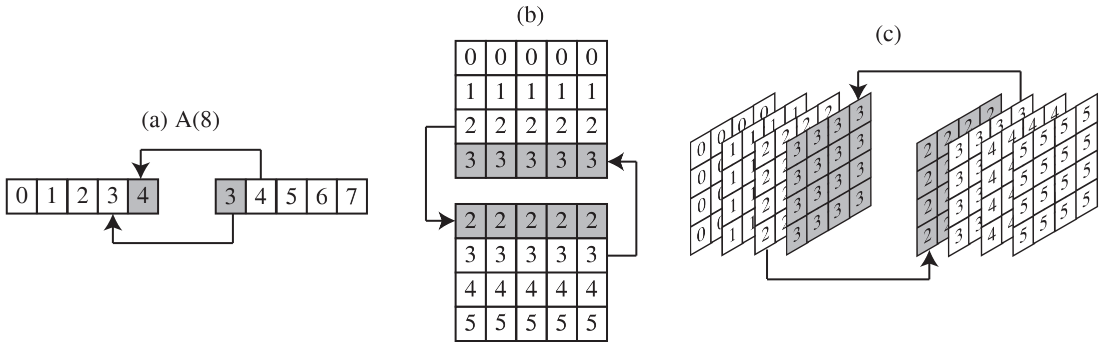


Figure 3.11: Examples of ghost regions for (a) 1-D, (b) 2-D and (c) 3-D problems in a problem divided by two devices

```
Algorithm syncGhosts(A)
    input: Array with overlapped subarrays
    foreach ghost_region in A do
        $N = size(ghost\_region)$
        $SL = ghost\_region.getLowerSubarray()$
        $LU = ghost\_region.getUpperSubarray()$
        $SL(Range(size(SL) - N)) = SU(Range(N/2, N)))$
        $SU(Range(0, N/2)) = SL(Range(size(SL) - N/2))$
    end
```

Figure 3.12: 1-D `syncGhosts` algorithm

For each ghost region of the `Array` several steps are done. First, its size is computed in line 3. It deserves to be mentioned that regions always have an even size, because the number of ghost elements that each subarray has, matches with those of the neighbor in charge of their update and vice versa. Then, the overlapped arrays are selected: SL or lower positions subarray and SU or upper positions subarray (lines 4 and 5). After this, two memory copies are done. In line 6, the data of SU needed to update the ghost region in SL are transferred. In line 7, the appropriate data of SL are copied in the ghost region of SU. HPL takes into account the location of the most up-to-date copy of these subarrays so that only the minimum number of memory copies will be done to keep updated the ghost regions. The algorithm shown in the figure does not express all the details for the different shapes of the ghost regions. The algorithm in 2-D and 3-D problems is much more complex due to the introduction of one or two additional dimensions respectively. The process is also optimized so that these memory copies do not necessarily imply a true exchange of data between two devices. They will be done only if they are strictly necessary, i.e., only ghost regions that are not up-to-date will be updated.

The *syncGhosts* algorithm is run on an `Array` when the user invokes the new method `syncGhosts()` on it. Figure 3.13 shows the code of Figure 3.10 after replacing the manual updates with a *syncGhosts* call. The `obtain_subarray_sizes()` method has the same body as in the example of Figure 3.10. It deserves to be mentioned that the benefit of the `syncGhosts()` call is not exclusively related to the lower number of lines of code, but also with the complexity of the computations of the regions of data to exchange. Using the array distributing annotations, the same example of Figure 3.13 could be rewritten as in Figure 3.14.

Note that using the coupled solution of annotations with `syncGhosts()`, the code related with the definition of subarrays, such as function `obtain_subarrays_sizes()`, is no longer needed. As we can see, the complexity related to the creation of subarrays and their exchange is highly reduced using this new approach.

3.4. Evaluation

This section evaluates the programmability and performance of the improvements to the HPL library introduced in this chapter. Since the HPL backend is OpenCL, this is the standard tool with which it is fairer to compare our library. As in the previous chapter, the C++ OpenCL API has been chosen for the baseline, as this is the language in which HPL, and thus its benchmarks, have been developed, so that both approaches enjoy the same language. The codes used as baselines in this chapter do not contain the cumbersome initialization of OpenCL (device selection, creation of context and command queue, loading and compilation of kernels, etc.), which we have encapsulated in routines that are invoked from the baselines. This way these baselines contain the minimum amount of code that users need to write using the OpenCL host C++ API.

The evaluation has been performed in three stages. First, we assess the new coherency scheme presented in Section 3.1 following a naïve multi-device implemen-

```

int obtain_subarrays_sizes(std::vector<int>& s_s, std::vector<int>& s_e, const int ndevices
      )
{...}
...
Array<float,1> input(N), accum(N);
vector<int> s_s;      // Subarray start points
vector<int> s_e;      // Subarray end points
const int ndevices = getDeviceNumber(GPU);
obtain_subarrays_sizes(s_s, s_e, ndevices);
...
for(i = 0; i < 10; i++)
{
  for(int j = 0; j < ndevices; j++)
    eval(simpleKernel).device(GPU, j)(input(Range(s_s[j],s_e[j])), accum(Range(s_s[j],s_e[j])));

  for(int j = 0; j < ndevices; j++)
    eval(update).device(GPU, j)(accum(Range(s_s[j],s_e[j])), input(Range(s_s[j],s_e[j])));

  input.syncGhosts();
}
```

Figure 3.13: Example on multiple GPUs using subarrays with `syncGhosts`

```

Array<float,1> input(N), accum(N);

for(i = 0; i < 10; i++)
{
  eval(PART1(input,1), PART1(accum,1));
  eval(PART1(accum,1), PART1(input,1));
  input.syncGhosts();
}
```

Figure 3.14: Example on multiple GPUs using annotations with `syncGhosts`

tation that uses independently and manually defined `Arrays` for each array to be used in a device. Then, the benefits of the three mechanisms presented in Section 3.2 aimed to improve the basic implementation are measured and commented. The section finishes with the evaluation of our last proposal to enhance the programmability of applications that use several heterogeneous devices, namely the `syncGhosts` mechanism.

3.4.1. Naïve multi-device support

This evaluation is based on six benchmarks described in Table 3.1 in terms of the number of source lines of code, excluding comments and empty lines (SLOCs) of their OpenCL C++ implementation, the number of kernels involved in unique (u) invocations and in repetitive (r) invocations (i.e. inside a loop, so that each kernel is invoked several times), the most common pattern of communication between subtasks when they are split among several devices and the source lines of code (SLOCs) and Halstead's programming effort [55] (PE, expressed in thousands) of the host-side implementation of the baseline.

The EP and FT benchmarks, already used in Chapter 2, come from the SNU NPB suite [89], an optimized implementation of the NAS Parallel Benchmarks in OpenCL. EP is an embarrassingly parallel application that is easy to distribute among several devices. FT is a more complex benchmark that uses three kernels for its initialization before entering an iterative process that invokes 7 kernels in each iteration, all of them parallelizable among all the devices available. This benchmark computes the Fourier Transform of a 3-D array along its three dimensions. Since the array is partitioned along one of its dimensions in order to split the work among the devices, when the Fourier Transform is to be computed along that dimension, the array has to be permuted or rotated so that the array becomes partitioned by other of its dimensions, and the originally distributed dimension fits locally in each device, enabling the local computation. This leads to an all-to-all pattern of communication between the devices.

MMRow is the matrix multiplication distributed by rows used as example in Figure 3.2. Summa implements the Summa algorithm for matrix multiplication [48], which divides the three matrices in tiles and interleaves stages of local multiplication in each device with stages of communications consisting of broadcasts across columns and across rows of tiles of the two input arrays. The efficient implementation of these broadcasts in our case does not involve copies between devices, but transfers of different portions of the input arrays from the host to each device in each step.

Benchmark	Kernels	Data exchanges	Baseline	
			SLOCS	PE (Ks)
EP	1 u	-	325	1612
FT	3 u + 7 r	all to all	1656	35219
MMRow	1 u	-	220	1082
Summa	1 r	broadcast	298	1867
ShWa	3 r	stencil	572	3430
N-body	1 r	all to all	160	1027

Table 3.1: Benchmarks characteristics.

This way this benchmark stresses the communications between the host and each device.

Benchmark ShWa, also used in Chapter 2, is a shallow water simulator with transport of contaminants developed in [71]. This application divides a surface into square volumes that interact with their neighbor volumes through their four edges, having a pattern of computation in stencil. This way, its kernels are parallelized using the well-known approach of ghost or shadow regions [54] that replicate a portion of the data in another processor. These regions need to be refreshed in each new time step as the original data is modified. Our baseline exchanges the data between devices by means of device to host, and then host to device, transfers, as they were the best method for our GPUs in Section 3.1.1. Finally, N-body is a simulation of a dynamical system of particles that presents an all-to-all communication pattern because in each time step of the algorithm each particle influences the behavior of all the other particles. Its data exchanges are implemented using `clEnqueueCopyBuffer`, as it is the natural way to make data copies in OpenCL programs.

Figure 3.15 measures the programmability improvement provided by HPL with respect to the OpenCL C++ baseline in terms of the reduction of programming effort metrics measured in the code of the host side of the application. The kernels have not been included in the measurement because their code is very similar both between OpenCL and HPL and between single-device and multi-device versions of the applications, thus the extensions described in this chapter play a small role in them. The metrics used are the same ones used in previous chapter, and along the whole Thesis: SLOCs and programming effort (PE). We can see that the effort is consistently much smaller in HPL, particularly if we take into account the relative complexity of each line of code. On average, HPL reduces the SLOCs and the

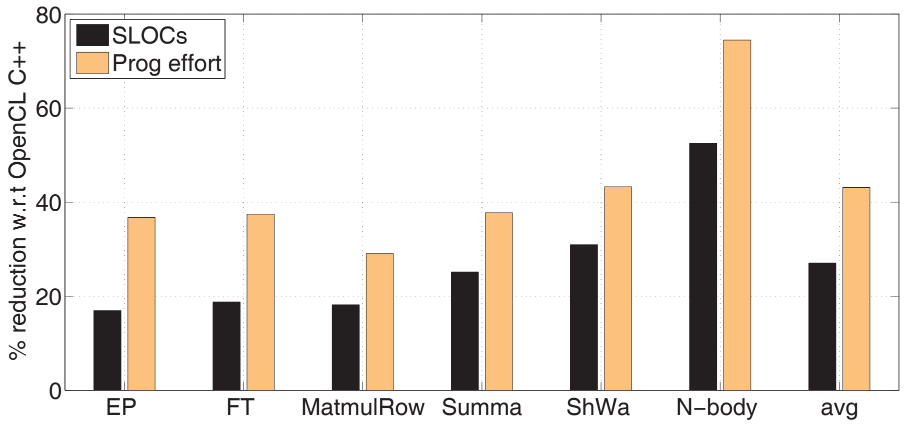


Figure 3.15: Reduction in the number of SLOCs and programming effort of the host side of the application when using HPL with respect to the OpenCL C++ baseline.

programming effort of the baseline by 27.1% and 43.1%, respectively, even when the baseline is a streamlined version with minimal code for the initialization, as we have explained.

The performance evaluation relies on three systems that are described in Table 3.2: a system with a NVIDIA Tesla Fermi S2050, another one with 3 Nvidia Tesla Kepler K20m, and one with two Intel Xeon Phi 5110P. The compiler was g++ 4.7.2 with optimization level `-O3`.

Figures 3.16 to 3.18 show the speedup of our baseline and HPL versions when using all the devices with respect to an OpenCL single-device implementation using a single device in each one of the systems just described. It deserves to be mentioned that the baselines are the same for all devices. This implies using 2 GPUs and 2 Xeon Phis in Figures 3.16 and 3.18, respectively. The SNU NPB, just as the original NPB, requires a number of devices that is a power of two, thus EP and FT only use two K20 in Figure 3.17, while the other benchmarks use three. EP and FT were run for classes D and B, respectively. The matrix products used matrices of 6000×6000 double-precision floating point elements. Finally, ShWa was run with a 1000×1000 mesh representing an actual stuary and N-body worked on 192K particles.

As we can see, HPL matches or outperforms OpenCL in most applications, sometimes experiencing some degradation introduced by its runtime. In ShWa there is an additional overhead derived from the unavailability of mechanisms in this naïve

		System S2050	System K20	System Xeon Phi
CPU	Processor	Intel Xeon X5650	2x Intel E5-2660	2 x Intel E5-2660
	Frequency(GHz)	2.67	2.20	2.20
	#cores	6 (12 HT)	8 (16 HT)	8 (16 HT)
	Memory Capacity (GB)	12	64	64
	Peak Memory Bandwidth(GB/s)	32	51.2	51.2
GPU	Processor	Nvidia S2050 (2x Nvidia M2050)	Nvidia K20m	Intel Xeon Phi 5110P
	Frequency(GHz)	1.55	0.705	1.053
	#cores	448	2496	60 (240 HT)
	Memory Capacity (GB)	3	5	8
	Peak Memory Bandwidth(GB/s)	148	208	320

Table 3.2: The Hardware Platform Details

implementation to select a portion of an array for a copy or kernel execution. For this reason, HPL ShWa must make more work to copy the rows that must be exchanged between the devices to and from separate buffers that are used for the exchanges and it is the benchmark with the largest overhead, reaching a maximum of 9% in the Xeon Phi. This overhead is removed when using portions of `Arrays` as it is explained in Section 3.4.2. In FT, however, HPL is noticeably faster than OpenCL in all the systems (up to 59% in the K20 system) for two reasons. One is that in the transfers between GPUs the HPL runtime uses the two-transfer mechanism described in Section 3.1.1, instead of the slower `clEnqueueCopyBuffer` found in the SNU NPB. The second reason is that some of the FT array copies take place between arrays that are actually located in the same device. While the SNU NPB implementation always uses the same `clEnqueueCopyBuffer` mechanism, the HPL runtime detects this situation and avoids any transfer, just making a copy inside the device by means of a kernel. The impact of these optimizations is large because FT requires many array transfers, making HPL the winner in terms of average speedup in every device for this benchmark. Something similar happens with N-body, whose data exchanges are an important part of its runtime, and are much faster under the policy applied by the adaptive HPL in the GPUs. As a result, HPL is on average 21.4%, 25.7% and 2.1% faster than the OpenCL baseline across the applications tested in the S2050, K20 and Xeon Phi systems, respectively.

Figures 3.19 to 3.21 show the speedup of HPL with respect to OpenCL for different problem sizes of FT, ShWa and N-Body, respectively, as they are the three algorithms that exchange data between devices. Since FT and N-body are based on `clEnqueueCopyBuffer`, which offers bad performance in GPUs but is the best option in the Phi, HPL clearly outperforms OpenCL in the GPUs for all the sizes.

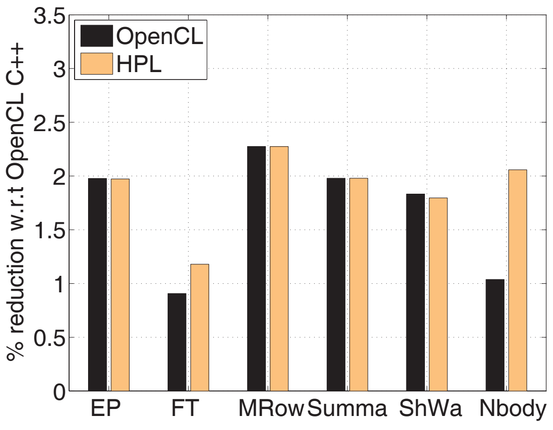


Figure 3.16: Speedups with S2050

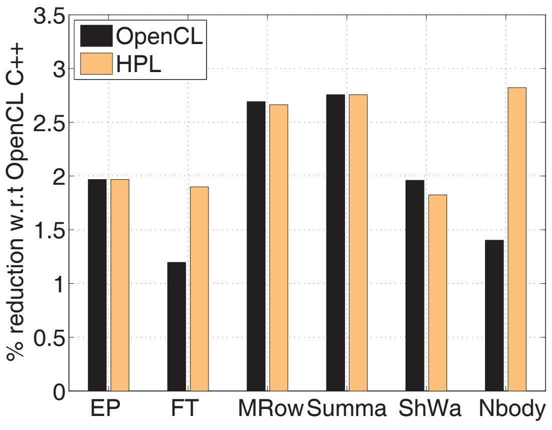


Figure 3.17: Speedups with K20

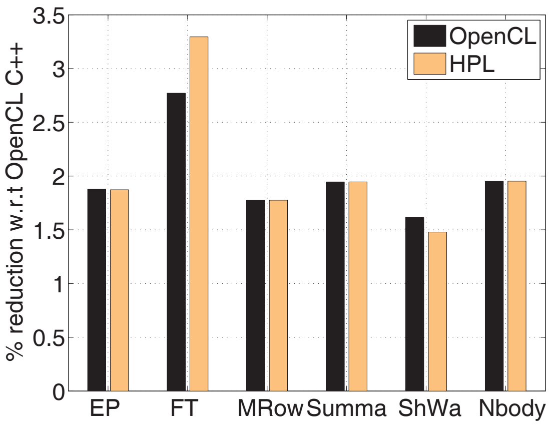


Figure 3.18: Speedups with Xeon Phi

It also outperforms OpenCL FT in the Xeon Phi because of the second advantage mentioned in the previous paragraph: HPL detects that some copies that the SNU NPB FT code always blindly performs by means of `clEnqueueCopyBuffer` have their source and destination in the same device, so HPL performs them by a faster copy inside a kernel. The N-body baseline only copies between devices the data that is strictly needed, so HPL and OpenCL get exactly the same performance on the Xeon Phi. Finally, the baseline OpenCL ShWa is optimal in the GPUs because it uses the two transfers mechanism, so HPL performs worse due to the library overheads and the additional copies that its restriction to operate on whole arrays imply in this algorithm that only exchanges one row between neighboring devices. In fact, since the amount of data exchanged is small, the adaptive nature of HPL, which allows it

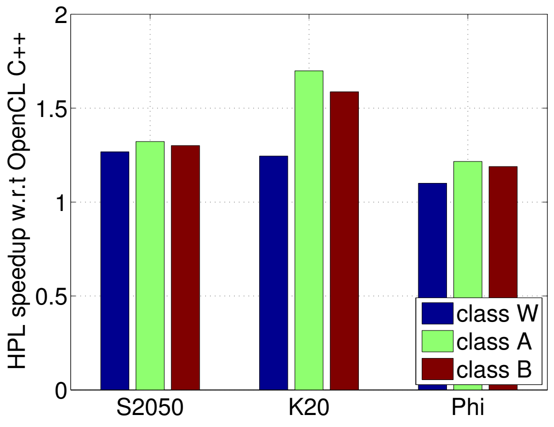


Figure 3.19: Speedup of HPL over OpenCL for different problem sizes of FT

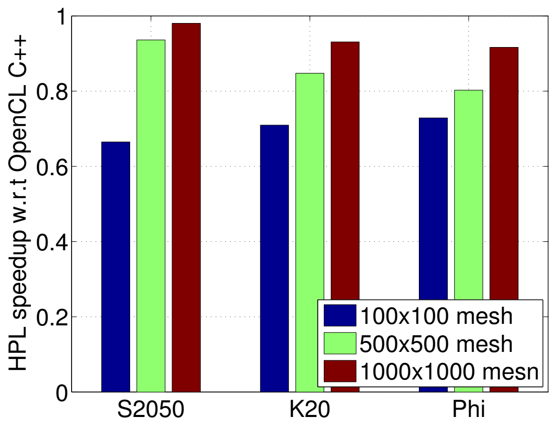


Figure 3.20: Speedup of HPL over OpenCL for different problem sizes of ShWa

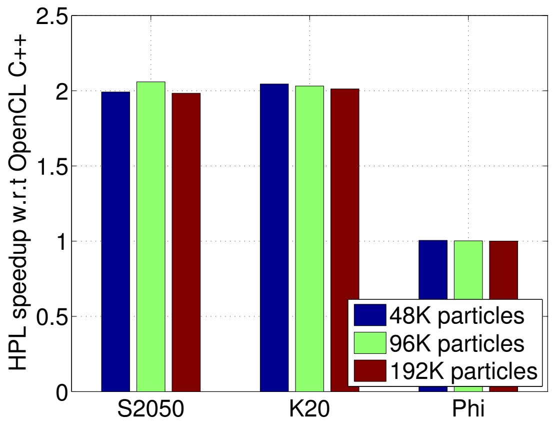


Figure 3.21: Speedup of HPL over OpenCL for different problem sizes of N-Body

to use in the Xeon Phi the faster `clEnqueueCopyBuffer` alternative (see Section 3.1.1), does not help it to reach the baseline performance in this accelerator. We see however that as the problem size grows, these overheads become an increasingly smaller portion of the runtime, thus reducing the overhead of HPL. In FT and N-body, however, HPL advantage remains basically constant across problem sizes because the whole arrays used in the problem are exchanged. The only exception is FT in the K20, where when the problem size grows from W to A we get a HPL relative speed bump, probably because W is a small problem size with many kernels and the K20 is a powerful accelerator, so the overheads of HPL do not allow it to reach its maximum advantage for a small size. Overall, HPL was 28% faster than OpenCL across this set of experiments, clearly showing its advantage in applications that exchange data between devices.

3.4.2. Improved multi-device support

In this second part of the evaluation, we assess the improvements exposed in Section 3.1.1. This way, we have replaced the N-body benchmark with the MG benchmark, which has a more complex structure with a programming effort orders of magnitude higher. Its main information is illustrated in Table 3.3. This benchmark, like FT and EP, comes from the SNU NPB suite [89], an optimized OpenCL implementation of the NAS Parallel Benchmarks.

Programmability

Table 3.4 shows the percentual reduction in SLOCs and programming effort with respect to the baseline when the applications are developed with basic multi-device HPL (md) in the same way as in Section 3.1, subarrays (sub), subkernels based on annotating the distribution of the arguments (ann) and subkernels based on execution plans (exp). These three last techniques were explained in Sections 3.2.1, 3.2.2 and 3.2.3 respectively. Notice that a higher reduction in any of the metric indicates a better programmability. Annotations can only be used when the i-th regions obtained in the partitioning of the arrays are always used together, that is, the first subkernel uses the first subarray of all the inputs, the second subkernel the second subarray, etc., which is not the case in Summa.

Programming effort reductions are always stronger than SLOC reductions because this indicator takes into account the complexity of each line, OpenCL API often having many parameters. Since we argue that this more complex metric is fairer than SLOCs, this is good news.

The techniques introduced in Section 3.2 provide better programmability than HPL-md in all the tests except the HPL-sub and HPL-exp implementations of the NPB applications. In the case of HPL-sub the reason is that the arguments of the kernels of these applications have different sizes, which does not allow to reuse

Benchmark	Kernels	Data exchanges	Baseline	
			SLOCS	PE (Ks)
MG	6 u + 37 r	stencil	3076	106666

Table 3.3: MG benchmark characteristics.

Benchmark	HPL-md		HPL-sub		HPL-ann		HPL-exp	
	ΔSLOC	ΔPE	ΔSLOC	ΔPE	ΔSLOC	ΔPE	ΔSLOC	ΔPE
EP	16.9	36.7	15.1	32.6	17.5	39.0	16.0	33.1
FT	18.8	37.4	15.7	31.3	25.9	42.1	16.7	25.9
MG	24.3	30.7	23.4	26.1	25.7	31.8	21.1	24.4
MMRow	18.2	29.0	18.6	32.3	29.1	51.9	20.5	40.2
Summa	25.2	37.7	39.9	61.8	-	-	37.3	56.4
ShWa	31.0	43.3	40.2	52.2	56.3	76.7	50.0	58.8

Table 3.4: Programmability improvement for several strategies.

`Ranges` in their indexing. Nevertheless subarrays positively impact MMRow, and largely improve upon HPL-md in Summa and ShWa. Their programmability metrics improvements over the baseline are in fact between 21% and 64% larger in relative terms that those of HPL-md for these benchmarks.

HPL-exp is the next technique in terms of easiness for the programmer, as it achieves an average 26.9% SLOCs reduction with respect to the baseline, compared to the 22.4% of HPL-md and the 25.5% of HPL-sub. Similarly, it reduces the programming effort by a noticeable 39.8%, above the 35.8% of HPL-md and the 39.4% of HPL-sub. HPL-exp main programmability advantages are that execution plans avoid loops and the computations of most of the ranges required to split the work. On the other hand, this strategy requires defining the execution plan and the related partitioner. Also, the kernels with arguments of different sizes make sometimes insufficient the predefined `Ranges` provided by the execution plans. So in these cases the user must define new Ranges, as in the HPL-sub case; this being the reason why HPL-exp does not offer better programmability than HPL-md in the NPB. The biggest asset of HPL-exp with respect to the other options is that it is the only one that allows to exploit the automatic load balancing features described in Section 3.2.3, which are evaluated in Section 3.4.2.

Finally, HPL-ann systematically improves upon HPL-md and the other alternatives presented in this work. This happens even in the simplest benchmarks, where it is more complicated, as the baseline is all the host code, including host computations and data initialization. This way, for example annotations remarkably achieve up to 60% larger SLOCs reduction over the baseline than HPL-md in MMRow. The largest improvement takes place in this application and ShWa, where HPL-ann almost doubles the programming effort reduction of HPL-md over OpenCL. On average, in the five benchmarks where it can be used, HPL-ann reduces the SLOCS and the programming effort by 30.9% and 48.3% with respect to the baseline, respectively, making it the default option when it is applicable and no automatic load balancing is needed. Overall these results largely justify the interest of our proposals.

Performance

The performance evaluation uses two of the platforms described in Table 3.2: S2050 and K20 equipped with two Nvidia Fermi GPUs and three Nvidia K20, respectively. In this evaluation we only use two GPUs of the K20 system. In future evaluations in this chapter the three GPUs will be considered to measure the impact of having more than two devices when there is communication among them. The problem sizes used in the experiments are C, B and B for EP, FT and MG, respectively. MMRow and Summa multiply double precision matrices of 8000×8000 elements, while ShWa processes a 2000×2000 mesh.

Figures 3.22 and 3.23 show the speedup of the HPL versions with respect to the OpenCL baseline in executions using 2 GPUs in each one of the systems. Notice that there is no data for Summa using annotations because this benchmark cannot be written using this strategy. The adaptive runtime, described in Section 3.1.1, allows HPL to noticeably outperform the manually optimized FT and MG multi-device implementation from [89] because HPL chooses a better strategy to exchange data between the devices. In the other benchmarks, HPL and OpenCL perform similarly. ShWa HPL-md is slightly slower than the baseline (1% in Fermi and 1.7% in K20) because the lack of subarrays complicates the refresh of the shadow region of one row in each GPU, requiring additional buffers and copies (overhead already commented in Section 3.4.1). The slowdown is reduced to 0.5% and 0.6% when HPL incorporates the novelties described in Section 3.2, allowing to use a simple assignment to a subarray for these updates. The complex shadow region

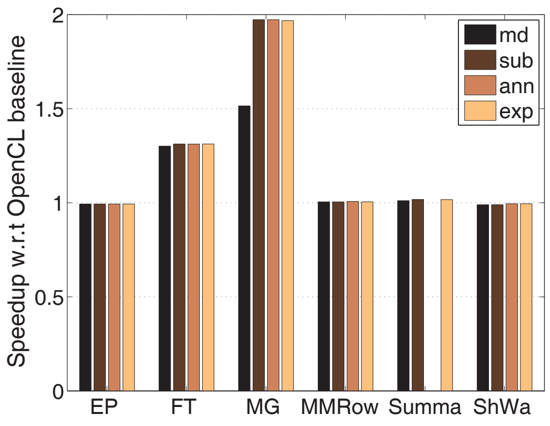


Figure 3.22: Performance in the S2050 system using both GPUs

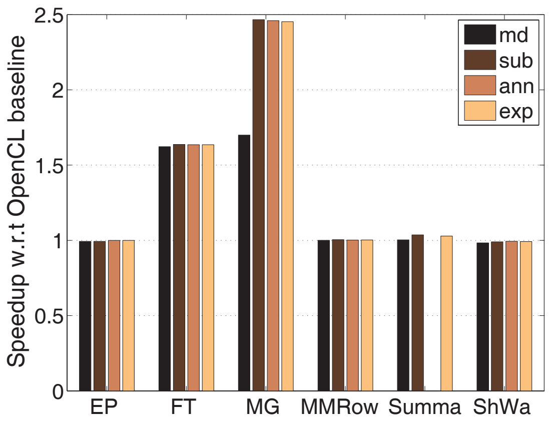


Figure 3.23: Performance in the K20 system using two GPUs

updates is also why MG md is much slower than the HPL versions enabled by the proposals evaluated in this section, which reach speedups of 98% and 146% in Fermi and K20, respectively. The reason is that the MG md does not have subarrays to efficiently perform those updates. Nevertheless, even without such feature, MG md is noticeably faster than the OpenCL baseline (51% in Fermi and 70% in K20).

All in all, the programmability improvements proposed have from a neutral to a very positive impact on performance while allowing a much more natural way of expressing the algorithms at hand.

Automatic load balancing

The main interest of the execution plan approach lies in its ability to automatically optimize the distribution of work among different devices. We have measured the performance of the automatic load balancing provided by execution plans in the most time consuming kernel of each one of our benchmarks. Summa was excluded of the experiment because it is based on the assumption that each parallel task operates on a sub-matrix of the same size. While the kernels of EP, MMRow and ShWa have a high arithmetic intensity, those of FT and MG have a high ratio of memory accesses per computation. They also have different patterns, as ShWa and MG follow a stencil pattern, MMRow operates on a tiled way on its matrices, FT computes a complex FFT (Fast Fourier Transform) using a scratch array and EP makes most of its computation in private and local memory. This way, our tests rely on kernels with very different nature.

We performed the experiments using two configurations. In the first one, HPL was asked to automatically distribute the work among the CPU and one GPU in the K20 system. In the second configuration HPL had to split the work among the CPU and two K20 GPUs. The OpenCL driver used for the CPU was the version 1.2.0.8 from Intel. Also, in both cases the two algorithms proposed in Section 3.2.3 were tried. We also sought for the optimal distribution using its exhaustive search feature using steps of 1% of the workload. Finally, in all the experiments the CPU benefited from the automatic unified memory support provided by HPL (see Section 3.2.4).

In the five benchmarks the two automatic distribution algorithms based on analytical models provided the optimum distribution identified by the exhaustive search in the CPU+single GPU scenario. Figure 3.24 shows the relative performance of the distributions found by the analytical models with respect to the best one found by exhaustive search when using the CPU in conjunction with two K20 GPUs. Both models found again the best point for the FT and MG kernels, and they were just 0.4% slower than the optimal distribution in MMRow. In ShWa they chose a distribution that is only 2% different from the optimal one (namely, it assigned to the CPU 8% of the total work, while the optimum portion was 6%), but since this is a problem extraordinarily well suited for GPUs [103], this distribution was 6.3% slower than the optimum one, which is still a very good value. Finally, in EP the `ITERATIVE_MODEL` showed that it can better pinpoint the best distribution than the `SINGLE_STEP_MODEL`, as it found the optimum distribution, while the simpler model found a distribution just 1.4% slower. Table 3.5, which shows how many times slower is the worst distribution with respect to the best one for each kernel and configuration considered, further helps to assess the quality of our balancing algorithms. We can see that the algorithms achieve optimal or near-optimal distributions in environments in which the worst distribution is between 2.76 and 51.61 times, or in percentages, between 176% and 5061%, slower than the best one.

Regarding the search cost, we used the default HPL configuration that runs each test twice to reduce the measurement noise. This way, since the `SINGLE_STEP_MODEL` requires evaluating a single distribution, while the `ITERATIVE_MODEL` always converged in two, or very seldomly, three iterations, the optimization processes based

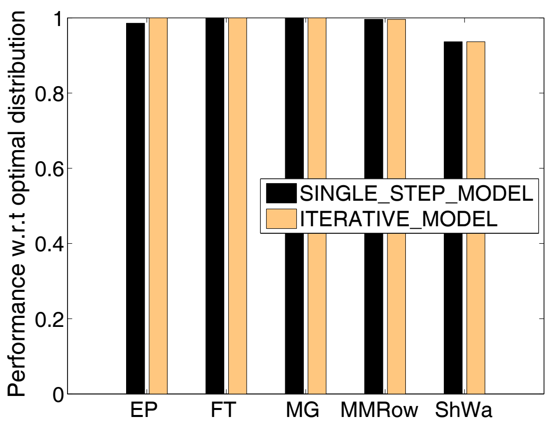


Figure 3.24: Relative performance of the distributions found by the analytical models with respect to the best one found when using one CPU and two K20 GPUs.

on analytical models only required between 2 and 6 executions. It also deserves to be mentioned that the non-first one executions of the `ITERATIVE_MODEL` start from a point near the optimal one, making them often much faster than the initial execution. This way, in practice the `ITERATIVE_MODEL` only required 48% more time than the `SINGLE_STEP_MODEL`.

Overall, we find these results to be very satisfactory, not only because of the quality of the distributions found and the very reasonable cost of our models, but also because of the simplicity of the API involved.

3.4.3. Improved stencil applications

The automatic mechanism for updating ghost regions aims to improve the programmability of the stencil computations in multi-device environments. This improvement has also the virtue of reducing the errors typically related with the memory management essentially linked to this kind of problems. It is also important to ensure that these improvements do not hurt the performance of applications. Therefore in this section both the performance and the programmability impact of our extension are evaluated.

HPL has proven to be a good alternative to low level approaches like OpenCL or CUDA in terms of programmability and performance as we can see in Chapter 2.

Environment	EP	FT	MG	MMRow	ShWa
CPU + GPU	2.76	17.73	51.61	3.98	6.68
CPU + 2 GPU	4.22	17.73	51.61	7.75	10.80

Table 3.5: Slowdown of the worst distribution with respect to the best one in the two configurations tested.

Also, its evaluation in Section 3.4.2 showed that the usage of subarrays largely improved the programmability while having a negligible impact on performance when compared to native OpenCL. Thus, the baseline for our evaluation is an HPL implementation using subarrays written following the strategy explained in Section 3.2.1. We measured only the host code because the feature presented does not affect the kernels.

For these studies, we have selected five well-known benchmarks that use stencil computations causing the occurrence of ghost regions in multi-device implementations. The main characteristics of each benchmark are shown in Table 3.6 as follows. After the name, the number of dimensions of each problem appears in the second column and the third one defines the size of the problem. The last three are the shape of the stencil used in each benchmark, the number of iterations needed in each execution and finally, the number of kernels launched in each iteration/execution. Briefly, CANNY and GAUSS are two image processing benchmarks. CANNY is an algorithm used for detecting edges in images. A simple solution consists in four steps, one kernel per step. First, a Gaussian filter is applied to smooth the image. Then, an edge detection operator is applied (e.g. Sobel). Third, a non-maximum suppression is performed as an edge thinning technique. Finally, a double threshold is applied to reduce the variety of output values. GAUSS is an image processing algorithm aimed at reducing the noise in images. Regarding their structure, the main difference between GAUSS and CANNY, is that the former one is usually performed inside a loop instead of a fixed number of steps. JACOBI3D is an algorithm that is very used in scientific computations, as it is the simplest approach to a numerical solution of the 3-D Laplace Equation via relaxation. LIFE is the Conway's Game of Life, a game that simulates the evolution of a 2-D environment. Each cell can be dead or alive and following several behavior rules each cell changes its state during the game. Finally ShWa is the shallow water simulator with pollutant transport

Benchmark	Dimensions	Problem size	Stencil shape	Iterations	Kernels
CANNY	2-D	4096×4096	5×5	1	4
GAUSS	2-D	4096×4096	11×11	1000	1
JACOBI3D	3-D	512×512×512	3×3×3	1	1
LIFE	2-D	2048×2048	3×3	5000	1
SHWA	2-D	2000×2000	3×3	100000	3

Table 3.6: Benchmark details

used in Chapter 2 and Sections 3.4.1 and 3.4.2. This iterative benchmark needs three stages/kernels to compute the evolution of a mesh of finite volumes: in the first one, the flux variation among the elements of the mesh is computed. Then, the global time step is computed for each iteration. Finally, the new flux value is computed taking into account the variations computed at the first stage.

Programmability

Table 3.7 shows the percentual reduction of SLOCs and programming effort of two HPL versions based on annotations (Section 3.2.2) with respect to versions written using subarrays (Section 3.2.1). One of the versions based on annotations performs manually the update of the ghost regions (*no-syncGhosts*), while the other one uses *syncGhosts*. Both versions reduce significantly the programming effort (PE) for all the benchmarks. The largest programmability improvement achieved by annotations without *syncGhosts* takes place in ShWa, where the SLOCS and PE are reduced by 21.92% and 34.06%, respectively. The improvements are much bigger in all the benchmarks when *syncGhosts* is used, reaching its maximum values in CANNY, which requires 69.77% and 96.67% fewer SLOCS and PE than the subarray-based HPL version, respectively. In fact, while the average PE reduction when using annotations instead of subarrays is 20.48%, this reduction grows to 79.5% (3.9 times larger) when *syncGhosts* is also used. In terms of SLOCS, the average reduction achieved is 16.96% for the first case, and 46.98% for the second one.

Performance

The performance evaluation took place in two systems described in Table 3.2: S2050 and K20. The compiler used to obtain our measurements is g++-4.7.2 with optimization level -O3. The size of the input problem chosen for each benchmark is

Benchmark	no-syncGhosts		syncGhosts	
	ΔSLOC	ΔPE	ΔSLOC	ΔPE
CANNY	17.05	18.08	69.77	96.67
GAUSS	15.15	17.04	49.24	85.90
JACOBI3D	11.86	18.04	39.83	83.38
LIFE	18.80	15.18	51.13	79.51
SHWA	21.92	34.06	24.92	52.05

Table 3.7: Programmability improvement without and with the syncGhosts mechanism of HPL programs based on annotations with respect to versions based on explicit subarrays.

reflected in the third column of Table 3.6. Regarding performance, another relevant parameter to take into account in this kind of applications is the shape of the stencil used in each case (fourth column). Finally, the fifth column includes the number of iterations of each benchmark. In summary, CANNY and GAUSS need 4096×4096 pixels image as input. CANNY uses a stencil shape's extent of 2 in each direction and GAUSS a stencil with 5 elements in each direction and 1024 iterations. A 3-D matrix of 512 elements per dimension was used in JACOBI3D. A 2048×2048 mesh and 5000 iterations is the configuration of LIFE, while SHWA performed simulations of a mesh of 2000×2000 cells. In these three last benchmarks the extent of the stencil shape was of one element in each direction.

Figures 3.25 and 3.26 show the speedup of the HPL versions based on annotations with respect to the subarray versions when two devices are used in our Fermi and K20 platforms, respectively. As we expected, the *syncGhosts* mechanism does not add any overhead to the manual mechanism. In fact, the performance differences among the three versions do not reach 1% for any benchmark in both systems.

In order to demonstrate that the good performance obtained is independent of the number of devices, the same benchmarks were reimplemented using the three devices available in our K20 system. It is remarkable that in this configuration, there is a device that exchanges ghost regions with the other two. Figure 3.27 shows the speedup of the two versions based on annotations that update the ghost regions manually (*no-syncGhosts*) and automatically (*syncGhosts*) with respect to our baseline HPL version based on subarrays in this environment. The differences of the three versions are again minimal. The maximum performance difference observed between any two versions of the same benchmark is below 2%. These results indicate that our *syncGhosts* mechanism is optimal independently of the

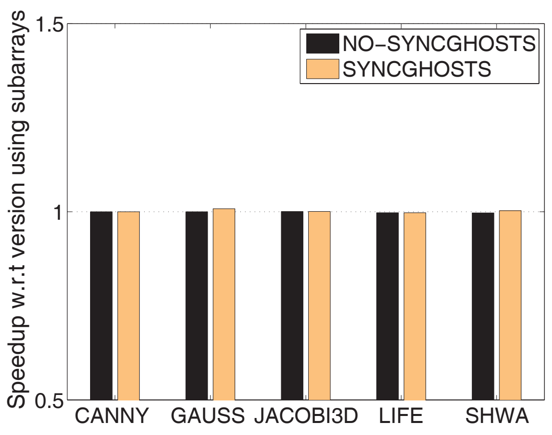


Figure 3.25: Performance in the S2050 system using both GPUs

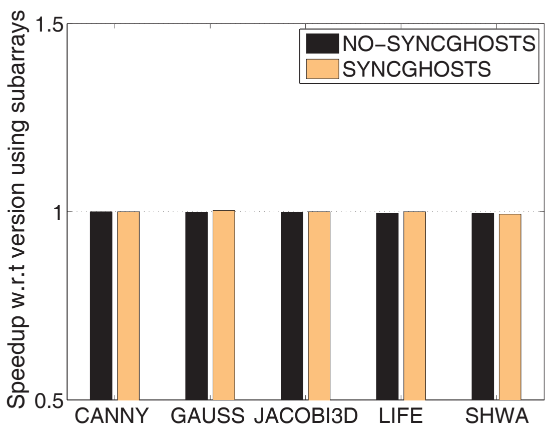


Figure 3.26: Performance in the K20 system using two GPUs

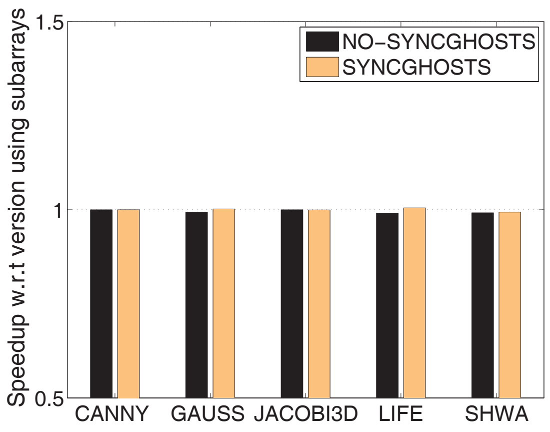


Figure 3.27: Performance improvement using 3 devices

number of devices and that it does not cause any meaningful performance penalty in comparison with manual versions.

Ghost Cell Expansion

One of the most common optimization techniques applied to applications with stencil computations in distributed memory systems is known as Ghost Cell Expansion [33]. In those systems, ghost region updates are done at the process level and they involve message passing among processes. These messages typically reduce the performance of the application. The ghost cell expansion (GCE) technique reduces their impact by decreasing their frequency. This is achieved using larger ghost regions so that more iterations can be performed without requiring an update, as the size of the updated region shrinks in each iteration. The price to pay is that each update is more expensive, as it involves a wider ghost region. For this reason the best performance is typically found with an intermediate ghost region width that balances the number of updates and their weight.

The GCE technique can be also used in multi-device systems since they are a kind of distributed memory systems. Using this technique in multi-device systems, the user performs less but heavier memory copies between devices. For example, in a 2-D problem with a ghost region of one row, each device needs to update its ghost region every iteration. With two rows per region, in the first iteration, one of the ghost rows can be updated avoiding its update in that iteration. In general, with N rows per ghost region, the ghost rows that will be read in the next iteration can be updated N-1 times without perform any memory copy between devices. The implementation of this technique in HPL using annotations is straightforward thanks to the freedom that the user has to locate the *syncGhosts* calls where it is necessary, either once per iteration or each N iterations.

The syncGhosts algorithm and the notation introduced in Section 3.2.2 largely simplify the application of GCE in heterogeneous applications involving several devices. For this reason, we tested its application to our set of benchmarks using our notation, the results for two GPUs in the K20 system being shown in Figure 3.28. Notice that CANNY is not included because it does not have an iterative nature, and thus GCE is not appropriate for it. The y axis indicates the percentage of reduction of the execution time of the *syncGhosts* version using ghost regions of different sizes with respect to the *syncGhosts* version with a ghost region of only one row. The x axis specifies the extra width of each ghost region. With ghost regions of width 2, the *syncGhosts* call is located every 2 iterations; with width 4, every 4 iterations, and so on.

GCE is beneficial for all the benchmarks but JACOBI3D. The reason is that this is a 3-D problem, and thus the increase of the width of the ghost region in one unit involves adding a whole slice (in this case of 512×512 elements) to the region. In this situation the exchange of regions becomes too expensive to compensate for the reduced update frequency. All the other benchmarks show some degree of im-

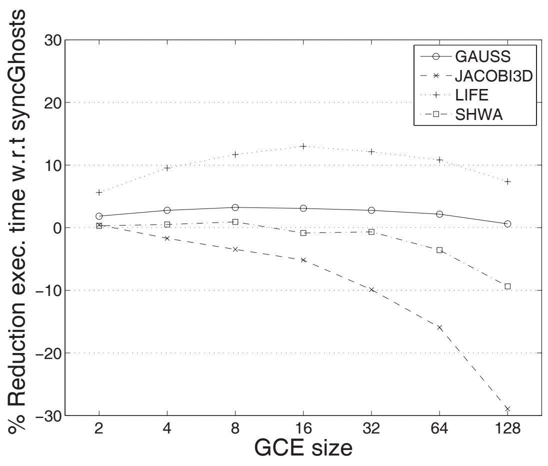


Figure 3.28: Performance of *syncGhosts* versions varying the ghost region sizes and using two GPUs in K20 system

provement for moderate amounts of GCE, ranking from a small 1% improvement of SHWA for 8 elements to a worthy 13% for LIFE with 16 elements, in both cases each element implying a row of the underlying array. The effect of GCE on the different benchmarks is quite different because it depends on many parameters such as the increase of the ghost region size in bytes it implies (not only because of the number of elements, but also because of the number of bytes required to represent each element), the computational cost associated to the larger ghost regions, whether there is a need to synchronize the devices because of other kernels used in the application, etc.

As we can see, this technique can often improve the performance of stencil multi-device codes, and our notation makes it very easy to implement it and experiment with its width. For example, using annotations it is just enough to increase the extent of the overlapped region, by changing a single number, and to reduce the frequency of invocations to the *syncGhosts* method just by adding an appropriate conditional.

Overall, the good results achieved by our solution both in terms of performance, this is, the absence of overhead, and programmability turn HPL into an excellent tool for the development of stencil codes.

3.5. Related work

Most efforts to facilitate the use of multiple heterogeneous devices mainly try to avoid communication APIs in clusters. These proposals provide a programming model in which a sequential program can allocate buffers and submit tasks to the devices that exist in a cluster. While some of these works [35, 67, 94] are based on CUDA, which restricts their portability, many [16, 40, 51, 61] rely on OpenCL. Most of these latter proposals closely follow the OpenCL API and concepts with some extensions, and thus require a much lower level management than HPL. Exceptions that abstract some details are [16, 51]. Nevertheless, the Many GPUs Package [16] involves compiler directives that must indicate the inputs and outputs of each task and specify synchronization points, or a library that in addition to these specifications explicitly uses contexts and buffers. It also includes a gather-scatter API which, unlike HPL, requires to scatter and gather the data in a single task which is the only one that can work with the chunks. Similarly, libWater [51] relies on explicit kernel creation processes, buffers associated to devices that are explicitly read and written, and synchronizations based on OpenCL-like events, supporting neither subbuffers nor automated kernel partitioning. HPL is currently restricted to the exploitation of the devices in a single node, but it offers a much higher level view. This way n-dimensional arrays rather than buffers in a given memory or device are the objects that users manipulate, being able to work even on subregions of these arrays, and leaving all the synchronizations, buffer allocations, data transfers and consistency management to the HPL runtime.

Our work is also related to the task superscalar paradigm, because HPL synchronizations and scheduling are automatically defined by the task data dependencies. Nevertheless, the existing proposals to apply this paradigm to heterogeneous computing [36, 15] require users to explicitly annotate the tasks inputs and outputs, contrary to the fully automated extraction of the dependencies of HPL. They also suffer from long boilerplate codes to use OpenCL and lack mechanisms to split a kernel in parallel subtasks with a single command. In addition [36] requires a special compiler and does not provide convenient array classes with mechanisms to define and operate on subarrays. Partitioning arrays using predefined distributions is allowed by [15], although unlike HPL, it does not allow selecting arbitrary subarrays. A task superscalar project that, like ours, automatically extracts the data dependencies of the parallel tasks is [49], but it only supports regular CPUs.

Multiple devices can also be exploited using compiler directives like OpenMP [82] and OpenACC [81]. This strategy suffers from lack of clear performance model, reduced user capability to control the result, and strong dependence on the compiler quality. These problems are even more important in accelerators, whose performance is very sensitive to implementation decisions.

Skeleton libraries [39, 75, 2, 92] are another approach to exploit multiple heterogeneous devices with reduced programming effort. HPL has a much wider scope of application than these tools, as they can only express computations whose structure conforms to one of their skeletons.

The fact that our proposal can automatically find a suitable distribution of work among multiple heterogeneous devices is another difference with the preceding works, and it relates it to the works on automatic work distribution. For example, [72] is restricted to CPUs and Nvidia GPUs, only considers a CPU and a GPU, and is based on an offline training, being thus less adaptive and general than our dynamic system, which relies on runtime information and supports any arbitrary combination of devices. A work that shares the first two limitations but uses dynamic measurements is [74]. OpenCL is the base for [53][65], but they rely on offline static models whose construction requires extensive training runs that need to be repeated when there changes in the platform. Another problem of these approaches is that their decisions are based on static code features and straightforward runtime features, thus kernels whose behavior can strongly vary depending on the contents of the input data, either as a whole or in different work-items, are not well suited for them.

More recently, [90] defines an algorithm to compute the distribution of work according to the profiling information measured through two different strategies: offline and online profiling. Following an *offline profiling*, all the profiling information is obtained before the execution of the algorithm. This way, the application is executed several times with different input sizes and the profiling results are stored. Taking into account these values, the predicted work distribution is computed during the execution of the application by means of a linear regression according to the current input size. An important drawback of this approach, as in the case of other offline static models, is that it does not take into account the parameters that can change for each execution, such as the nature of the data or the size of the local work space. In addition, the proposed algorithm requires knowing platform-dependent parameters such as several CPU and device caches sizes or the number of processing elements in the devices. These values can widely vary, and worse, cannot be obtained using a portable approach such as OpenCL, which makes impossible to apply this strategy in a portable and automated way. The *online profiling* strategy is more suitable for applications with few problem sizes, but its cost in terms of execution time is larger. An important reason is that it requires executing the whole workload to distribute in each device that can participate in the work distribution. With the profiling results from these executions, the algorithm obtains the percentages of work for each device following a process that has similar requirements, and thus portability and automation restrictions, as the offline strategy. On the contrary, HPL follows a totally portable and automatic approach and it obtains its profiling information by evenly dividing the work among all the devices involved instead of making a whole execution in each device involved, which is noticeably faster.

Interestingly, the strategies presented in [90] were also extended to tackle irregular problems in multi-device environments by sorting the input workload into a more regular shape attending to the similarity of the behavior of the data points and then distributing this more regular workload using the mechanisms described above. Other works that have improved the automatic distribution of work in irregular problems are [4, 3] which explore static load balancing strategies among the threads and blocks/work-groups when a single GPU is considered, and dynamic strategies when using multi-GPU systems.

Regarding stencil computations, some of proposals aimed to improve the multi-device support of the heterogeneous systems, like ours, are specifically targeted to stencil codes, given the difficulties for their development, particularly when multiple heterogeneous devices are used. In particular, HLSF [37] provides a high-level interface for describing stencils hiding the low level details on single-device systems. It is built on top of CUDA so that only CUDA-capable devices are supported. PARTANS [73] is focused on autotuning stencil applications than on improving their programmability, which is the main goal of HPL. PATUS [29] is an autotuner of stencil applications in the same fashion as PARTANS but while it allows the user to define stencils and strategies to parallelize and optimize them, it only supports single-device environment based on either CPUs by means of OpenMP, or CUDA-capable devices. Also SkelCL has been extended in [24] with two approaches to address stencil computations. An important limitation of SkelCL is that since all its interaction with heterogeneous devices is based on skeletons, it can only be applied to applications in which all the kernels have computational patterns that can be accommodated to one of these skeletons. In contrast, HPL is a completely general solution.

Another limitation of multi-device solutions like [24, 73] is that they do not allow separate the kernel launch from the synchronization of the ghost regions. This prevents us from delaying the synchronization and thus to take advantage of the gap between the kernel execution and the synchronization in order to let the host to do useful work in the meantime. With HPL users can freely locate useful work between the kernel execution and the swapping of the ghost regions. This is a very interesting option when GCE is applied because the kernels are even heavier because of the recomputed cells.

Finally, HPL has the unique feature with respect to all the preceding works that its kernels are written in a language embedded in C++. This allows to exploit run-time code generation under the control of the programmer and thus to dynamically adapt and optimize the kernel codes for different platforms and inputs, which can enable large performance improvements [43][42].

3.6. Conclusions

One of the biggest problems for the exploitation of heterogeneity is the associated programming complexity, which grows when several devices are in use. In this chapter we first extended HPL with an automated and optimized coherency system for arrays than can be used across multiple accelerators as well as the host of a computing node. This system is also adaptive, as it chooses the most efficient mechanism to perform the copies and it avoids transfers when it detects the source and the destination in the same device. Using a naïve multi-device implementation, the new coherency system allows HPL to reduce on average the programming cost metrics of SLOCs and programming effort of multi-device OpenCL C++ baselines by 27% and 43%, respectively. Regarding performance, its adaptive nature allows to obtain noticeable speedups with respect to hand-coded OpenCL in applications that exchange data between devices, achieving an average and a maximum speedup on a series of tests for this kind of applications using different problem sizes of 28% and 106%, respectively.

In a second step, this chapter described and evaluated several mechanisms to facilitate the exploitation of multiple devices in a node on top of HPL. The first alternative consists in allowing the use of subarrays as kernel arguments as well as source and destination of array assignments, which is required even for the implementation of some algorithms in single-device environments, although this is not explored in this chapter. The second one consists in defining portions of a kernel to run in parallel in different devices, which can be achieved using a high-level notation that automatically partitions or replicates the kernel arguments in the devices. A third contribution is an execution plan in which the user provides a partitioning function that based on regions precomputed by our library selects the appropriate subarray of each argument to be used in the kernel execution performed in each device. A very relevant part of this last approach are analytical models that automatically determine the best partitioning based on run-time profiling.

The resulting schemes are highly flexible, enjoy task superscalar execution with automatic synchronization and can reduce up to 76.7% the programming effort with respect to streamlined OpenCL baselines. The overheads of our implementation are negligible, while the absolute performance can be in fact much larger than that of OpenCL thanks to the HPL adaptive runtime, which achieves in our tests up to a 146% speedup with respect to manually developed OpenCL codes. The quality of the work distributions chosen by our execution plans is also outstanding, as they were optimal in most of the experiments, experiencing a maximum slowdown of 6.3% with respect to the best distribution found using an exhaustive search. This way we think that HPL is a very promising approach for the exploitation of heterogeneous systems and it largely benefits from the contributions described in this chapter.

The last contribution presented in this chapter is an extension of HPL aimed at improving the programmability of stencil applications in multi-device environments. The complexity of these codes increases when several devices are used because of the tasks associated to the data distribution and the synchronization of the ghost regions. Thus, the simplification and automation of these tasks becomes a very attractive option.

The new mechanism, called syncGhosts has been evaluated with five very different applications, showing that it largely reduces the programming effort without increasing the execution time. For example, while the use of HPL annotations to express stencil multi-device applications reduces on average the programming effort on 20.5% with respect to the usage of HPL subarrays, this average reduction grows up to 79.5% when the syncGhosts mechanism described in this chapter is also used. The peak reduction, which is 96.7%, is achieved in CANNY, an application with four stencil kernels. Something similar can be said about the lines of code.

The experiments also show that the performance overhead introduced by syncGhosts is negligible. Concretely using two devices, the performance differences are always below 1%. In experiments using three devices, in which more memory transfers are needed to maintain the coherence of the arrays, the results were analogous thanks to the underlying careful implementation.

In addition, we applied the ghost cell expansion (GCE) technique in HPL obtaining interesting results. Namely, with almost no programmability costs we improved the performance of a multi-device execution using ghost regions of different sizes. In particular, we achieved a 13% of reduction of the execution time in one benchmark, using ghost regions of 16 rows in comparison to the same benchmark with ghost regions of one row. In other words, we obtain a noticeable improvement only changing the frequency of the *syncGhosts* call in a loop.

Chapter 4

Heterogeneous clusters support

This chapter presents a framework for facilitating the programming of heterogeneous clusters. This framework has been designed based on the premises that simple semantics and high levels of abstraction must be provided to the user, the API must be concise and powerful, and its performance must be similar to low level approaches. The framework proposed relies on a data type that represents a distributed array on which data-parallel operations can be applied. These operations can be run either on the regular CPUs or in the heterogeneous devices of a cluster depending on the specification of the user and they are encapsulated in the data type methods, which make the associated underlying management as transparent to the user as possible. Our proposal has been developed as an extension of the existing Hierarchically Tiled Array (HTA) project [6] (`http://polaris.cs.uiuc.edu/hta`). This project provides a data type with all the required properties except the support for heterogeneous computing. We have provided the accelerator support for the HTA class by reusing the runtime and integrating the API of the Heterogeneous Programming Library (HPL) project given its portability, good performance and intuitive notation. An initial experience on heterogeneous clusters, also described in this chapter, showed that the use of both libraries in the same application separately provided good results in terms of programmability and performance, although it required some manual management operations and a duplication of handles for arrays, which complicated the programming. This led us to focus our efforts on the development of an integrated solution. The result, which we call the Heterogeneous

Hierarchically Tiled Array (H^{2}TA), is a high level proposal for the programming of heterogeneous clusters that presents important advantages with respect to the strategies that are currently used while presenting negligible overheads with respect to them.

The rest of this chapter is organized as follows. First, the HTA data type is introduced in Section 4.1. Then, our extension to support heterogeneous clusters is described in Section 4.2, followed by an experimental evaluation in Section 4.3 and a discussion on related work in Section 4.4. Finally, Section 4.5 presents our conclusions.

4.1. Hierarchically Tiled Arrays

The Hierarchically Tiled Arrays (HTAs) project is built around the HTA data type [6]. This data type represents an array that can be optionally partitioned into tiles. These tiles can be either conventional arrays or lower level HTAs. HTAs allow to use tiling both to exploit locality and parallelism, as different tiles can be processed in parallel following data parallel semantics that are embedded in the methods of the class. Also, the tiles of an HTA can be stored in a single node or they can be distributed across the nodes of a cluster. In this latter case the top level tiles are the ones that are distributed. For example, Figure 4.1 shows how to create in C++ an HTA that is divided into 2×2 tiles of 7×7 single-precision floating point elements each. The HTA is built in a distributed fashion using a cyclic distribution of its tiles on a grid of 2×2 processors that is specified by the object `dist` built in the first line. As a result each tile is placed in a different processor Pi, resulting in the mapping illustrated in the figure. The distribution object is an optional input, the cyclic distribution being the default one.

HTAs allow complexing indexings thanks to the support of two indexing operators, the parenthesis, which operate at tile level, and the brackets, that operate at element level. This way, given a bidimensional HTA `h`, `h({i,j})` selects the tile in the row `i` and column `j`, while `h[{i,j}]` refers to the element (`i`, `j`) of the underlying matrix, that is, disregarding its tiled structure. Because HTAs are a hierarchical data type, their indexing can also be applied recursively. As a result, as Figure 4.2

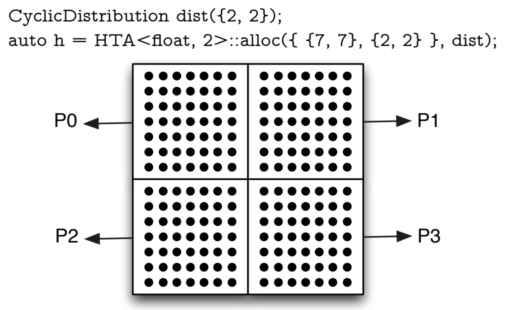

CyclicDistribution dist({2, 2});
auto h = HTA<float, 2>::alloc({ {7, 7}, {2, 2} }, dist);
P0
P1
P2
P3

Figure 4.1: HTA creation

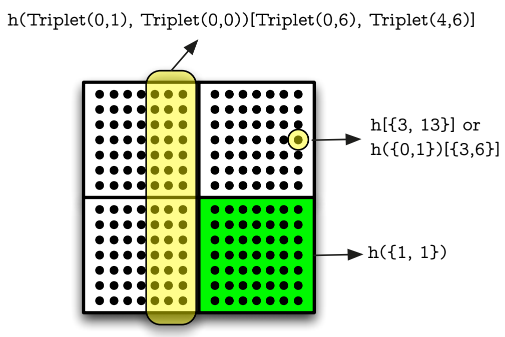
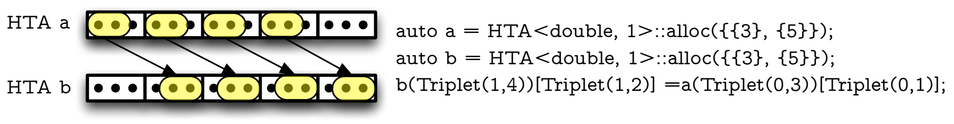


Figure 4.2: HTA indexing

shows, the element $(3, 13)$ of the HTA built in Figure 4.1 can be selected using its absolute position in the HTA using bracket-based indexing at the top level, or choosing the associated tile, and then applying the relative indexing within it. It is also possible to choose ranges of elements by using the type `Triplet`. Figure 4.2 also exemplifies the selection of the three last columns of elements of the first column of tiles in a single expression.

A final component of the HTA data type are its methods for point-wise, collective and higher-order operations among others [46], which express parallelism as different tiles can be processed in parallel. Point-wise methods are the usual array operations, including assignments, that affect individually each scalar of the structure and they use a natural notation thanks to the support of operator overloading by C++. For example, given HTAs `a`, `b` and `c`, `a=b+c` will add `b` and `c` into `a` on the condition that they are conformable [6], i.e., they have the same topology and the corresponding tiles in the topology have sizes that allow to operate them. Notice that in the case of distributed HTAs assignments imply communications if the tiles involved are located in different nodes. Also, it is interesting to notice that thanks to the flexible indexing supported by HTAs, their assignments allow elaborated movements of data such as the one illustrated in Figure 4.3 where the 2nd and 3rd elements of the tiles 1-4 of HTA `b` are replaced with the 1st and 2nd elements of tiles 0-3 of HTA `a`. Finally, HTAs can be also operated with scalars. In this case the scalar is operated with, or assigned to, each element of the HTA involved.

Collective operations are those that change the distribution of an HTA as a whole, giving place to a new HTA. For example, this is the case of permutations



```
auto a = HTA<double, 1>::alloc({{3}, {5}});
auto b = HTA<double, 1>::alloc({{3}, {5}});
b(Triplet(1,4))[Triplet(1,2)] =a(Triplet(0,3))[Triplet(0,1)];
```

Figure 4.3: HTA complex assignment example

of dimensions either at scalar or at tile level. Higher-order operations are those that take as input a function and they apply it in parallel to the HTA tiles. These operations allow to perform reductions or simply apply user-defined functions in parallel to the tiles of the HTA. For example, the function `hmap` allows to apply in parallel a user function to the tiles of an HTA. When several HTAs are provided to `hmap`, each parallel invocation operates on the corresponding tiles of the input HTAs. In this situation, if the HTAs are distributed the associated tiles should be located in the same node. Figure 4.4 shows an example of one of these `hmap` operations. This example assumes that `x`, `y` and `alpha` are distributed in the same way, function `saxpy` is applied in parallel to their tiles 0 in one node, their tiles 1 in another one, and so on. The example also illustrates that `hmap` requires that the input HTAs have the same top level structure so that their tiles can be matched, but the internal structure and size of those tiles can be different.

As we can see, in the context of a cluster HTA programmers manipulate a data type that represents a whole data structure distributed on the cluster under a given specification. All the parallelism is encapsulated in the tile-level parallel operations supported by the data type, which can apply both standard and arbitrary user-defined functions. This way, users have a single-threaded view of the execution coupled with a global view of the data. As for communications, they are conveniently expressed by means of either assignments between tiles located in different nodes or the collective operations provided by the data type. This gives place to a high level programming style that offers great programmability advantages with respect to the traditional MPI-based programming of clusters. Unfortunately, HTAs lacked until now of an integrated mechanism to exploit heterogeneity in their applications, which is the subject of this work.

```
void saxpy(HTA<float,1> y, HTA<float,1> x, HTA<float,1> alpha)
{
  int size = x.shape().size()[0];
  for(int i = 0; i < size; i++)
    y[i] = alpha[0] * x[i] + y[i];
}
...
auto x = HTA<float, 1>::alloc({ {N}, {M} });
auto y = HTA<float, 1>::alloc({ {N}, {M} });
auto alpha = HTA<float, 1>::alloc({ {1}, {M} });
...
hmap(saxpy, x, y, alpha);
```

Figure 4.4: Parallel application of a user-defined function to the tiles of HTAs

4.2. Heterogeneous Hierarchically Tiled Arrays

Motivated by the growing usage of specialized coprocessors in HPC clusters, we seek to provide users with high level approaches to program heterogeneous clusters. We propose to explore answering this problem using HTAs because of their excellent properties for parallel distributed computing.

This section presents our extension of the HTA data type to support heterogeneity, giving place to the Heterogeneous Hierarchically Tiled Arrays (H^{2}TAs). This improvement allows this data type to exploit all the of parallelism available in heterogeneous clusters. In order to tackle this extension, we took as basis the Heterogeneous Programming Library (HPL), which has proven to be a interesting alternative to low level solutions. In addition, we reused its runtime and some ideas of HPL for the notation of the H^{2}TAs. For this reason, we will first describe the use of HTAs and HPL separately in the same application for programming heterogeneous clusters. Lastly, we discuss the H^{2}TA proposal taking into account the advantages and limitations of our initial approach that will be presented in the next Section.

4.2.1. Separate use of HTA and HPL

As we can see, HTA and HPL serve very different purposes. While HTAs are well suited to express the top-level data distribution, communication and parallelism across cluster nodes, HPL largely simplifies the use of the heterogeneous computing resources available in a node. Their joint usage in one application requires solving two problems that we discuss in turn in this section.

Type integration

These frameworks require different data types to store the data they manipulate, the HTAs and the HPL `Array`s respectively. Once we are forced to handle these two types, and since the top-level distribution of data of the HTAs is made at tile level, the best approach would be to build an HPL `Array` associated to each (local) tile that will be used in heterogeneous computations. The ideal situation is to be able to use the same host memory region for the storage of the local HTA tile data and the host-side version of its associated HPL `Array`, as this would avoid the need for copies between both storages. Fortunately, the API of these datatypes is very rich, which enables programmers to achieve this ideal scenario using a relatively simple strategy illustrated in Figure 4.5. First, HTAs provide several methods to identify the tiles that are local to each process. In most situations, however, the identification is extremely simple, as the most widely pattern for the usage of HTAs is to make the distribution of the HTA along a single dimension, defining one tile per process. This is the case in our example, where line 1 gets the number of processes in variable `N` using the API of the HTA framework and line 2 builds a distributed HTA that places a 100 $\times$ 100 tile in each process, so that all the tiles together conform a (100$\times$`N`) $\times$ 100 HTA that is distributed by chunks of rows. Line 4 obtains the id `MYID` of the current process, so that choosing `h(MYID, 1)` will return the tile that is local to this process. Once this tile is identified, obtaining its storage is trivial, as HTAs provide a method `raw()` that returns a pointer to it. The final step involves making sure that the associated HPL `Array` uses the memory region that begins at that memory position for storing its host-side version of the array it handles. This is very easy to achieve in HPL, as the `Array` constructors admit a last optional argument to provide a pointer to this storage. This way, the `Array` can be built using

```
1 const int N = Traits::Default::nPlaces();
2 auto h = HTA<float, 2>({100, 100}, {N, 1});
3
4 const int MYID = Traits::Default::myPlace();
5 Array<float, 2> local_array(100, 100, h({MYID, 1}).raw());
```

Figure 4.5: Joint usage of HTAs and HPL

the syntax shown in line 5. From this point, any change on the local tile of HTA `h` will be automatically reflected in the host-side copy of the `Array` `local_array` and vice versa.

Coherency management

While HPL can automatically manage the coherency of its `Arrays` across all their usages in HPL, the changes that are due to HTA activities must be explicitly communicated to HPL. Again, this did not require any extension to the existing HPL API, as HPL Arrays, called `Arrays` for short, have a method `data` that allows to do this. The original purpose of this method is to obtain a pointer to the host-side copy of an `Array` so that programmers can access its data at high speed through this pointer, rather than by the usual indexing operators of the `Array`. The reason is that these operators check and maintain the coherency of the `Array` in every single access, thus having a considerable overhead with respect to the usage of a native pointer. The `data` method supports an optional argument that informs HPL of whether the pointer will be used for reading, writing or both, which is the default assumption when to specification is made. This is all the information HPL needs to ensure that the users will get coherent data from the pointer, and the devices will access a coherent view of the `Array` when it is used in the subsequent kernel invocations. Thus this simple mechanism also suffices to make sure that HTAs have a coherent view of the `Arrays` that have been modified by heterogeneous computations as well as to guarantee that HPL pushes to the heterogeneous devices fresh copies of those `Arrays` whose host-side copy has just been modified by an HTA operation.

Figure 4.6 shows an example of the memory management mechanism used in this initial approach. This example is based in the saxpy code of Figure 4.4. The

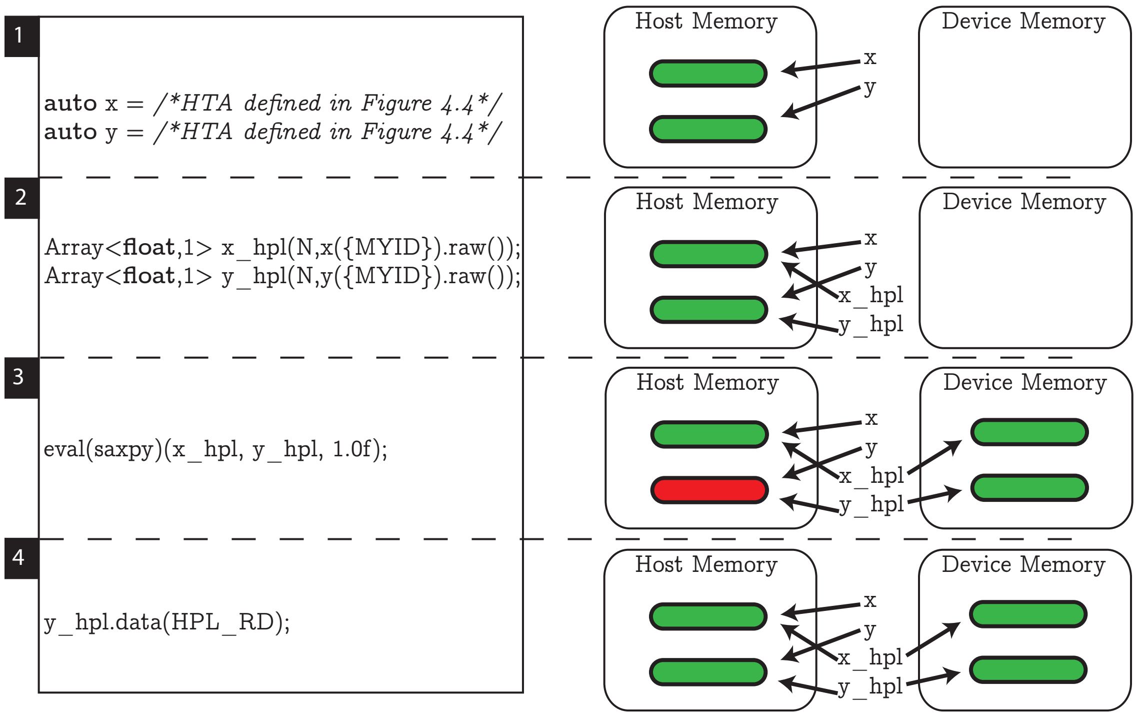

1
auto x = /*HTA defined in Figure 4.4*/
auto y = /*HTA defined in Figure 4.4*/
2
Array<float,1> x_hpl(N,x({MYID}).raw());
Array<float,1> y_hpl(N,y({MYID}).raw());
3
eval(saxpy)(x_hpl, y_hpl, 1.0f);
4
y_hpl.data(HPL_RD);
Host Memory
Device Memory
x
y
x_hpl
y_hpl

Figure 4.6: Example of the memory coherency mechanism implemented when HTA and HPL are separately used.

left part of the Figure contains the adapted HPL host code to work jointly with the HTAs. The code is divided into 4 sections. For each section, the right side of the Figure shows the evolution of the HTAs or Arrays stored both in the host and in the device memories. A green color means that the copy of an Array in that memory is valid, while the red color means that it is invalid. Section 1 of the code shows the declaration of the two HTAs involved in the code, `x` and `y`. The right side of the Figure shows that at this point, the HTAs are stored in host memory and their status is valid. Section 2 of the code shows the declaration of the associated HPL Arrays, `x_hpl` and `y_hpl`. The right side of the Figure shows that these HPL Arrays point to the same valid copy stored in host memory. Section 3 executes the kernel through an eval operation. The right side shows that after the kernel execution, the device memory has two valid copies of each array, while the copy of array `y` in the host memory is now invalid. The reason is that the saxpy kernel modifies this array in the devices. Finally, Section 4 updates this copy, which becomes valid, as the right side of the figure shows. Now, both memory spaces have valid copies.

The two problems commented in this Section are responsible for the main limitations of this separate approach, whose impact in the programmability and performance will be measured in Section 4.3. The most relevant limitations are related to the management of the two kinds of array containers used by both libraries independently. This lack of integration forces users to maintain manually two memory spaces for each kind of array and to use *ad-hoc* arrays as a workaround in order to perform efficient copies of regions of arrays. Namely, these applications require global HTAs to partition the data and enhance the communication among processes. Also, locally at each process, the kernels are executed in the accelerators by means of HPL, which requires the user to convert the local data of HTAs into local per-process HPL Arrays. Similarly, if the results of kernel executions have to be communicated through the mechanisms provided by HTAs, the HTA arrays have to be manually updated with the data of the associated HPL Arrays. In addition, the update of subregions of HTAs with the data of the HPL Arrays also suffers the lack of integration of the involved libraries. A possibility would be to update the whole host side of the HPL Arrays, which could have an unacceptable overhead, as it involves copying all the data when only a subset is needed. HPL also provides mechanisms to copy portions of arrays between the devices and the host, but while they avoid this performance overhead, they involve additional coding and thus higher cost in terms of programmability. The proposal presented in the next Section avoids these performance and programmability shortcomings thanks to a total integration of HTA and HPL.

4.2.2. Heterogeneous cluster programming with H^{2}TAs

The resulting library after the integration of HTA and HPL, H^{2}TA , can exploit both the general CPUs and the heterogeneous devices of a cluster using the same mechanisms as HTAs. Heterogeneous devices are exploited by means of kernels defined using any of the two strategies explained in Chapter 2: native kernels and HPL kernels. In both situations the H^{2}TAs are the only data structure required in the host side, while the heterogeneous kernels are written using the HPL `Array` data type (or OpenCL C strings) because no hierarchical sub-partitioning or tile-level manipulation is supported inside them. This representation allows to seamlessly mix in the same application parallel computations that are run in the CPUs (by means of HTA mechanisms) and operations that are performed in the accelerators (by means of kernels). The set of eligible accelerators has the same members as in any other HPL program. In addition, following the spirit of both the HTA and the HPL projects, the integration automatically keeps a coherent view of the H^{2}TAs across the CPUs and the devices of the system, avoiding explicit copies. The API for the heterogeneous executions is based on that of HPL because it facilitates the specification of details such as the kernel global and local spaces when the default values are not suitable or the best ones. This way, the most important component of the new API is the function `evalHTA(`*f*`)`, where *f* is the C++ function associated to an heterogeneous kernel, which plays a role analogous to that of `eval(`*f*`)` in HPL, but accepting as inputs H^{2}TAs or scalars. Just as in `hmap`, the H^{2}TAs should have the same top-level structure, that is, number of dimensions and top level tiles per dimension, so that the associated tiles of each one of the H^{2}TAs would be processed together in the same kernel execution.

Figure 4.7 exemplifies the high level programming style enabled by our extension. H^{2}TAs keep the same name and creation process as the original HTAs. Also, their usages in CPU as well as communications by means of assignments follow exactly the same notation as in the original HTA. When heterogeneous computing is required, an `evalHTA` rather than an `hmap` invocation is used (lines 6 and 8), all the complexity (buffer creations, transfers, kernel compilations, etc.) being hidden from the user. The heterogeneous kernels in H^{2}TA can be implemented using any of the two mechanisms exemplified in Figure 4.8 using exactly the same notation than that of HPL. For example the `saxpy` kernel used in line 6 of Figure 4.7 can be the one from Figure 4.8. Also, in between `evalHTA` and the kernel arguments

```
1 auto h1 = HTA<float, 1>::alloc({ {1000}, {NNodes} });
2 auto h2 = HTA<float, 1>::alloc({ {1000}, {NNodes} });
3 float alpha;
4 ...
5 h1(Tuple(1,Nodes−1)) = h2(Tuple(0,NNodes−2)) + 4;
6 evalHTA(saxpy)(h1, h2, alpha);
7 hmap(user_CPU_function, h2, h1);
8 evalHTA(user_GPU_kernel)(h2(Tuple(1, NNodes−1))[Tuple(0, 499)]), h1);
```

Figure 4.7: H^{2}TA example code

```
1  void saxpy(Array<float,1> y, Array<float,1> x, Float alpha)
2  {
3     y[idx] = alpha * x[idx] + y[idx]; // idx is an HPL variable that contains the
          global thread ID
4  }
5  ...
6  const char *string = "__kernel void saxpy(__global float *y, __global float *x,
       float alpha) { ... }";
7
8  void saxpy_handle(InOut< Array<float,1> > y, In< Array<float,1> > x,
       Float alpha) {}
9  ...
10 Array<float, 1> x(1000), y(1000);
11 float alpha;
12
13 eval(saxpy)(y, x, alpha);
14
15 nativeHandle(saxpy_handle, "saxpy", string);
16 eval(saxpy_handle)(y, x, alpha);
```

Figure 4.8: HPL example code

it is possible to insert the same `global`, `local` and `device` modifiers as in the case of `eval`. The specifiers have been extended to support H^{2}TAs as their arguments. These H^{2}TAs should have the same top-level structure as the `evalHTA` arguments so that the value of each given tile would be used to parameterize the kernel execution on the corresponding argument tiles. Also, in the case of the `device` modifier, it is possible to provide a kind of device (GPU, CPU or accelerator) rather than a specific device. In this case the tiles are processed in their home node using devices of that kind. If there are several tiles and more than one device, the tiles in the same node are evenly distributed on the existing devices to maximize the parallelism of the runtime. Line 8 exemplifies how the H^{2}TAs used in the heterogeneous executions do not need to be the whole data structures defined by the user. Rather, one can choose to operate on a subset of their tiles, and thus only in some nodes of the cluster, assuming we are executing on an heterogeneous cluster, which is our target. In addition, in each tile we can choose to operate on the whole tile or only in a portion using the high level indexing notation of H^{2}TAs.

As we can see the result is very powerful and easy to use thanks to its intuitive and simple semantics. Users manipulate the abstract arrays required by their application rather than the underlying different copies of them that are required because of the disjunct memories of each host and its devices. The data objects seen by the programmer enjoy a simple sequential consistency model [66] that is automatically provided by their data type. This feature coupled with the single threaded view of the programming model and the global view of the distributed data structures that H^2TAs inherit from HTAs, largely simplify the programming of parallel applications.

Regarding the implementation details, the copies of the same tile that are used in different memories are managed by the runtime under a multiple-readers/single-writer (MRSW) policy [95] with an invalidation protocol on writes [70], which together with a lazy copy policy that only updates a copy when it is actually required, minimizes the transfers between the host and the devices. The cost of data copies was also reduced to the minimum possible one by ensuring that whenever only a portion of a tile is required in a memory where it is outdated or inexistent, only that region is copied, rather than the whole tile. Relatedly, the H^2TA runtime remembers which portions of each tile are updated or outdated in each memory, so that the coherency mechanism granularity dynamically adjusts to the size of the tile regions manipulated by the user, which is needed to ensure a minimum number of transfers with the smallest possible cost. The HPL runtime provides other critical performance optimizations such as the caching of buffers and kernels to avoid repetitive creation processes, while the HTA implementation provides other performance enhancement techniques such as the caching of HTAs or asynchronous communications between nodes [46]. Finally, just as the libraries it integrates, H^2TAs also heavily rely on the compile-time polymorphism and optimizations enabled by C++ templates [17] rather than in the more expensive dynamic polymorphism also supported by this language.

4.3. Evaluation

A high level approach to program a system must show programmability improvements with respect to existing alternatives to motivate its interest. Also, its abstractions must incur in reasonable performance costs. Thus, both sides of the problem are tackled in section, which evaluates separately the independent usage of HTA and HPL (HTA+HPL) and the integrated H^2TA library. In order to better assess the advantages of the integrated H^2TA with respect to HTA+HPL, our evaluation takes as baseline MPI+HPL versions, so that the improvement that HPL means with respect to the usage of standard OpenCL bindings is already present in the baseline. This comparison is fairer because it removes the improvement that HPL provides with respect to OpenCL.

The main characteristics of the benchmarks used in the evaluation are shown in Table 4.1, where the first column represents the number of source lines of code excluding comments and empty lines (SLOCs) of the host side of their baseline HPL+MPI version. The size of the kernels has been here dismissed because the H^2TA versions use exactly the same OpenCL kernels, so they play no role in the comparison. The remaining columns contain the number of kernels that are invoked just once during their execution, the number of kernels that are invoked inside loops and the nature of the data exchanges between processes they have. As we can see, these programs present very different patterns, going from codes with no exchange of information among processes to iterative applications with several data exchanges in each iteration. We describe now in turn the basics of these benchmarks, which have already used in previous chapters.

The first two benchmarks are two of the OpenCL codes developed in [89], namely EP and FT. The first one gets its name from being embarrassingly parallel, although it requires inter-node communications for reductions that happen at the end of the main computation. The second one repetitively performs Fourier Transforms on each one of the dimensions of a 3D array. This requires fully rotating the array in each main iteration of the algorithm, which implies an all-to-all communication between the cluster nodes. The third problem, MMRow, is a distributed single precision dense matrix product in which each node computes a block of rows of the result matrix. The fourth benchmark is a simulation on time of the evolution of a pollutant on the surface of the sea depending on the tides, oceanic currents, etc. called ShWa and parallelized for a cluster with distributed GPUs in [103]. The simulation partitions the sea surface in a matrix of cells that interact through their borders. Thus in every time step each cell needs to communicate its state to its neighbors, which implies communications when they are assigned to different nodes. The distributed arrays are extended with additional rows of cells to keep

Benchmark	SLOCs host	Unique invocation	Repetitive invocation	Data exchanges
EP	248	1 kernel		final reduction
FT	1263	3 kernels	7 kernels	all to all
MMRow	184	1 kernel		none
ShWa	386		3 kernels	stencil and reduction
Canny	209	4 kernels		stencil

Table 4.1: Benchmarks characteristics.

this information from the neighbor cells in other nodes, following the well known ghost or shadow region technique. The fifth application is Canny, an algorithm that finds edges in images by following a series of four steps, each one implemented in a different kernel. The parallelization comes from the processing or different regions of the kernel in parallel. Communications between neighboring regions of arrays used in the computations are required for some of the kernels. This gives place to the application of the already mentioned shadow region technique, which replicates portions of the borders of the distributed arrays which need to be updated when the actual owner of the replicated portion (rows, in the case of this algorithm) modifies it.

4.3.1. Programmability

Our programmability comparison will be based on three metrics of this kind, profusely used along the Thesis: SLOCs, the cyclomatic number [77] and the programming effort [55].

Figure 4.9 shows the reduction of the three metrics used in applications written using separately HTA and HPL (HTA+HPL) and the proposed H^{2}TA library, compared to the baseline counterparts written using MPI and HPL (MPI+HPL). The measurements are based on the host side of the applications, since kernels are identical in the three versions. There are two kinds of benchmarks attending to the strength of the reduction. In the programs that have none or very little communication, which are EP and MMRow, the programmability of the three versions are very similar because the sources have few differences and HPL already provides very good programmability metrics to the baseline for the exploitation of hetero-

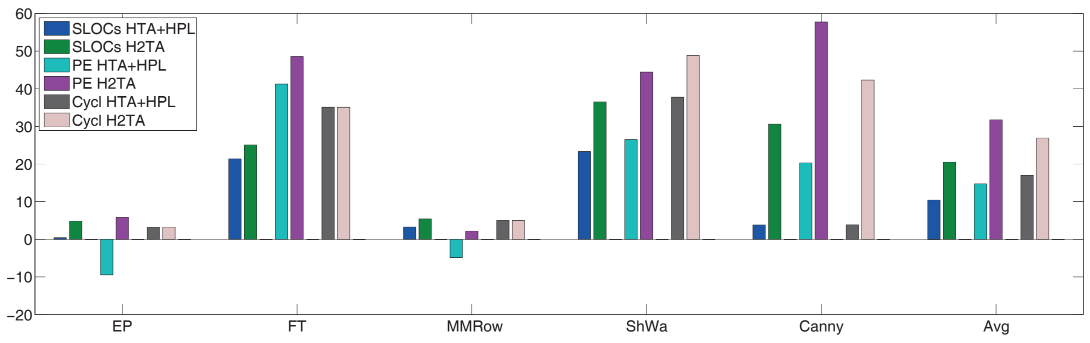


Figure 4.9: Reduction of programming complexity metrics of HTA+HPL and H^{2}TA programs with respect to versions based on MPI+HPL.

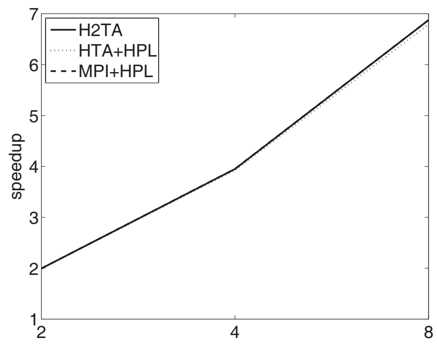
geneity. However, while H^{2}TA always obtains a positive result, the HTA + HPL version requires more programming effort than the baseline because of the duality of the arrays used. This bad behavior is mitigated in the rest of the benchmarks because of their larger complexity. The other group consists of the benchmarks with more complex communication patterns, which make the applications benefit more from the high level semantics of the HTAs. In the case of ShWa and Canny this complexity is related with the management of the ghost regions needed in these benchmarks with stencil computations. Finally, the rotation of the 3D array that requires FT, which implies an all-to-all communication coupled with transpositions, is well covered by the HTA interface, as it includes a rich set of global collective operations. In this second group, although HTA+HPL usually obtains good results, H^{2}TA always improves them, except in the cyclomatic number of FT, where they achieve the same value. The large improvement that H^{2}TA obtains with respect to HTA+HPL for all the metrics in ShWa and Canny is particularly outstanding. The reason is related with the synchronization of the ghost regions of these two applications based on stencil computations. While HTA+HPL and MPI+HPL need a more manual process to keep the memory coherence of the ghost regions, H^{2}TA allows a more convenient and concise process thanks to its better integration.

4.3.2. Performance

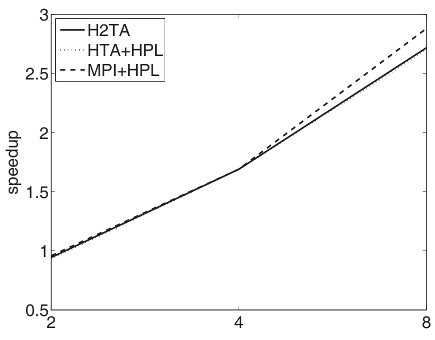
Two different heterogeneous clusters were used to assess the performance of our proposal. The first one, called Fermi, has 4 nodes with an Intel Xeon X5650 CPU with 6 cores and 12 GB of memory each. Additionally, each node is connected to 2 Nvidia M2050 GPUs with 3GB per GPU. The interconnection network is a QDR InfiniBand. The second system, called K20, has 8 nodes. Each one has a 2xIntel Xeon E5-2660 8-core CPUs and 64 GB of RAM. In this case, the accelerator present in each node is a K20m GPU with 5 GB. The interconnection network for this system is a FDR InfiniBand. The compiler used in both cases is g++ 4.7.2 with optimization level O3 and the MPI implementation is the OpenMPI 1.6.4. The problem sizes used for the NPB tests were classes D and B for EP and FT, respectively. MMRow multiplies two matrices of 8192 $\times$ 8192 elements, ShWa computes the evolution of a mesh of 1000 $\times$ 1000 volumes and Canny filters an image of 9600 $\times$ 9600 pixels.

Figures 4.10 and 4.11 show the speedups of the H^{2}TA, the HTA+HPL and the MPI+HPL versions of the benchmarks when using a varying number of accelerators in the Fermi and K20 clusters, respectively. The baseline for the speedup in each figure is an HPL version, without MPI or HTAs, that uses a single accelerator of the corresponding cluster.

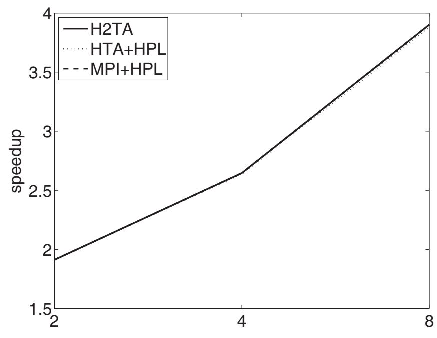
The HTA-based versions have negligible performance differences. The exception is ShWa where the average overhead of HTA+HPL with respect to H^{2}TA rises up to 1%, reaching a maximum of 4.5% for 8 devices in K20. Here, the lack of integration of both libraries increases the need of additional code to maintain manually the ghost regions in stencil computations (see Section 4.2.1), which increases the overhead. Additionally, the behavior of the MPI-based applications is similar to that of the H^{2}TA applications for most cases, which, taking into account the successful history of MPI in HPC, further confirms the robustness of our proposal. In fact, only for FT the differences are slightly different, around 2% and 5% of average overhead in Fermi and K20 respectively. Not surprisingly, the overhead is more apparent for FT, where the HTA takes care of a very complex all-to-all communication pattern, which is also the reason behind the very strong reductions of the measured programmability metrics. All the other overheads are below 1%. This way, wen we look at the big picture, the peformance overhead of H^{2}TA with respect to MPI+HPL is minimal, with an average of just 0.6% and 1.3% in the Fermi and K20 clusters, respectively. Clearly, the large programmability improvements measured in Sect. 4.3.1 totally justify this reduced overhead.



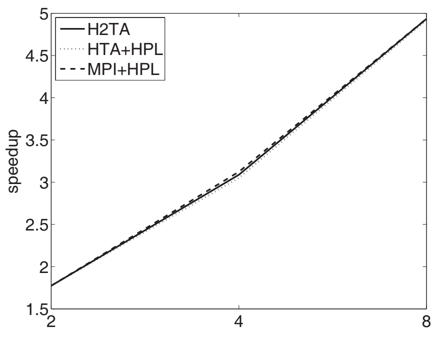
(a) EP



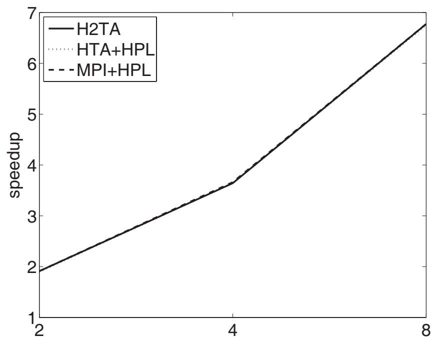
(b) FT



(c) MMRow



(d) ShWa

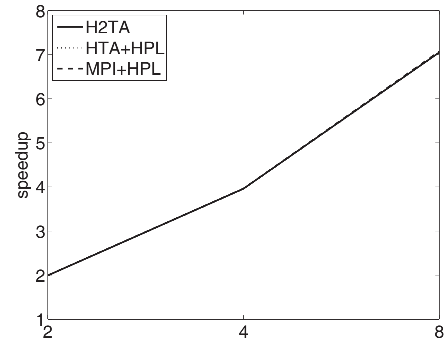


(e) Canny

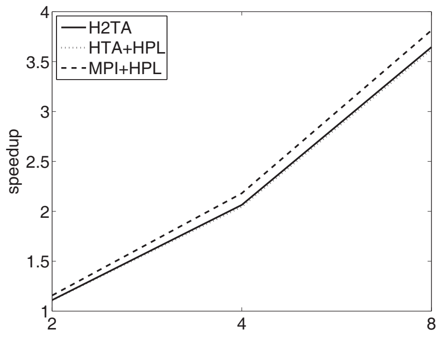
Figure 4.10: Speedup of the executions in Fermi using multiple devices with respect to an execution using a single device

4.4. Related work

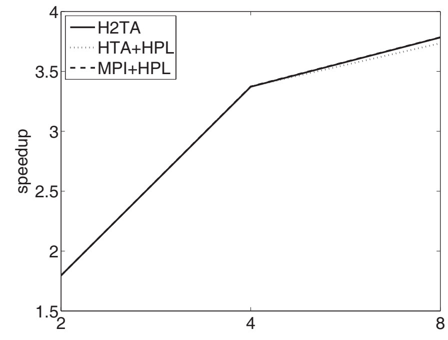
While there has been a considerable amount of work on the enhancement of the programming of heterogeneous clusters in the past few years, as we will see



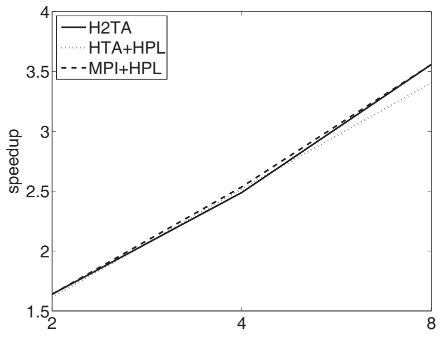
(a) EP



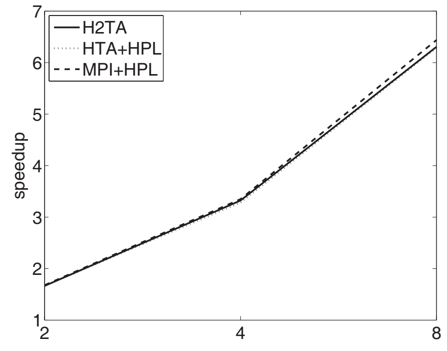
(b) FT



(c) MMRow



(d) ShWa



(e) Canny

Figure 4.11: Speedup of the executions in K20 using multiple devices with respect to an execution using a single device

now, the alternatives proposed operate at a lower level than H^2TAs. Like ours, most of these proposals take the form of libraries, as this facilitates code reuse and requires less development effort than approaches that require developing or modifying a compiler. Some projects are based on vendor-specific tools, thus severely limiting their portability. This is the case of [94], which extends CUDA to support its programming model in clusters of GPUs and multi-GPU systems. Similarly, [67] extends the MPI API to support message-passing point-to-point and collective communications of data stored on the GPU using the CUDA interface, thus keeping the semantic level of MPI+CUDA applications. As a final example, enabling the use of shared remote GPUs in clusters is the aim of the distributed CUDA API called rCUDA [35].

As for the OpenCL-based approaches [7, 11, 16, 40, 52, 61, 63, 83, 106] to facilitate the programming of clusters, most of them expose abstractions, and thus APIs, that are at the low level of OpenCL, being sometimes in fact nearly identical. The most outstanding efforts of abstraction have been performed by the Many GPUs Package (MGP) [16] and libWater [52]. MGP allows to run unmodified OpenCL applications in clusters on top of MOSIX VCL. It also supports a C++ object-oriented API that, while simplifying the process, is still based on low-level concepts such as buffers, contexts or tasks with explicit enqueuings and synchronizations. In addition it presents important restrictions to the processing of distributed data; for example only a scatter and a gather communication patterns are supported and only one task can be associated to the data they distribute. MGP also has task-based OpenMP-like directives that are restricted to the execution of individual kernels in each node. Regarding [52], it relies on explicit kernel creation processes, buffers that must be manually associated to specific devices and that require the user to specify the read and write transfers on them, as well as synchronizations based on events. Therefore it is at a considerably lower level than H^{2}TAs with their globally distributed data structures that abstract away any idea of buffer and make totally transparent all the management related to heterogeneous devices.

A proposal to program heterogeneous clusters based on compiler directives is the task-based programming model of OmpSs [27], which lacks the H^{2}TA fine-grained control over device selection, globally distributed data structures and implicit data-parallelism across cluster nodes. Finally, StarPU is a large project with APIs based on libraries, directives and language extensions that supports two programming models for clusters. While [14] operates at a lower level exposing MPI-like messages to the programmer, [13] task-based approach is quite similar to that of OmpSs. As a result it shares similar limitations, the most important difference being that it allows to define distributed arrays as a collection of tiles located in different nodes, but lacking all the tile-level semantics, advanced syntax, collective manipulation capabilities and data-parallel operations of H^2TAs.

4.5. Conclusions

Developing parallel applications for heterogeneous clusters requires simultaneously facing the complexity inherent to distributed memory environments and heterogeneous systems, which leads to increased development times, debugging difficulty, maintenance costs, etc. While there have been many proposals to improve the programmability of these systems, the ones we are aware of, either require programmers to manage low level details such as explicit buffers or communications, or support very restricted computation and communication patterns. In this chapter we propose a high level approach to program these systems that is based on an abstract data type that represents an array partitioned into tiles. Such tiles can be distributed on a cluster and processed in parallel following data-parallel semantics, giving a global view of the distributed data structure and exposing a single high-level thread of execution to the user. The data type, called Heterogeneous Hierarchically Tiled Array (H^2TA), extends the existing Hierarchically Tiled Array (HTA), which was oriented to traditional distributed memory clusters, adding support for arbitrary computing devices that support OpenCL, thus maximizing the portability of our solution. Rather than exposing the user to the raw OpenCL API, H^2TA relies on the Heterogeneous Programming Library (HPL), which substantially reduces the development complexity of OpenCL-based applications. H^2TAs inherit the high level notation of HTAs for communications between cluster nodes and add total transparency and automated management of the kernels, buffers, transfers between host and devices memory, etc. required by heterogeneous computing.

H^2TAs vastly improves the programmability of heterogeneous clusters with respect to existing approaches. Even if we consider baselines that exploit the advantages of HPL but resort to the traditional MPI library for communications, H^2TAs reduce their programming complexity metrics by an average of 20.5%, 31.8% and 26.9% in terms of SLOCs, Halstead's programming effort and cyclomatic number, respectively. These improvements are twice larger than those achieved by separately using the HTA and HPL libraries, which further justifies the interest of this proposal. Also, the H^2TA runtime is very light, with average slowdowns around 1% that peak at 6% with respect to a MPI-based solution, thus making our proposal a very appealing approach for the programming of current complex heterogeneous clusters.

Chapter 5

Conclusions

Over the years, High Performance Computing (HPC) has been driven by general-purpose traditional CPUs. Clusters, historically seen as aggregations of computers with one or more CPUs, have permitted the execution of parallel applications until now by means of frameworks such as OpenMP or MPI. These frameworks are very mature today and are profusely used in shared and distributed memory systems, respectively. The introduction of heterogeneous devices, such as FPGAs, GPUs or many-core coprocessors in HPC, has awaken the interest in creating programming tools for these platforms. Most alternatives to program these devices have a strong dependence on a specific device or vendor. OpenCL is the first standard that intends to decouple the developed code from a specific vendor or device family, by providing effective portability of the code among platforms. A big number of manufacturers have provided their own implementations of the OpenCL standard for their devices. As a consequence, OpenCL codes can be executed in a wide range of heterogeneous devices without changing the source code. The main limitations of OpenCL are: (1) the programming effort required to program OpenCL applications is high for programmers not familiarized with parallel programming, (2) OpenCL does not provide automatic performance portability, thus, in order to maximize the performance, we have to hand-tune a code for each platform where it is executed, and (3) OpenCL does not support the programming of distributed systems, it has to be combined with MPI for this purpose. A number of works have addressed these limitations in many ways. One of them is the Heterogeneous Programming Library

(HPL) [22], which is based on OpenCL and noticeably facilitates the development of single-device applications, addressing the first one of the mentioned OpenCL limitations. This Thesis deepens in the usage of HPL as tool to overcome the aforementioned OpenCL limitations by: improving even further the programmability of HPL kernels, improving performance portability and providing support for the programming of distributed systems composed of heterogeneous nodes. In addition, originally HPL did not provide mechanism to program multi-device applications, where several devices of the same node are used at the same time. This limitation has been also overcome in this Thesis.

One of the main limitations of HPL was that the HPL kernels had to be written in an embedded language similar to C++. The usage of native OpenCL kernels was not supported, which limited the usage of legacy OpenCL code and low-level or vendor-specific OpenCL optimizations. This limitation has been overcome in Chapter 2 with the extension of HPL to support native OpenCL kernels, in addition to those written in the original embedded language. This chapter also introduces other new features to facilitate the writing of HPL kernels using the embedded language. The evaluation of these HPL extensions provided very satisfactory results. Along the whole Thesis, the evaluation of each new HPL feature has been done both in terms of programmability and performance. The programmability evaluation relies on three metrics: the source lines of code, programming effort [55] and cyclomatic number [77] (SLOCs, PE and CN from now on, respectively). The performance is always evaluated by comparing the execution of the library with one new feature to a baseline. When HPL kernels are written using the embedded language, the reduction of the SLOCs, PE and CN of the whole program is 34%, 44% and 30%, respectively with respect to baselines written using OpenCL C++. Meanwhile, the average overhead of HPL in terms of performance is below 5%. It deserves to be mentioned that this performance evaluation was performed using devices of different vendors. Following this trend, the support of native kernels has also obtained good performance and programmability results. This way, HPL reduces the SLOCs and PE of the host program in 23% and 42%, respectively, keeping the performance overhead close to zero. This overall improvement of the metrics can be also seen after the comparison done between HPL and one of the most mature and analogous approaches, ViennaCL [86]. As in Chapter 2, all the improvements done on HPL along the Thesis were evaluated by means of verifiable benchmarks including a real application of shallow water simulation presented in [102].

The lack of multi-device support was an important limitation of the initial version of HPL. This limitation has become increasingly more important, as it is usual that one node or computer is composed of several OpenCL capable devices, and while OpenCL can be used to program applications that use several devices at the same time, this requires an important programming effort. In order to tackle with this reality, HPL has been extended to support the programming of multi-device applications. This extension implied changes both in the internals of HPL and in its programming API. Regarding the internals, this extension required the definition of a new memory coherency mechanism allowing HPL to support the different copies of an HPL Array located in the chosen devices. The HPL API was also extended to support the usage of several devices at the same time. The new memory coherency mechanism, tested with a simple multi-device implementation, achieved a reduction of SLOCs and the PE of 27% and 43% respectively, with respect to OpenCL C++ baseline implementations. The performance was also improved thanks to the adaptive nature of HPL given by the automatic selection of the more efficient method to perform the exchange of data between the memory spaces of the devices. This is an example of performance portability mechanism introduced by HPL in this Thesis. This fact is clearly visible in the applications with a larger amount of data exchanges among devices, achieving an average speedup of 28% and a maximun of 106% with respect to OpenCL C++ baselines for this kind of applications.

The changes in the HPL API are mainly due to the mechanisms proposed to distribute a given workload among different devices. The proposals tested go from the most manual mechanism based on *subarrays*, where the user has to select the portion of the Array (subarray) to be processed in each moment, to the most automatic one based on annotations, where the user only specifies the dimension of the Arrays to be divided among the devices. There is an intermediate mechanism based on execution plans, which provides more freedom to the user but avoiding the definition of subarrays. This last mechanism optionally allows the automatic balancing of the total workload among the devices available in the system in a very easy way. In particular, the user only has to specify the devices and HPL will compute the most suitable workload distribution for them. These three mechanisms reach a maximum PE reduction of 76.7% with respect to OpenCL C++ baselines. Additionally, thanks to the adaptive runtime of HPL, the improvement of the performance reaches a peak at 146% in comparison with the same baselines. Moreover, the automatic workload distribution supported by the mechanism based on execution plans also obtains optimal results in the majority of the experiments. Actually, in the worst case our automatic workload distribution is only 6.3% slower than the best workload distribution obtained by means of an exhaustive search. The last improvement for the support of multi-device applications in HPL proposed in this Thesis consists in a new mechanism for the automatic update of the ghost regions that usually appear in applications with stencil computations parallelized using several nodes or devices. This proposal, called `syncGhosts`, was evaluated through several experiments including real applications. The results obtained are even better than those measured after the application of the other distribution mechanisms proposed. This way, while following a mechanism based on annotations the PE average reduction reached 20.5% in comparison to use the subarrays mechanism, this same reduction rises to 79.5% when the scheme based on annotations is used coupled with the synchGhosts technique. Particularly, for the image processing application, CANNY, the reduction reaches a maximum at 96.7%. In order to measure more accurately the impact of this mechanism on the total execution time, we measured its performance in systems with two and three devices and in none of them the performance differences were larger than 1%, ensuring the robustness of the implementation with more complex data distribution schemes.

The last issue tackled in this Thesis is the programming of distributed systems composed of nodes with heterogeneous devices. This is achieved through the integration of the Hierarchically Tiled Arrays (HTAs) framework [6] and the HPL library, giving place to the Heterogeneous Hierarchically Tiled Arrays (H^2TA) framework. This library is based on the HTA abstract data type, which represents an array hierarchically divided in tiles. These tiles can be distributed on a cluster and can be also processed in parallel providing a global view of the distributed data. The H^2TA proposed library allows programmers to use the OpenCL devices available in a cluster taking advantage of the combined properties of the HTAs and the simple API and semantics of HPL. The results obtained are very positive. For example, comparing the programmability of H^2TA with that of an approach consisting in combining HPL with the MPI library for the communications, which already enjoys a programmability improvement with respect to low level solutions, H^{2}TA reduces the SLOCs, PE and CN by 20.5%, 31.8% and a 26.9%, respectively. These results are also better than those obtained using both libraries separately as it was demonstrated in Chapter 4 by comparing applications based on H^{2}TA with versions written combining HTAs and HPL. Regarding performance, the average overhead of H^{2}TA with respect to solutions based on MPI is below 1%, which taking into account the programmability results, further justifies the interest of the library.

5.1. Future Work

In the future, the programmability of HPL can be improved by integrating more object-oriented features in its kernels. Another important aspect of HPL that can be improved is performance portability. This feature can be extended in HPL by adding optimization annotations in the kernels and by adding tuning capabilities to its runtime.

Other line of future work can be to improve HPL including convenient features for the development of imbalanced applications, which are applications where each executing thread performs a different amount of work. The fact that some threads can perform much more work than others can result in a performance bottleneck. In these cases, the redistribution of the original workload into a more balanced one could be very beneficial, particularly if this can be done with as little user intervention and complexity as possible. This improvement could be also ported to multi-device systems in order to balance the workload of these applications in such environments.

Regarding H^{2}TA , a possible ambitious line of future work for this project would be to implement an HTA-aware compiler that improves the programmability of these systems and applies optimizations that are more difficult to identify using a library-based implementation.

Bibliography

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming.* Addison-Wesley, 2004.

[2] A. Acosta and F. Almeida. Skeletal based programming for dynamic programming on multigpu systems. *The Journal of Supercomputing*, 65(3):1125–1136, 2013.

[3] A. Acosta, V. Blanco, and F. Almeida. Towards the dynamic load balancing on heterogeneous multi-gpu systems. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 646–653, July 2012.

[4] A. Acosta, R. Corujo, V. Blanco, and F. Almeida. Dynamic load balancing on heterogeneous multicore/multigpu systems. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 467–476, June 2010.

[5] N. Aeronautics and S. Administration. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/, last access Dec 30, 2013.

[6] G. Almási, L. De Rose, B. B. Fraguela, J. E. Moreira, and D. A. Padua. Programming for locality and parallelism with Hierarchically Tiled Arrays. In *16th Intl. Workshop on Languages and Compilers for Parallel Computing, (LCPC 2003)*, pages 162–176, 2003.

[7] A. Alves, J. Rufino, A. Pina, and L. Santos. clOpenCL - supporting distributed heterogeneous computing in HPC clusters. In *Euro-Par 2012: Parallel Pro-*

cessing Workshops, volume 7640 of *Lecture Notes in Computer Science*, pages 112–122. Springer, 2013.

[8] AMD. AMD stream computing user guide, 2009. v1.4.0a.

[9] AMD and AccelerEyes. clMath. http://github.com/clMathLibraries.

[10] A. AMD. Close to metal. *Technology Unleashes the Power of Stream Computing.*, pages 11–4147, 2006.

[11] R. Aoki, S. Oikawa, T. Nakamura, and S. Miki. Hybrid OpenCL: Enhancing OpenCL for distributed processing. In *IEEE Intl. Symp. on Parallel and Distributed Processing with Applications (ISPA 2011)*, pages 149–154, 2011.

[12] P. J. Ashenden. *The Designer's Guide to VHDL, Volume 3, Third Edition (Systems on Silicon) (Systems on Silicon).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2008.

[13] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault. StarPU-MPI: Task programming over clusters of machines enhanced with accelerators. In *Recent Advances in the Message Passing Interface*, volume 7490 of *Lecture Notes in Computer Science*, pages 298–299. Springer Berlin Heidelberg, 2012.

[14] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *IEEE 16th Intl. Conf. on Parallel and Distributed Systems (ICPADS 2010)*, pages 291–298, Dec 2010.

[15] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[16] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *2010 IEEE Intl. Conf. on Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*, pages 1–7, 2010.

[17] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples.* Addison-Wesley Longman Publishing Co., Inc., 1994.

[18] O. Beckmann, A. Houghton, M. Mellor, and P. H. J. Kelly. Runtime code generation in C++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*, pages 291–306. Springer Verlag, 2004.

[19] N. Bell and J. Hoberock. *GPU Computing Gems Jade Edition*, chapter 26. Morgan Kaufmann, 2011.

[20] S. Bihan, G. Moulard, R. Dolbeau, H. Calandra, and R. Abdelkhalek. Directive-based heterogeneous programming. a GPU-accelerated RTM use case. In *Proc. 7th Intl. Conf. con Computing, Communications and Control Technologies*, july 2009.

[21] D. Blythe. The direct3d 10 system. *ACM Transactions on Graphics (TOG)*, 25(3):724–734, 2006.

[22] Z. Bozkus and B. B. Fraguela. A portable high-productivity approach to program heterogeneous systems. In *2012 IEEE 26th Intl. Parallel and Distributed Processing Symp. Workshops PhD Forum (IPDPSW)*, pages 163–173, may 2012.

[23] J. Breitbart. CuPP - a framework for easy CUDA integration. In *IEEE Intl. Symp. on Parallel Distributed Processing (IPDPS 2009)*, pages 1 –8, may 2009.

[24] S. Breuer, M. Steuwer, and S. Gorlatch. Extending the SkelCL Skeleton Library for Stencil Computations on Multi-GPU Systems. In *High-Performance Stencil Computations*, Vienna, Austria, 2014.

[25] I. Buck, T. Foley, D. Horn, J. Sugerman, P. Hanrahan, M. Houston, and K. Fatahalian. Brookgpu, 2003.

[26] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. Badia, E. Ayguade, and J. Labarta. *Euro-Par 2011 Parallel Processing: 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part I*, chapter Productive Cluster Programming with OmpSs, pages 555–566. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[27] J. Bueno, J. Planas, A. Duran, R. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of GPU clusters with OmpSs. In *2012 IEEE 26th Intl. Parallel Distributed Processing Symp. (IPDPS)*, pages 557–568, 2012.

[28] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proc. 16th ACM symp. on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, 2011.

[29] M. Christen, O. Schenk, and H. Burkhart. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687, May 2011.

[30] M. Cole. *Algorithmic skeletons: structured management of parallel computation.* MIT Press, 1991.

[31] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley Professional, 2000.

[32] A. Danalis, G. Marin, C. Mccurdy, J. Meredith, P. Roth, K. Spafford, and J. Vetter. The Scalable HeterOgeneous Computing (SHOC) benchmark suite. In *Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU3)*, pages 63–74, 2010.

[33] C. Ding and Y. He. A ghost cell expansion method for reducing communications in solving PDE problems. In *Proc. Supercomputing 2001*, SC 2001, pages 50–50, 2001.

[34] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[35] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 Intl. Conf. on High Performance Computing and Simulation (HPCS 2010)*, pages 224–231, 2010.

[36] A. Duran, E. Ayguadé, R. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[37] F. Dütsch, K. Djelassi, M. Haidl, and S. Gorlatch. HLSF: A High-Level; C++-Based Framework for Stencil Computations on Accelerators. In *Proceedings of the Second Workshop on Optimizing Stencil Computations*, WOSC '14, pages 41–4, New York, NY, USA, 2014. ACM.

[38] J. Enmyren and C. Kessler. SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proc. 4th intl. workshop on High-level parallel programming and applications*, HLPP '10, pages 5–14, 2010.

[39] S. Ernsting and H. Kuchen. Algorithmic skeletons for multi–core, multi–GPU systems and clusters. *International Journal of High Performance Computing and Networking*, 7(2):129–138, 2012.

[40] B. Eskikaya and D. Altilar. Distributed OpenCL distributing OpenCL platform on network scale. *IJCA Special Issue on Advanced Computing and Communication Technologies for HPC Applications*, ACCTHPCA(2):26–30, July 2012.

[41] J. F. Fabeiro, D. Andrade, and B. B. Fraguela. OCLoptimizer: An iterative optimization tool for OpenCL. In *Proc. Intl. Conf. on Computational Science (ICCS 2013)*, pages 1322–1331, 2013.

[42] J. F. Fabeiro, D. Andrade, and B. B. Fraguela. Writing a performance-portable matrix multiplication. *Parallel Computing*, 52:65 – 77, 2016.

[43] J. F. Fabeiro, D. Andrade, B. B. Fraguela, and R. Doallo. Writing self-adaptive codes for heterogeneous systems. In *Proc. 20th Intl. Conf. Euro-Par 2014 Parallel Processing*, pages 800–811, 2014.

[44] J. F. Fabeiro, D. Andrade, B. B. Fraguela, and R. Doallo. Automatic generation of optimized opencl codes using ocloptimizer. *Comput. J.*, 58(11):3057–3073, 2015.

[45] P. Faber and A. Größlinger. A Comparison of GPGPU Computing Frameworks on Embedded Systems. *IFAC-PapersOnLine*, 48(4):240 – 245, 2015.

13th IFAC and IEEE Conference on Programmable Devices and Embedded SystemsPDES 2015.

[46] B. B. Fraguela, G. Bikshandi, J. Guo, M. J. Garzarán, D. Padua, and C. von Praun. Optimization techniques for efficient HTA programs. *Parallel Computing*, 38(9):465–484, Sept. 2012.

[47] B. B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas. Programming the FlexRAM parallel intelligent memory system. *ACM SIGPLAN Not.*, 38(10):49–60, June 2003.

[48] R. A. V. D. Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency and Computation: Practice and Experience*, 9(4):255–274, Apr. 1997.

[49] C. H. González and B. B. Fraguela. A framework for argument-based task synchronization with automatic detection of dependencies. *Parallel Computing*, 39(9):475–489, Sept. 2013.

[50] S. Gorlatch and M. Cole. Parallel skeletons. In *Encyclopedia of Parallel Computing*, pages 1417–1422. 2011.

[51] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer. LibWater: heterogeneous distributed computing made easy. In *Intl. Conf. on Supercomputing (ICS'13)*, pages 161–172, 2013.

[52] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer. A uniform approach for programming distributed heterogeneous computing systems. *Journal of parallel and distributed computing*, 74(12):3228–3239, 2014.

[53] D. Grewe and M. F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using OpenCL. In *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 286–305. Springer Berlin Heidelberg, 2011.

[54] J. Guo, G. Bikshandi, B. B. Fraguela, and D. Padua. Writing productive stencil codes with overlapped tiling. *Concurrency and Computation: Practice and Experience*, 21(1):25–39, 2009.

[55] M. H. Halstead. *Elements of Software Science.* Elsevier, 1977.

[56] T. Han and T. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Trans. on Parallel and Distributed Systems*, 22:78–90, 2011.

[57] J. Herrington. *Code Generation in Action.* Manning Publications, 2003.

[58] High Performance Fortran Forum. High Performance Fortran Specification Version 2.0, January 1997.

[59] IBM, Sony, and Toshiba. *C/C++ Language Extensions for Cell Broadband Engine Architecture.* IBM, 2006.

[60] IBM, Sony, and Toshiba. *Cell Broadband Engine Architecture.* IBM, 2006.

[61] P. Kegel, M. Steuwer, and S. Gorlatch. dOpenCL: Towards uniform programming of distributed heterogeneous multi-/many-core systems. *J. Parallel Distrib. Comput.*, 73(12):1639–1648, 2013.

[62] Khronos OpenCL Working Group. The OpenCL Specification. Version 2.0, Nov 2013.

[63] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *Proc. 26th ACM Intl. Conf. on Supercomputing (ICS'12)*, pages 341–352, 2012.

[64] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157 – 174, 2012.

[65] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proc. 27th Intl. ACM Conf. on Supercomputing*, ICS '13, pages 149–160, New York, NY, USA, 2013. ACM.

[66] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979.

[67] O. Lawlor. Message passing for GPGPU clusters: CudaMPI. In *IEEE Intl. Conf. on Cluster Computing and Workshops (CLUSTER'09)*, pages 1–8, 2009.

[68] O. Lawlor. Embedding OpenCL in C++ for expressive GPU programming. In *Proc. 5th Intl. Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC 2011)*, May 2011.

[69] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proc. 2010 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2010.

[70] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[71] J. Lobeiras, M. Viñas, M. Amor, B. B. Fraguela, M. Arenaz, J. García, and M. Castro. Parallelization of shallow water simulations on current multi-threaded systems. *Intl. J. of High Performance Computing Applications*, 27(4):493–512, 2013.

[72] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proc. 42Nd Annual IEEE/ACM Intl. Symp. on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[73] T. Lutz, C. Fensch, and M. Cole. PARTANS: An Autotuning Framework for Stencil Computation on multi-GPU Systems. *ACM Trans. Archit. Code Optim.*, 9(4):59:1–59:24, Jan. 2013.

[74] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang. GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures. In *Proc. 41st Intl. Conf. on Parallel Processing (ICPP 2012)*, pages 48–57, Sept 2012.

[75] M. Majeed, U. Dastgeer, and C. Kessler. Cluster-SkePU: A multi-backend skeleton programming library for GPU clusters. In *Proc. Intl. Conf. on Parallel and Distr. Processing Techniques and Applications (PDPTA 2013)*, July 2013.

[76] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 896–907. ACM, 2003.

[77] T. McCabe. A complexity measure. *IEEE Trans. on Software Engineering*, 2:308–320, 1976.

[78] C. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z.-G. Wang, Z. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *9th Annual IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO 2011)*, pages 224–235, April 2011.

[79] T. Ngo. *The Role of Performance Models in Parallel Programming and Languages.* PhD thesis, Dept. of Computer Science and Engineering, University of Washington, 1997.

[80] Nvidia. *CUDA Compute Unified Device Architecture.* Nvidia, 2008.

[81] OpenACC-Standard.org. The OpenACC Application Programming Interface Version 2.0a, Aug 2013.

[82] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, Nov. 2015.

[83] R. Ozaydin and D. Altilar. OpenCL Remote: Extending OpenCL platform model to network scale. In *2012 IEEE 14th Intl. Conf. on High Performance Computing and Communication & 2012 IEEE 9th Intl. Conf. on Embedded Software and Systems (HPCC-ICESS)*, pages 830–835, 2012.

[84] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande. accULL: An OpenACC Implementation with CUDA and OpenCL Support. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 871–882, Berlin, Heidelberg, 2012. Springer-Verlag.

[85] R. J. Rost, B. M. Licea-Kane, D. Ginsburg, J. M. Kessenich, B. Lichtenbelt, H. Malan, and M. Weiblen. *OpenGL shading language.* Pearson Education, 2009.

[86] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL - a high level linear algebra library for GPUs and multi-core CPUs. In *Intl. Workshop on GPUs and Scientific Applications*, GPUScA, pages 51–56, 2010.

[87] F. P. Russell, M. R. Mellor, P. H. J. Kelly, and O. Beckmann. DESOLA: An active linear algebra library using delayed evaluation and runtime code generation. *Science of Computer Programming*, 76(4):227–242, 2011.

[88] S. Sengupta, M. Harris, Y. Zhang, and J. Owens. Scan primitives for gpu computing. In *Proc. 22nd ACM SIGGRAPH/EUROGRAPHICS symp. on Graphics hardware*, GH '07, pages 97–106, 2007.

[89] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *Proc. 2011 IEEE Intl. Symp. on Workload Characterization*, IISWC '11, pages 137–148, 2011.

[90] J. Shen, A. Varbanescu, Y. Lu, P. Zou, and H. Sips. Workload partitioning for accelerating applications on heterogeneous platforms. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2015.

[91] S. St-Laurent. *The complete effect and HLSL guide*. Paradoxal press, 2005.

[92] M. Steuwer and S. Gorlatch. SkelCL: a high-level extension of OpenCL for multi-GPU systems. *The Journal of Supercomputing*, 69(1):25–33, 2014.

[93] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - a portable skeleton library for high-level GPU programming. In *2011 IEEE Intl. Parallel and Distributed Processing Symp. Workshops and Phd Forum (IPDPSW)*, pages 1176 –1182, may 2011.

[94] M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl. CUDASA: Compute unified device and systems architecture. In *Eurographics Symp. on Parallel Graphics and Visualization (EGPGV 2008)*, pages 49–56, 2008.

[95] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *Computer*, 23(5):54–64, May 1990.

[96] P. Thoman, K. Kofler, H. Studt, J. Thomson, and T. Fahringer. Automatic OpenCL device characterization: Guiding optimized kernel design. In *Euro-Par'11*, volume 6853 of *LNCS*, pages 438–452. Springer-Verlag, 2011.

[97] D. E. Thomas and P. R. Moorby. *The Verilog® Hardware Description Language*, volume 2. Springer Science & Business Media, 2002.

[98] T. Veldhuizen. C++ Templates as Partial Evaluation. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, pages 13–18, Jan. 1999.

[99] M. Viñas, Z. Bozkus, and B. B. Fraguela. Exploiting heterogeneous parallelism with the Heterogeneous Programming Library. *J. of Parallel and Distributed Computing*, 73(12):1627–1638, Dec. 2013.

[100] M. Viñas, Z. Bozkus, B. B. Fraguela, D. Andrade, and R. Doallo. Developing adaptive multi-device applications with the Heterogeneous Programming Library. *The Journal of Supercomputing*, 71(6):2204–2220, 2015.

[101] M. Viñas, B. B. Fraguela, Z. Bozkus, and D. Andrade. Improving OpenCL programmability with the Heterogeneous Programming Library. In *Proc. Intl. Conf. on Computational Science (ICCS 2015)*, pages 110–119. Elsevier, 2015.

[102] M. Viñas, J. Lobeiras, B. B. Fraguela, M. Arenaz, M. Amor, and R. Doallo. Simulation of pollutant transport in shallow water on a CUDA architecture. In *2011 Intl. Conf. on High Performance Computing and Simulation (HPCS)*, pages 664 –670, july 2011.

[103] M. Viñas, J. Lobeiras, B. B. Fraguela, M. Arenaz, M. Amor, J. García, M. Castro, and R. Doallo. A multi-GPU shallow-water simulation with transport of contaminants. *Concurrency and Computation: Practice and Experience*, 25(8):1153–1169, June 2013.

[104] R. Weber and G. Peterson. Poster: Improved OpenCL programmability with clUtil. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion*, pages 1451–1451, 2012.

[105] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[106] S. Xiao and W.-C. Feng. Generalizing the utility of GPUs in large-scale heterogeneous computing systems. In *2012 IEEE 26th Intl. Parallel and Distributed Processing Symp. Workshops PhD Forum (IPDPSW)*, IPDPSW'12, pages 2554–2557, 2012.

Appendices

A. OpenCL Programming

The vast majority of the OpenCL applications follow the same steps to prepare the environment and to execute commands in the OpenCL devices. These steps as well as the OpenCL API functions in charge of them are briefly described in this appendix in order to illustrate the common OpenCL vocabulary and procedures. It deserves to be mentioned that most functions return an integer value that indicates whether the function finished correctly (CL_SUCCESS) or not. OpenCL provides a wide variety of error codes to allow the user to determine the reason of the failure. The most relevant error codes of each OpenCL API function are explained along this appendix.

1. **Platform discovery.**

   The first step in every OpenCL program consists in requesting information about the platforms installed in the system:

   ```
   cl_platform_id platforms;
   cl_uint num_platforms;

   //query for 1 available platform
   cl_int err = clGetPlatformIDs(
         2,                //number of platforms wanted
         &platforms,       //platform identifiers obtained
         &num_platforms);  //total number of platforms found
   ```

   The values returned in `err` can be:

   - **CL_SUCCESS**, which means that the call was completed successfully.

- **CL_INVALID_VALUE** if number of platforms wanted is 0 and `platforms` not NULL, or `num_platforms` and `platforms` are NULL.

Then, the device identifiers of the platforms chosen can be obtained using the function `clGetDeviceIDs`. Following with our example, we can obtain the identifier of a GPU of the platform selected previously with:

```
cl_device_id device_id;
cl_uint num_devices;

//query for 1 GPU
cl_int err = clGetDeviceIDs(
      platform_id,            //platform identifier previously selected
      CL_DEVICE_TYPE_GPU,     //device type wanted
      1,                      //number of devices wanted
      &device_id,             //device identifiers obtained
      &num_devices);          //total number of devices found
```

The values returned in `err` can be:

- **CL_SUCCESS**, which means that the call was completed successfully.
- **CL_INVALID_PLATFORM** if `platform_id` is not a valid platform.
- **CL_INVALID_DEVICE_TYPE** if the device type wanted is not a valid one (see Table 1).
- **CL_INVALID_VALUE** if the number of devices wanted is 0 and the device type is not NULL or if both variables are NULL.
- **CL_DEVICE_NOT_FOUND** if there are no matches for the device type wanted.

cl_device_type	**Description**
CL_DEVICE_TYPE_CPU	OpenCL capable CPUs
CL_DEVICE_TYPE_GPU	OpenCL capable GPUs
CL_DEVICE_TYPE_ACCELERATOR	Rest of capable devices such FPGAs, Xeon PHI, ...
CL_DEVICE_TYPE_DEFAULT	The default OpenCL device in the system
CL_DEVICE_TYPE_ALL	All OpenCL devices available in the system

Table 1: OpenCL device types.

2. **Creating contexts for the OpenCL devices.**

   Contexts are used by the OpenCL runtime to manage command queues, programs, kernels and to facilitate the sharing of buffers among the buffers associated with the same context. They are built using the function `clCreateContext` as follows:

```
cl_context context;
cl_context_properties cps[3] = {
      CL_CONTEXT_PLATFORM,                 // name of the property
      (cl_context_properties)(platform_id),// value of the property
      0 };                                 // must be terminated with 0

//query for 1 GPU
context = clCreateContext(
      cps,            // list with context properties
      1,              // num of devices that will be associated to this
                      // context
      &device_id)     // list with the devices
      NULL,           // Optional error callback function
      NULL,           // Argument of the callback function (if required)
      &err);          // return code
```

   The values returned in `err` can be:

   - **CL_SUCCESS**, which means that the context was created successfully.
   - **CL_INVALID_PLATFORM** if `cps` is NULL or the platform value is not a valid platform.
   - **CL_INVALID_VALUE** if context property name is not a supported one; if the list of devices is NULL; if the number of devices is 0 or callback function is NULL but its argument it is not.
   - **CL_DEVICE_NOT_AVAILABLE** if a device in the list is currently not available.
   - **CL_OUT_OF_HOST_MEMORY** if host program cannot allocate the memory space required by the context.

Once the context is created, command queues can be created to send commands to execute in the associated device:

```
cl_command_queue command_queue;

command_queue = clCreateCommandQueue(
      context,            // a valid context
      device_id,          // device that will be associated to
                          // this command queue
      0                   // command queue properties
      &err);              // return code
```

The command queue properties is a bit-field that can accept:

- `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`: if set, the commands are executed in an out-or-order fashion.
- `CL_QUEUE_PROFILING_ENABLE`: if set, the user can know profiling information of the commands.

The values returned in `err` can be:

- **CL_SUCCESS**, which means that the command queue was created successfully.
- **CL_INVALID_CONTEXT** if `context` is not a valid context.
- **CL_INVALID_DEVICE** if `device_id` is not a valid device or is not associated to `context`.
- **CL_INVALID_VALUE** if values of command queue properties are not valid.
- **CL_INVALID_QUEUE_PROPERTIES** if values of command queue properties are not supported.
- **CL_OUT_OF_HOST_MEMORY** if host program cannot allocate the memory space required by the command queue.

3. **Creating programs that will be executed in one or more devices.**

An OpenCL program is a collection of kernels written using OpenCL C. This program must be compiled by the OpenCL runtime compiler for its execution on a particular device. A kernel is a C-like function, C with little differences, which will be executed in parallel by each work-item of the global work index.

To compile the program containing the kernels, it is necessary to create a program object. This object encapsulates the sources of the program (online compiling) or a binary file (offline compiling), the executable after its compilation, the set of devices for which the program was compiled and the compilation options. If the source code of the program is available, which will be the most common case, its creation is done by means of the `clCreateProgramWithSource` function:

```
const char *kernel_code =
``__kernel void vectorAddition(__global int src, __global int dst) {
      dst[get_global_id(0)] += src[get_global_id(0)];
}'';

cl_program program;
program = clCreateProgramWithSource(
      context,                      // a valid context
      1,                            // Number of strings of the next
                                    // parameter
      (const char**) &kernel_code,  // Array of strings with the source
                                    // code
      NULL                          // length of each string or NULL
                                    // if they are null-terminated
      &err                          // return code
);
```

The values returned in `err` can be:

- **CL_SUCCESS**, which means that the program object was successfully created.
- **CL_INVALID_CONTEXT** if `context` is not a valid context.
- **CL_INVALID_VALUE** if `device_id` if the number of strings is zero or any string is NULL.

- **CL_OUT_OF_HOST_MEMORY** if host program cannot allocate the memory space required by the program object.

Once the program object is created, the building of its executable is the next step:

```
err = clBuildProgram(
      program,    // a valid program object
      1,          // number of devices in the device list
      NULL,       // list with the devices associated to the program
                  // object or NULL to select all of them.
      NULL,       // compiler options and preprocessor options
      NULL,       // callback function to notify the end of the routine,
                  // successfully or not.
      NULL        // argument for the callback function
);
```

This call modifies the program object to include the executable. There is a complete set of compiler options that can be specified and which can be classified as: math intrinsics, optimization and request/suppress warnings. This example defines the variable NUM_GPUs with value 2 and disables all the compiler optimizations: `char buildoptions = ‘‘-D NUM_GPUs=2 -cl-opt-disable’’`.

This function can return a wide variety of values. The most common ones are:

- **CL_SUCCESS**, which means that the program object was successfully created.
- **CL_INVALID_PROGRAM** if `context` is not a valid program object.
- **CL_INVALID_VALUE** if the list of devices is NULL and number of devices is not zero; the list is not NULL and the number of devices is 0; callback function is NULL but its argument not.
- **CL_INVALID_DEVICE** if the devices of the list are not associated to the program.
- **CL_INVALID_BUILD_OPTIONS** if the build options are invalid.
- **CL_BUILD_PROGRAM_FAILURE** if there is a failure to build the program executable.

In order to see the log of the compilation of a program object, users can use:

```
char buffer[4096];
size_t length;

err = clGetProgramBuildInfo(
      program,            // valid program object
      device_id,          // device whose executable was built
      CL_PROGRAM_BUILD_LOG, // information required.
      4096,               // maximum size of the buffer
      buffer              // buffer that will be filled with the
                          // output information.
      &length             // actual size in bytes copied to buffer.
);
```

4. **Choosing the kernels from the program object.**

   Kernel objects are a representation of each kernel function in the host program memory. They will be used to launch their execution through a command queue.

```
cl_kernel kernel;
kernel = clCreateKernel(
      program,           // a valid program already compiled
      ``vectorAddition'', // Name of the kernel
      &err               // return code
);
```

   The values returned in `err` can be:

   - **CL_SUCCESS**, which means that the kernel object was successfully created.
   - **CL_INVALID_PROGRAM_EXECUTABLE** if `program` was not successfully compiled.
   - **CL_INVALID_KERNEL_NAME** if the name of the kernel was not found in the program.

- **CL_INVALID_KERNEL_DEFINITION** if the signature of the kernel is not the same for all the devices.
- **CL_INVALID_VALUE** if the kernel name is NULL.
- **CL_OUT_OF_HOST_MEMORY** if host program cannot allocate the memory space required by the kernel object.

5. **Creating memory objects in the device.**

Memory objects are used to transfer data from the host and the device. They can be classified in two kinds: buffer objects and image objects. Buffer objects are more generic and used in heterogeneous computing. The image objects are designed for 2D and 3D images. Because of their larger independence of the domain, this brief introduction only includes the methods to manage buffer objects. OpenCL offers analogous methods to deal with image objects.

```
cl_mem input;
input = clCreateBuffer(
      context,            // a valid context
      CL_MEM_READ_ONLY,   // usage information (see Table 2)
      sizeof(float) * 1024,// size in bytes of the buffer (1024 floats)
      h_input,            // pointer to the data allocated in the host
      &err                // return code );
```

The values returned in `err` can be:

- **CL_SUCCESS**, which means that the buffer object was successfully created.
- **CL_INVALID_VALUE** if the flags of the bit-field are not valid.
- **CL_INVALID_BUFFER_SIZE** if size is 0 or greater than the maximum allowed for the device.
- **CL_INVALID_HOST_PTR** if the host pointer is NULL but CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR are set and vice versa.
- **CL_MEM_OBJECT_ALLOCATION_FAILURE** if there is another failure to allocate memory.

cl_mem flags	**Description**
CL_MEM_READ_WRITE	Buffer can be read or written in the device.
CL_MEM_WRITE_ONLY	Buffer can be only written in the device. Reads have an undefined behavior.
CL_MEM_READ_ONLY	Buffer can be only read in the device. Writes have an undefined behavior.
CL_MEM_USE_HOST_PTR	The host application wants the OpenCL implementation to use memory referenced by 4th argument.
CL_MEM_COPY_HOST_PTR	The host application wants the OpenCL implementation to use and copy memory referenced by 4th argument.
CL_MEM_ALLOC_HOST_PTR	The host application wants the OpenCL implementation to allocate memory from host accessible memory.

Table 2: OpenCL device types.

- **CL_OUT_OF_HOST_MEMORY** if host program cannot allocate the memory space required by the buffer object.

6. **Uploading data from the host to the device.**

After the buffer creation and before its use in a kernel execution, it is necessary to copy the data of the host program to the device memory in order to launch the kernel with the correct values of the data.

```
err = clEnqueueWriteBuffer(
      command_queue,         // a valid command queue
      d_buffer,              // memory buffer to write to
      CL_TRUE,               // indicate blocking write
      0,                     // offset in the buffer object (in bytes)
      sizeof(float) * 1024,  // size in bytes to be write
      h_buffer,              // pointer to buffer in host memory to
                             // copy data to
      0,                     // number of events in the event list
      NULL,                  // list of events to finish before this
                             // starts
      NULL                   // return code
);
```

The most common values returned in `err` can be:

- **CL_SUCCESS**, which means that command was successfully executed.
- **CL_INVALID_COMMAND_QUEUE** if `command_queue` is not valid.
- **CL_INVALID_CONTEXT** if the context associated to `command_queue` and the buffer object is not the same.
- **CL_INVALID_MEM_OBJECT** if `d_buffer` is not a valid buffer object.
- **CL_INVALID_VALUE** if the region being read is out of bound or host pointer is NULL.
- **CL_OUT_OF_HOST_MEMORY** if host program cannot allocate the memory space required by the OpenCL implementation on the host.

7. **Setting up arguments to the kernel.**

   The arguments that will be used on a kernel execution have to be indicated before the kernel execution starts.

```
err = clSetKernelArg(
      kernel,           // a valid kernel object
      0,                // index of the corresponding index
      sizeof(cl_mem),   // the size of the argument data
      &input_data,      // pointer to the data used as argument
);
```

   The index of the argument is the ordinal of its position in the kernel argument list, starting at zero. If the argument is a local memory variable, the pointer to the data is NULL.

   The most common values returned in `err` can be:

   - **CL_SUCCESS**, which means that the kernel object was successfully set.
   - **CL_INVALID_KERNEL** if `kernel` is not a valid object.
   - **CL_INVALID_ARG_INDEX** if index is not a valid one.
   - **CL_INVALID_ARG_VALUE** if the pointer is NULL and and the argument does not belong to the local memory space.
   - **CL_INVALID_MEM_OBJECT** if the argument is not a valid memory object.

- **CL_INVALID_ARG_SIZE** if the size of the argument does not match with the actual argument.

8. **Executing the kernel by means of the appropriate command queue.**

   After the global index space and the local index space are specified, the kernel execution can start after enqueueing the request to execute it in its corresponding queue:

```
err = clEnqueueNDRangeKernel(
      command_queue,     // a valid kernel object
      kernel,            // index of the corresponding index
      1,                 // the work problem dimensions
      NULL,              // offset in the global index
      &global,           // global index work
      NULL,              // local index work or NULL for let the decision
                         // to the OpenCL implementation
      0,                 // number of events in the event list
      NULL,              // list of events to finish before this starts
      NULL,              // event returned by this call
);
```

   The most common values returned in `err` can be:

   - **CL_SUCCESS**, which means that the kernel was executed successfully.
   - **CL_INVALID_PROGRAM_EXECUTABLE** if there is no successfully built program available for the device associated to `command_queue`.
   - **CL_INVALID_COMMAND_QUEUE** if `command_queue` is not a valid command queue.
   - **CL_INVALID_KERNEL** if `kernel` is not a valid kernel object.
   - **CL_INVALID_CONTEXT** if the context associated with `kernel` and `command_queue` is not the same.
   - **CL_INVALID_KERNEL_ARGS** if kernel arguments have not been specified.
   - **CL_INVALID_WORK_DIMENSION** if the dimension is not between 1 and 3.

- **CL_INVALID_WORK_GROUP_SIZE** if global work size is not evenly divisible by local work size of the value specified is not valid for the selected device.

9. **Offloading data from the device to the host.**

   In order to allow the host program to access the results computed in the device, these data must be copied from the device to the host using the command:

```
err = clEnqueueReadBuffer(
      command_queue,         // a valid command queue
      d_buffer,              // memory buffer to read from
      CL_TRUE,               // indicate blocking write
      0,                     // offset in the buffer object (in bytes)
      sizeof(float) * 1024,  // size in bytes to be write
      h_buffer,              // pointer to buffer in host memory to
                             // copy data to
      0,                     // number of events in the event list
      NULL,                  // list of events to finish before this
                             //starts
      NULL                   // return code
);
```

   The most common values returned in `err` can be:

   - **CL_SUCCESS**, which means that command was successfully executed.
   - **CL_INVALID_COMMAND_QUEUE** if `command_queue` is not valid.
   - **CL_INVALID_CONTEXT** if the context associated to `command_queue` and the buffer object is not the same.
   - **CL_INVALID_MEM_OBJECT** if `d_buffer` is not a valid buffer object.
   - **CL_INVALID_VALUE** if the region being read is out of bound or host pointer is NULL.
   - **CL_OUT_OF_HOST_MEMORY** if host program cannot allocate the memory space required by the OpenCL implementation on the host.

The OpenCL API contains many more procedures not covered in this brief manual. Among others, it includes: those in charge of the release of memory, kernel and program objects, the mechanisms to request information of the platforms, devices, etc. or the different ways to synchronize the commands launched. In the OpenCL Programming Guide, which is available in the Khronos Group website, the API is explained in detail.