UNIVERSIDAD DE CANTABRIA

Departamento de Ingeniería Informática y Electrónica

Programa de Doctorado en Ciencia y Tecnología

Tesis Doctoral

Soporte para co-ejecución eficiente en sistemas heterogéneos

PhD Thesis

Efficient co-execution support in heterogeneous systems

Borja Pérez Pavón

Dirigida por Jose Luis Bosque Orero

Escuela de Doctorado de la Universidad de Cantabria

Santander 2019

Agradecimientos

Este documento representa el trabajo de 5 años, fruto del esfuerzo y apoyo de muchas más personas de las que podrían aparecer en la portada. Las siguientes líneas son para reconocer su importancia en esta tesis.

- A Jose Luis, mi director de tesis. Gracias por tus esfuerzos, por haber confiado en mi y por haberme infundido el gusto por la investigación e, incluso, por la docencia.

- A todos los miembros del grupo de investigación de Arquitectura y Tecnología de Computadores. Vuestro trabajo diario hace posible que sigamos investigando. Especialmente a Esteban, por estar siempre dispuesto a echarme una mano, y a Mon, por su apoyo y sus lecciones que van más allá de lo académico.

- A mis padres, abuelas y abuelos. Este camino empezó hace mucho tiempo y vuestra ilusión, cariño y apoyo constantes son los que han permitido que llegue a la meta.

- A mis amigos de siempre. Porque, pese a que nuestras vidas nos alejen, el tiempo no pasa, y se que siempre estaréis ahí.

- A mis amigos nuevos, en sus encarnaciones musicales y carpinteras. Porque el verdadero cariño se expresa en la descalificación continua. Si esta tesis es prácticamente un tema de hip-hop es por vosotros.

- A ti, porque aunque algunas cosas cambien, las más importantes siguen igual.

Abstract

Heterogeneous architectures have become prevalent due to their outstanding performance and energy efficiency. However, programming these systems is far from trivial. Current heterogeneous programming models impose a host-device approach and favour task-parallelism. Applications developed following these kind of models are executed in the CPU and only certain compute intensive parts are explicitly offloaded to accelerators. On the contrary, co-execution, the cooperative operation of all the available devices computing for a single workload in a data-parallel manner, is overlooked. If desired, the programmer will be in charge of manually managing the devices, the data distribution and the division of the workload, effectively expressing co-execution in terms of task-parallelism and representing a significant amount of work.

For co-execution to be useful it has to be effortless. This means that using every device in the computation of a single kernel should represent the same work as using a single device. To achieve this, the programmer needs to be abstracted from the underlying hardware and spared load balancing decisions. The former enables a transparent use of the whole system that eases programming and guarantees portability, as the programmer no longer has to worry about the devices. The latter aids the programmer to obtain good performance from co-execution, as these decisions are complex, depending on both the irregularity of the co-executed workload and the heterogeneity of the system itself.

This dissertation makes several contributions towards effortless co-execution in heterogeneous systems, tackling abstraction and load balancing from both the software and hardware angles. HGuided and Sigmoid, two novel load balancing algorithms specially designed for co-execution, are proposed and evaluated, achieving outstanding performance. Maat, a new OpenCL-based load balancing library has been designed and implemented. It enables the abstract management of the whole system by providing the illusion of a single device representing all the available resources. Taking abstraction even further, co-execution and load balancing have also been implemented in OmpSs, as an evaluation of the interest of co-execution in task-based programming models. Lastly, a design for a dispatcher that enables hardware supported co-execution in integrated heterogeneous systems is presented and evaluated. These contributions ease the programming of heterogeneous systems and represent a significant improvement on both performance and energy efficiency.

Resumen

Gracias a su rendimiento y eficiencia energética, las arquitecturas heterogéneas se han convertido en un elemento común en cualquier sistema de cómputo. Desde el supercomputador hasta los dispositivos móviles, todos los sistemas actuales integran CPUs tradicionales y aceleradores especializados en ciertas cargas de trabajo. La aparición de este tipo de arquitecturas también supuso el desarrollo de nuevos modelos de programación que permitieran aprovechar sus capacidades. Estos modelos parten de que una aplicación heterogénea se ejecutará principalmente en la CPU, mientras que ciertas funciones especialmente intensivas en cómputo y masivamente paralelas en datos, serán descargadas en los aceleradores. Esta descarga debe ser realizada explícitamente por el programador, el cual es responsable de gestionar manualmente cada uno de los dispositivos disponibles, las tareas que ejecutarán y los datos que utilizarán. A esta forma de operar se le suele llamar modelo *host-device* y favorece el paralelismo de tareas, pero también desaprovecha la capacidad de cómputo y la energía de la CPU cuando se ejecuta sólo en la GPU. Sin embargo, otra forma de paralelismo también es posible: la co-ejecución.

La co-ejecución se define como la cooperación de todos los dispositivos disponibles en el sistema, ejecutando la carga de trabajo asociada a una sola tarea y aprovechando el paralelismo de datos. Esto quiere decir que todos los dispositivos ejecutarán el mismo código, pero producirán porciones de los resultados disjuntas. Esta aproximación al paralelismo, además, se ajusta a las características de las aplicaciones que habitualmente se descargan en aceleradores, pues estos basan sus capacidades en el paralelismo de datos. Sin embargo, los modelos de programación actuales no tienen ningún soporte para co-ejecución. Si se desea que los dispositivos cooperen en la ejecución de una única tarea, el programador tendrá que repartirla manualmente entre los dispositivos, gestionándolos individualmente y transfiriendo los datos que necesiten. En definitiva, los modelos de programación obligan a expresar la co-ejecución utilizando paralelismo de tareas.

Para que la co-ejecución sea una opción verdaderamente útil, los modelos de programación tienen que ofrecer las capacidades necesarias para que los dispositivos cooperen sin precisar trabajo adicional por parte del programador. Una co-ejecución que no requiera esfuerzos se construye sobre dos pilares: la abstracción y el equilibrio de carga.

La abstracción se refiere a la interacción del programador con el sistema, que debe realizarse a

través de interfaces de alto nivel, que eviten que el programador tenga que preocuparse de la gestión individual de los dispositivos. Esto no sólo facilita la co-ejecución, sino que garantiza la portabilidad de las aplicaciones, pues al abstraer la programación de los dispositivos subyacentes, un código desarrollado en un sistema usará todos los dispositivos disponibles en cualquier otro.

El equilibrio de carga está relacionado con la división de la carga de trabajo entre los dispositivos disponibles, con lo cual es fundamental para obtener un buen rendimiento. En general, las decisiones de equilibrio de carga son complejas, pues dependen tanto del comportamiento de la carga a distribuir como de los propios dispositivos. La carga de trabajo puede tener un comportamiento variable o irregular a lo largo de su ejecución, con tiempos de ejecución diferentes para porciones de trabajo con el mismo tamaño a priori. Este tipo de cargas requieren que los algoritmos puedan adaptarse para obtener buenos rendimientos. Sin embargo, esta capacidad de adaptación no representa más que una sobrecarga para las cargas regulares que no la requieren. Por otro lado, los sistemas heterogéneos incluyen dispositivos con arquitecturas y capacidades de cómputo muy diferentes. Por tanto, el balanceo de carga tendrá que tener en cuenta las características del dispositivo que recibirá cada porción de trabajo. En definitiva, este tipo de decisiones deben facilitarse al programador lo más posible. Idealmente, la co-ejecución debe ir acompañada de algoritmos de balanceo de carga que obtengan buen rendimiento sin requerir ninguna información al programador. Estos son los algoritmos no informados, en contraposición con los que sí que la requieren, llamados informados.

Esta tesis presenta una serie de contribuciones que posibilitan la co-ejecución eficiente y sin esfuerzo en sistemas heterogéneos, abordando la abstracción y el equilibrio de carga, tanto desde un punto de vista software como hardware. En otras palabras, el objetivo de la tesis es ofrecer los medios necesarios para que ejecutar una tarea colaborativamente, aprovechando todos los dispositivos del sistema, represente el mismo trabajo para el programador que utilizar uno sólo de los dispositivos. Para desarrollar esta investigación, se ha recurrido tanto a ejecuciones reales, como a simulaciones utilizando gem5, la herramienta de simulación estándar de facto en la investigación en arquitectura de computadores. Para realizar las evaluaciones, se han utilizado los sistemas de cómputo del grupo de investigación de Arquitectura y Tecnología de Computadores (ATC) de la Universidad de Cantabria, así como una máquina del grupo TRASGO de la Universidad de Valladolid para realizar pruebas de escalabilidad.

En primer lugar, se proponen HGuided y Sigmoid, dos nuevos algoritmos de balanceo de carga especialmente diseñados para co-ejecución en sistemas heterogéneos. HGuided es un algoritmo dinámico que utiliza tamaños de paquete decrecientes, de modo que se reducen las sobrecargas sin perder capacidad de adaptación cerca del final de la ejecución, que es cuando realmente es necesaria. Este algoritmo también considera la velocidad de cómputo de cada dispositivo, de modo que no se asignen paquetes demasiado grandes a dispositivos lentos que generen desequilibrios. De esta manera, HGuided consigue equilibrar satisfactoriamente la carga tanto de aplicaciones regulares como irregulares, obteniendo un rendimiento y eficiencia energética excelentes. Sin embargo, este algoritmo es informado, pues recibe la velocidad de cómputo y el tamaño mínimo hasta el que menguarán los paquetes como parámetros. Esto requiere un esfuerzo al programador, pues los valores óptimos para los parámetros varían entre sistemas y aplicaciones. Por el contrario, Sigmoid es un algoritmo no informado, capaz de monitorizar la co-ejecución y de adaptar sus parámetros internos al comportamiento de las cargas de trabajo. Internamente, utiliza una función derivada de la sigmoide para controlar el tamaño de los paquetes que genera, de modo que tengan tamaños menores si se necesita mayor adaptabilidad. El resultado es un rendimiento y una eficiencia excelentes, sin requerir ningún esfuerzo por parte del programador.

Respecto a la abstracción, esta tesis presenta Maat, una librería de co-ejecución en sistemas heterogéneos basada en OpenCL. Maat simplifica la co-ejecución significativamente, creando la ilusión de que el sistema dispone de un único dispositivo que representa la capacidad de cómputo agregada de todos los recursos disponibles. Esta ilusión se construye a través de abstracciones implementadas respetando la filosofía de programación de OpenCL. De esta manera, se facilita la adaptación de aplicaciones pre-existentes para que aprovechen la co-ejecución. Maat también mejora la portabilidad de las aplicaciones, pues una gestión abstracta de los dispositivos permite que una aplicación programada para un sistema funcione en cualquier otro sin necesidad de modificaciones. Los resultados experimentales demuestran que Maat obtiene rendimientos cercanos a los ideales y ahorros energéticos importantes para las aplicaciones evaluadas, siendo Sigmoid el algoritmo de equilibrio de carga que mejores resultados obtiene, seguido de cerca por HGuided.

Para llevar la abstracción un paso más allá, esta tesis también propone la extensión de modelos de programación basados en tareas para soportar co-ejecución. Este tipo de modelos facilitan el

desarrollo de aplicaciones ofreciendo abstracciones de alto nivel que ocultan detalles complejos de la programación paralela. Sin embargo, tienen carencias importantes en lo que se refiere a co-ejecución. Como ejemplo para evaluar su utilidad, se ha añadido un nuevo módulo a OmpSs, pero las conclusiones extraídas pueden hacerse extensibles a otros lenguajes. Este nuevo módulo hace posible la co-ejecución de una sola tarea utilizando todos los recursos disponibles en el sistema sin requerir ningún esfuerzo del programador. Además, esta extensión apenas tiene impacto en la infraestructura original de OmpSs o en la forma de programar aplicaciones utilizándolo. De forma similar, los algoritmos implementados en el módulo respetan las ideas sobre equilibrio de carga propuestas por OpenMP, que es la base de OmpSs. Se proponen cuatro algoritmos. El algoritmo estático genera un único paquete por dispositivo, con un tamaño proporcional a la velocidad de cómputo de este en relación con la de todo el sistema. El algoritmo dinámico, por el contrario, divide la carga en gran cantidad de paquetes pequeños que se planifican en tiempo de ejecución. HGuided funciona de manera similar, pero utiliza tamaños de paquete decrecientes. Por último, Auto-Tune, es una evolución de HGuided capaz de monitorizar la co-ejecución, de modo que elimina la necesidad de parámetros, siendo un algoritmo no informado. Los resultados corroboran que la co-ejecución es beneficiosa para los modelos de programación basados en tareas, pues facilita el aprovechamiento de todos los recursos disponibles en el sistema. El algoritmo Auto-Tune consigue el mejor rendimiento medio, ya que es capaz de adaptarse a las aplicaciones irregulares y de conseguir un rendimiento casi igual al de Static en las regulares.

Esta tesis también presenta y evalúa un nuevo dispatcher para dar soporte hardware a la co-ejecución en sistemas heterogéneos integrados. Este tipo de arquitecturas aúnan en un solo SoC cores CPU y unidades de cómputo GPU, por lo que reducen las latencias y sobrecargas relativas a la interacción entre los dispositivos. Sin embargo, a pesar de la integración, la CPU y la GPU siguen tratándose como dispositivos individuales. Estos sistemas permitirían considerar diseños en los que el programador simplemente lanzara una tarea y fuera el hardware el que se encargara de distribuirla entre los dispositivos de forma eficiente. Así es como opera el nuevo dispatcher: reparte de forma autónoma work-groups OpenCL entre cores de la CPU y compute units de la GPU indistintamente. Además, el dispatcher monitoriza el tiempo que tardan en ejecutarse los work-groups en cada dispositivo. De esta manera, es capaz de desactivar temporalmente la co-ejecución si predice que la asignación de un work-group a un dispositivo supondrá un retraso. Esta propuesta ha sido evaluada utilizando el simulador gem5. Los resultados muestran que el

nuevo dispatcher permite obtener mejoras de rendimiento importantes. Sin embargo, también genera nuevos retos que pueden limitar la ganancia de rendimiento en ciertos casos, como contención en el acceso a memoria.

Finalmente, una tesis no sólo representa una respuesta a una pregunta, sino también el comienzo de nuevas líneas de investigación que se construyen sobre el trabajo realizado a lo largo del doctorado. En primer lugar, los algoritmos de equilibrio de carga constituyen un área de investigación de gran interés. Los aquí propuestos se centran en el rendimiento. Sin embargo, aunque el rendimiento y el consumo energético estén muy relacionados, algunos resultados de esta tesis muestran que una distribución de carga que maximiza el rendimiento no necesariamente minimiza el consumo. Por este motivo, podría pensarse en algoritmos que tuvieran por objetivo minimizar la energía. Por otro lado, los algoritmos propuestos son agnósticos: operan de forma idéntica para todas las cargas de trabajo. Sería interesante investigar algoritmos que fueran capaces de analizar el código de la carga a co-ejecutar, de modo que pudieran tomar decisiones de equilibrio de carga más informadas. Podrían incluso decidir enviar toda la carga a un único dispositivo, liberando al programador de la decisión de cuándo co-ejecutar. Finalmente, desde el hardware, podría investigarse en técnicas para reducir la contención en el acceso a memoria producida por la co-ejecución, o en diseños de dispatchers más inteligentes, capaces de identificar patrones de cómputo irregulares.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computing systems are the main driving force of science and technology. The need for greater accuracy, more complex models and greater volumes of data, are the sources for ever growing computing requirements. In order to match these needs, computer architects have to overcome technology limitations that challenge the way in which systems are designed. First, it was Dennard scaling, which gave rise to multicore architectures. Now it is Moore's law, which has led to dispute the notion of architectures formed by identical cores. As a result, systems now accommodate devices specialized in accelerating certain workloads, such as GPUs, FPGAs or TPUs, that achieve outstanding performance and energy efficiency. This excellent capabilities have made heterogeneous systems prevalent in the computing world, ranging from supercomputing to mobile devices. However, managing the different devices of a heterogeneous system also poses certain new challenges that need to be addressed. One such is the programming itself, which is far from trivial, requiring significant expertise if all the potential computing capabilities of the system are to be leveraged. Furthermore, these systems combine devices with very different architectures and computing capabilities. This complicates the extraction of their performance and its portability.

1.1. Programming heterogeneous systems

The development of code for heterogeneous systems currently relies on *host-device* programming models. Applications following this approach will run on the CPU, and only certain compute intensive parts will be offloaded to the accelerators. Offloads have to be explicitly performed, so the programmer has to decide what to offload and when to do it. Moreover, the management of the data required by the accelerators has to be explicitly handled too, as devices often have their own memory spaces. The result is that the programmer is responsible for managing the whole heterogeneous system and extracting its potential performance, which is a complex task. For this reason, heterogeneous applications are often tailored to a particular system, requiring modifications to work on other systems and compromising portability.

Matters get even worse as systems grow. The more accelerators available, the more orchestration that is necessary, so fully using all the available computing capabilities is ever more challenging and hardly portable. Furthermore, failing to use all the devices represents a waste of energy, as idle devices still consume. So the question is: how can a programmer efficiently use a modern system, such as the DGX SaturnV, which holds 8 GPUs per node? One answer is offloading different workloads to each GPU. This is known as task parallelism, and it is limited by the number of independent tasks that can be extracted from an application. The other option is co-execution.

1.2. Co-execution

As opposed to task parallelism, co-execution is based on the cooperation of all the available devices, working on the same problem in a data-parallel fashion. This means that, when a task is launched, it is split, so each device computes for a portion of the total workload. Consequently, each device will produce a disjoint portion of the results. This approach lends itself specially well to heterogeneous environments, as accelerators often base their success on data-parallelism. Co-execution enables the extraction of all the potential performance of the system and also favours its efficient use, specially regarding energy consumption, as all the

resources are contributing useful work instead of idling. Additionally, it does not force the programmer to generate parallelism through tasks, which sometimes are not easy to extract, but leverages the parallelism shown by data itself.

Nevertheless, there is an inconvenient. Current programming models favour task parallelism. This means that if co-execution is desired, the programmer will have to handle it himself. The management required includes manually splitting the task, handling the input and output data for each device and explicitly performing the offloads. In short, the programming model requires to express co-execution in terms of task-parallelism. This effectively turns heterogeneous co-execution into a second class citizen, requiring a significant amount of work to implement it.

For heterogeneous co-execution to be a useful option, it needs to be effortless. This is, the execution of a task using a single device should represent a similar effort to its efficient co-execution using all of them. This is, for example, the operation of the OpenMP `parallel for` clause, which enables the easy distribution of the iterations associated to a for loop to the available threads in a CPU. Effortless co-execution is based on two pillars: abstraction and load balancing.

1.2.1. Abstraction

To minimize the effort required for co-execution, the programmer should operate the same way regardless of the system that will execute the application. In other words, he should not have to be aware of the underlying architecture of the heterogeneous system. This is, all the tasks that directly depend on the devices should be internally handled. As a result, the programmer will no longer have to deal with manual offloads or data distribution to each of the individual devices. This not only eases programming and makes co-execution more accessible, but also favours portability, as an application programmed for a system will work in a different one unmodified, regardless of the number and kind of devices it holds.

1.2.2. Load balancing

The key to extracting all the performance a system has to offer through co-execution, is adequately distributing the workload among all the available devices. In general, the goal is to come up with a work partition that generates equal execution times in all the devices of the system, so they all finish simultaneously and idle times are minimized. In a heterogeneous environment, two aspects must be considered to obtain a balanced distribution: *irregularity* and *heterogeneity*. The former is related to the behavior of the workload, while the latter involves the devices.

Irregularity refers to the behavior of the co-executed workload throughout its runtime. Certain workloads, called *regular*, show equal execution times for equally sized portions of work. On the contrary, *irregular* workloads have varying execution times for chunks of work of the same size. For instance, a sparse matrix-vector product will be irregular, as each row may have a different number of non-zero elements, while a dense matrix-vector product will be regular. Consequently, irregular workloads are more challenging to balance, requiring a certain degree of adaptiveness to react to their changing behavior. However, adaptiveness comes at the price of overheads, which may harm the performance of regular workloads that do not require it.

Heterogeneity is related to the devices that take part in the co-execution. Heterogeneous systems include devices with very different architectures and computing capabilities. These should be taken into consideration to avoid scheduling too much work to a slow device, which might delay the completion of the whole workload. Furthermore, certain architectures require to be scheduled more work than others to fully use their resources, extracting all their potential performance.

Due to these two factors, load balancing decisions are complex, as they depend on the workload and the co-executing hardware. As a consequence, they should not be left to the programmer. A useful co-execution scheme should aid the programmer by offering load balancing algorithms capable of producing balanced work distributions. To make co-execution even easier, these algorithms should also require no input from the programmer. These are *uninformed* algorithms,

as opposed to *informed* ones, which require to be provided with certain parameters to operate.

1.3. Hypothesis

In light of the above, the aim of the work presented in this dissertation will be evaluating the following hypothesis:

Co-execution can improve the performance and energy efficiency of heterogeneous systems. However, to achieve this, it is necessary to solve two main problems: abstraction and load balancing. To prove this, both hardware and software techniques, including different programming models, will be evaluated. The final goal is to ease the programming of heterogeneous systems, providing support for effortless co-execution that extracts all the performance available. This guarantees that the programming work required to efficiently execute a workload using all the available devices is equivalent to that of using just one.

1.4. Major Dissertation Contributions

The most prominent contributions of the dissertation are listed next. Each of them will be explained in detail in a separate chapter of this document.

- Designing two novel load balancing algorithms: HGuided and Sigmoid are two new load balancing algorithms that specially target heterogeneous systems. HGuided is an informed algorithm that regards the computing speed of the devices and uses a diminishing package size for greater adaptiveness near the end of the co-execution. Sigmoid is an uninformed algorithm that monitors co-execution to adapt to the behavior of the workload. It uses the well-known sigmoid function to control the size of the packages it generates. Both algorithms obtain outstanding results regarding performance and energy efficiency.
- Proposing a heterogeneous co-execution library: Maat is an OpenCL library that achieves effortless co-execution by keeping all the management required by OpenCL under the hood

and implementing HGuided and Sigmoid. It offers high level abstractions that provide the illusion of dealing with a single OpenCL device that represents the aggregated computing capabilities of all the devices of the system. This eases the adaptation of pre-existing OpenCL applications to co-execution. Maat has been implemented with performance in mind. It has low overheads and leverages communication-computation overlapping when possible.

- Extending a task-based programming model with co-execution: OmpSs has been extended to support co-execution as a sample task-based programming model. This enables heterogeneous co-execution with an even higher level of abstraction, closer to sequential programming and not requiring to manage the heterogeneous environment at all. The new functionality has been implemented with minimum impact to OmpSs and preserving its programming philosophy, with load balancing algorithms inspired by OpenMP.

- Designing a dispatcher for hardware supported heterogeneous co-execution: A new hardware dispatcher for integrated heterogeneous systems has been proposed. It considers both CPU cores and GPU compute units in the scheduling of the OpenCL work-groups associated to a kernel launch. Consequently, it enables the programmer to enqueue a kernel to the heterogeneous system and let the hardware handle co-execution. The dispatcher is also capable of throttling co-execution to avoid delays if one of the devices is found to be too slow. The evaluation of the proposed design has been performed using the gem5 simulator.

1.5. Methodology

The results presented in this dissertation are based on experimental data obtained from real executions and simulation. This section introduces the platforms used throughout the evaluation, the selected metrics and the tools used to measure the energy consumption. Apart from the generic information provided in this section, later chapters may have their own methodology sections, devoted to the particular details of their experimentation.

1.5.1. Test platforms

Two different platforms have been used for the experimentation. A system called *Batel*, belonging to Universidad de Cantabria, has been used to carry out most of the performance and energy evaluation. To analyze the scalability of the proposed load balancing algorithms, *Hydra* has been used due to its greater device count, which belongs to Universidad de Valladolid. Details on the specific configuration of each of the platforms are provided next.

Batel Batel has two CPUs, two GPUs and 16 GBs of DDR3 memory. The CPUs are Intel Xeon E5-2620, with six cores that can run two threads each at 2.0 GHz. The CPUs are connected via QPI, which allows OpenCL to detect them as a single device. Therefore, throughout the remainder of this document, any reference to the CPU includes both Xeon E5-2620 processors. The GPUs are NVIDIA Kepler K20m with 13 SIMD lanes (or SMs in NVIDIA terminology) and 5 GBytes of VRAM each [NVI12]. These are connected to the system using independent PCI 2.0 slots.

Hydra Hydra holds four NVIDIA GeForce GTX TITAN Black GPUs, each one having 15 SIMD lanes and 6GB of VRAM. Its greater number of devices will be leveraged to evaluate the scalability of the proposed co-execution techniques.

1.5.2. Evaluated metrics

The contributions presented in this dissertation have been evaluated regarding their performance and energy consumption. The metrics considered in each case are explained next.

Performance

Performance has been evaluated using the response time of the selected benchmarks. This includes the time required by the communication between the host and the devices, comprising input and output data transfers, as well as the execution time of the co-executed workload itself.

The benchmarks are executed in two scenarios, the *heterogeneous system*, taking advantage of the GPUs and CPU, and the *baseline*, that only uses one GPU.

Based on these response times, two metrics are analyzed. The first is the speedup for each benchmark when comparing the baseline and the heterogeneous system response times. Note that, for the employed benchmarks, the CPU is much slower than the GPUs. Then, the maximum achievable speedup using n devices will not be n, but a fraction over the number of available GPUs that depends on the computing speed of the CPU for the application. The speedup for each application using a perfectly balanced distribution has also been used to give an idea of advantage of using the complete system. They were derived from the response time T_i of each device as shown in Equation 1.1.

$$S_{max} = \frac{1}{max_{i=1}^{n}\{T_i\}} \sum_{i=1}^{n} T_i \tag{1.1}$$

The second metric is the load balancing efficiency, obtained by dividing the obtained speedup by the speedup for the perfectly balanced distribution. The obtained value ranges between 0 and 1, giving an idea of the usage of the heterogeneous system. Efficiencies close to 1 indicate the best usage of the system is being made. The measured values do not reach this ideal because of the communication overheads and host-device interactions.

Energy

Regarding energy, the same two scenarios are considered as in the performance evaluation. However, in the baseline scenario the energy consumed by the idle devices is also taken into account. This is a fair comparison, as idle devices still consume static energy. Moreover, using an accelerator while the rest of the system is idle is a typical scenario favoured by host-device programming. Apart form the total consumed energy, the Energy Delay Product (EDP) [CCBB15] has also been used. It is a metric that combines performance and energy to evaluate the efficiency of the system. The tool used to measure the energy of the devices will be introduced next.

1.5.3. Energy measurements

To measure the energy consumption of the system it is necessary to take into account the power drawn by each device. Modern computing devices allow applications to monitor their functionality and performance. However, the power measured is associated to the device and not the kernel or process in execution. Together with the fact that it is impractical to add measurement code to all the test applications, this led to the development of a power monitoring tool named *Sauna*. It takes a program as its parameter, and is able to periodically query all the devices for power measurements throughout the execution of the program.

A significant amount of thought went into the conception of Sauna; the fact that it had to monitor several devices meant that it had to adapt to the particularities of each one while giving consistent and homogeneous output data. This started with the different APIs provided to perform these measurements. For the Intel CPUs, recent versions of the Linux kernel provide access to the *Running Average Power Limit (RAPL)* registers [RNR+11], which provide accumulative energy readings. On contrast, NVIDIA provides the *NVIDIA Management Library (NVML)* [NVI18] that gives instant power measurements. Naturally, Sauna had to be able to convert between the two magnitudes. A particularly interesting aspect of the development process of Sauna was studying the impact of the sampling frequency. In order to keep the program simple, it was necessary to use a single sampling period for all devices. Given that the power variations would be similar across devices, the idea seemed feasible.

To find the best frequency, a series of experiments were made for each device in *Batel* (Section 1.5.1). It was observed that each device reacted differently to the sampling frequency. The RAPL measurements grew with large frequencies. And more surprisingly, the NVIDIA devices slowed down noticeably when the sampling frequency was above a given threshold. This actually meant that the kernel running in the device took longer to complete. Figure 1.1 shows these effects as the magnitude of the variation of the measured CPU power depending on the sampling rate, together with the execution time of a Binomial kernel on a NVIDIA GPU. These graphs suggest adopting a low frequency, however, if the sampling period is too long, fast power spikes that may appear under irregular loads could be missed, leading to inexact results. As a

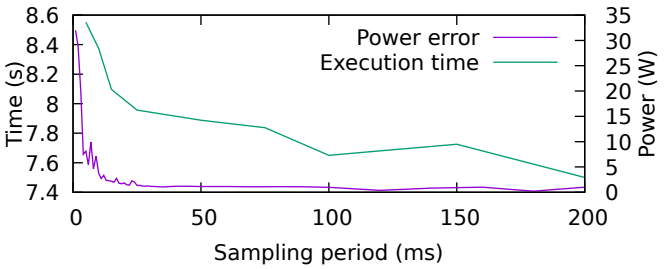


Figure 1.1: Impact of sampling period on power measurement and kernel execution time.

consequence it was decided to use 45ms as the sampling period for all the remaining experiments, as it balances overhead and accuracy.

1.6. Document Structure

This dissertation is organized as follows:

Immediately following this introduction, Chapter 2 provides some background on the programming of heterogeneous systems, including OpenCL and OmpSs, and introduces the gem5 simulator. It also analyses the most notable related work on co-execution, heterogeneous system abstraction and load balancing.

Chapter 3 defines the load balancing problem formally and presents two novel load balancing algorithms designed for heterogeneous co-execution. HGuided stems from the need to reduce the overheads of dynamic load balancing techniques while preserving adaptiveness. The algorithm is first explained to then elaborate on its limitations. Sigmoid builds on the foundation of HGuided. It is an uninformed algorithm that monitors co-execution at runtime. The reasoning to obtain the workload division function of Sigmoid is first introduced to then detail how its internal parameters are tuned.

Chapter 4 presents Maat, an OpenCL library for effortless heterogeneous co-execution that provides the programmer with the illusion of handling a single device that represents the aggregate computing power of the whole system. An overview of Maat is provided first, to then dive into the design of the abstractions that it enables. These strive to preserve the OpenCL approach to parallel programming to ease the use of the library. The implementation of Maat is

explained next, to then evaluate the performance, energy and scalability of the load balancing algorithms implemented in Maat, which are HGuided and Sigmoid.

Chapter 5 introduces an extension to OmpSs, as a sample task-based programming model, to support heterogeneous co-execution. First, the chapter elaborates on the best design approach to include co-execution in the OmpSs infrastructure. Then, the proposed load balancing techniques for co-execution in task-based programming models are introduced which, to accommodate to the philosophy of OmpSs, resemble the ones of OpenMP. How co-execution was implemented with minimum impact to OmpSs is explained next to finally evaluate the proposal in terms of performance and energy.

Chapter 6 proposes a new dispatcher design for co-execution in integrated heterogeneous systems. The chapter first discusses on the best way to support hardware aided co-execution in integrated CPU-GPU architectures, both from software and hardware. It then proposes a design with low impact on both OpenCL and the system architecture. How the dispatcher throttles co-execution to prevent delays from slow devices is explained next, to then dive into how the proposed design was implemented in the gem5 simulator. Finally, the proposed dispatcher is evaluated in terms of performance, as gem5 currently does not model the energy consumption of the GPU.

Lastly, the dissertation ends with Chapter 7, which presents some conclusions and future lines of work.

Chapter 2

Background and Related Work

Over the past few years, the development of heterogeneous platforms has been followed by an effort, both from academia an industry, to produce the necessary tools to efficiently use this kind of systems. This includes both the required software support to be able to program heterogeneous devices and the simulation environments to model their architecture and evaluate their capabilities. The purpose of this chapter is twofold: first, it introduces the key technologies and tools that the thesis has been built upon, and second, it presents some works that are related to the research field of co-execution.

2.1. Programming models

The main challenge of programming heterogeneous systems stems from their strength: the collaboration of several architectures to solve a problem may be highly advantageous, but the careful orchestration and management of very different devices is a problem to be solved in order to reap the benefits of heterogeneity. A very common approach is to deem the CPU as the host, while the rest of the devices are considered accelerators that will run certain offloaded parts of the applications. This concept, often referred to as "Host-Device Programming Model", lies at the core of the two most prominent heterogeneous programming frameworks: CUDA [KH10a] and OpenCL [GHK+11]. Both frameworks share many features and have a very similar

approach to heterogeneous systems. However, the former is a NVIDIA proprietary technology, so it is only supported by NVIDIA GPUs. Moreover, CUDA does not support the execution of a kernel in other kinds of devices, such as CPUs, which forces to develop and maintain several versions of the same program. For these reasons and considering that one of the goals of the thesis is being able to adapt to different configurations and types of heterogeneous devices, OpenCL has been used as a basis for the development and experimentation of this research.

Nevertheless, OpenCL has a caveat. It offers a low level of abstraction, leaving the programmer alone with the responsibility of manually managing data and devices individually. To explore a greater level of abstraction, the OmpSs extension to support OpenCL has also been used in this work, precisely in the areas related to Chapter 5. The following subsections are devoted to introducing the main features of OpenCL and OmpSs.

2.1.1. Heterogeneous programming in OpenCL

OpenCL [SGS10, KMSZ15], released by the Khronos Group in 2008, is a standard framework for the development of applications that will run on heterogeneous systems. One of the main advantages of OpenCL is its adaptiveness, as it is not limited to hardware of a certain type or from a particular vendor. Given an adequate OpenCL driver, which should be supplied by the hardware vendors, any application following the specification will run adequately on any device, ranging from GPUs [KH10b] or Xeon Phi [Rah13] to FPGAs [DGNGT+19]. This allows for the development of portable applications that can be accelerated regardless of the characteristics of the underlying system.

OpenCL implements a Host-Device approach to heterogeneous programming, depicted in Figure 2.1. One device, known as the host, is in charge of the management of the heterogeneous system, while the others act as accelerators, working on the more compute-intensive portions of the work offloaded to them by the host. To achieve this, OpenCL is formed by two parts: a C application programming interface (API) and a programming language. The API works as an abstraction layer. It implements hardware independent functions to manage the heterogeneous devices, offload work to them and perform data transfers. This works towards one of the main features

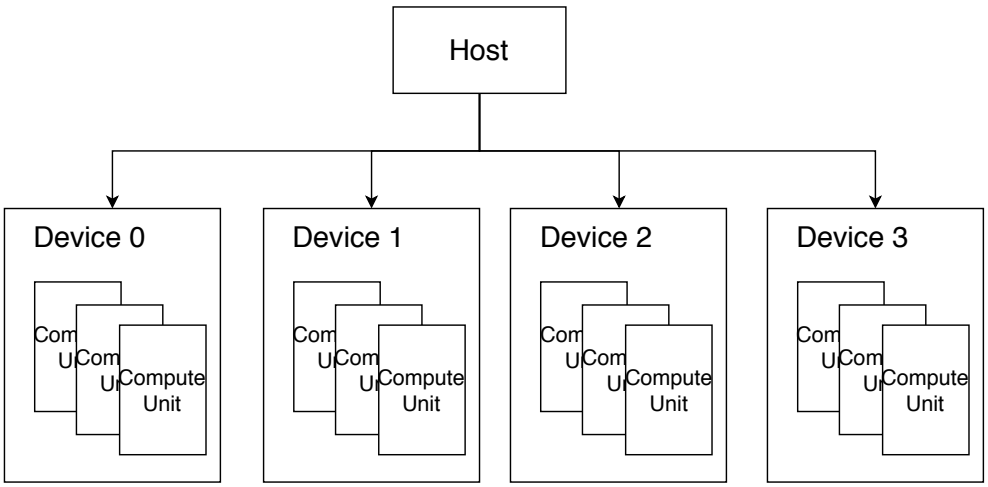


Figure 2.1: The Host-Device programming model.

of OpenCL, which is portability. The offloaded work is expressed in the form of functions implemented in OpenCL C, a subset of C99 with certain extensions for data-parallelism. The abstractions offered by OpenCL through its API and the data parallel programming model it follows will be explained next.

Device abstraction

To guarantee that an OpenCL application can run on any hardware configuration, devices are treated abstractly. The abstraction is based on four main concepts: compute units, platforms, contexts and command queues. Devices are defined as a combination of *compute units*, as shown in Figure 2.1, which are functionally independent computation elements. For example, a 4 core CPU is regarded as a combination of 4 compute units or a GPU with 13 stream multiprocessor as comprised by 13 compute units. Devices are also grouped in *platforms*, which are representations of the OpenCL implementations available in the system. These are usually vendor specific. For instance, lets define system A and system B, depicted in Figure 2.2. System A comprises an Intel CPU and an AMD GPU whereas in system B both the CPU and the GPU are AMD. Considering the definition of a platform, System A has two platforms, one for the CPU and another one for the GPU. For System B, in turn, only one platform would be available, comprising all the devices. The OpenCL specification then defines the *context* as a representation of a combination of devices belonging to a platform, that will be

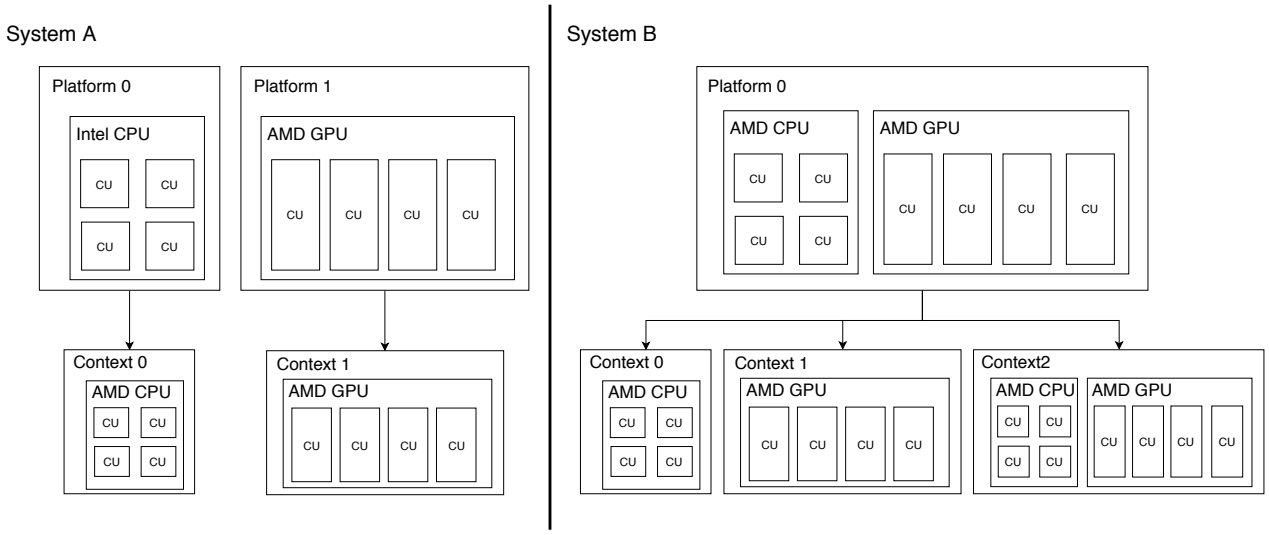


Figure 2.2: Platform and context possibilities for two sample systems.

used in the computation. In our example systems, one context per platform would be available in A, while 3 would be possible for B: one containing only the CPU, one containing only the GPU and a third one containing both the CPU and the GPU. Note that there is no context in system A comprising both the CPU and the GPU, as they belong to different platforms. Lastly, the *command queue* is the abstraction used to communicate with a specific device by performing data transfers, offloading work or waiting on certain events. The management of these structures is performed through the API. Considering the above, the initial steps of an OpenCL application would be:

1. Identifying the available platforms.
2. Selecting a context containing the devices that are to be used.
3. Creating the necessary command queues to communicate with the devices.

Execution abstraction

The OpenCL API offers functions to specify the work to be executed by the devices and perform the offload operation. *Kernels* are defined as functions that will be executed by a device. These represent the work to be performed by each launched thread or, as referred to in OpenCL terminology, *work-item*. OpenCL also defines the notion of a *work-group* as a set of work-items

that are able to progress in the presence of barriers and that can synchronize. This implies that the work-items of a work-group need to be executed concurrently in a compute unit, to guarantee that a switch is possible and relatively inexpensive provided a work-item has to block. The OpenCL specification enforces this requirement, however the actual mapping of work-groups to hardware components is both architecture and OpenCL implementation dependent. Synchronization between work-items belonging to different work-groups is undefined. Kernels are expressed in OpenCL C which, among other extensions, offers functions that enable the work-items to identify themselves in their work-group and in the whole body of work-items that have been launched.

On the host code side, kernels are represented in an abstract manner using a data structure also known as kernel. These are context and device dependent structures, as the OpenCL C code needs to be compiled for each of the architectures that will execute it. The API offers the necessary function to create kernels (data structure), compile them, specify the arguments they use and launch them. Each kernel offload operation has to be accompanied by a specification of the number of work-items to be launched in the device and their distribution. This is defined using two parameters of the kernel offload function: the *global_work_size* and the *local_work_size*. The former specifies the total number of work-items to spawn and the latter the number of work-items for each work-group, implicitly specifying the total number of work-groups. A third parameter, the *global_work_offset* is also available, which acts as a displacement for the work-item id calculation. These three parameters may have up to 3 dimensions to adapt to different problems and algorithmic approaches.

Memory abstraction

Considering that the heterogeneous resources of a system are usually independent pieces of hardware, the memory spaces of a device and that of its host have been traditionally considered as separated. There are some examples of recent hardware that implements shared memory between host and devices. However, in this kind of systems, memory sharing is often offered at the cost of a highly degraded performance due to overheads. For this reason and considering

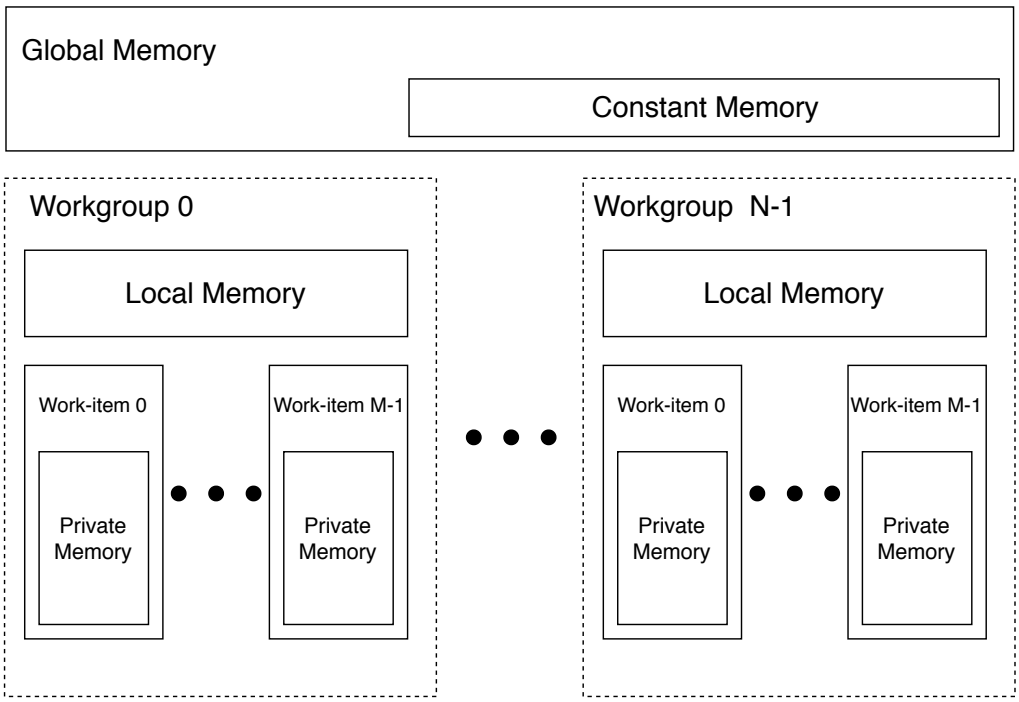


Figure 2.3: OpenCL memory model.

that separate memory spaces are still the norm, they have been the focus of this work. Moreover, the proposed techniques would also apply to shared memory systems if they become a viable option for performance in the future.

To account for separate memory spaces, data transfers are necessary to both provide the accelerators with input data for the offloaded work and to retrieve the results. Furthermore, these transfers need to be performed in an abstract fashion to preserve portability. The OpenCL standard defines the *buffer* as a memory address that is valid in the memory of an OpenCL device. The OpenCL API offers functions to create buffers on devices and to enqueue read and write operations on the command queues associated to them. These operations have the host as originator, so, for example, a read implies a data movement from a device memory to the host memory. Transfers can also be blocking or non-blocking, to control whether the host has to wait for the transfers to complete or can continue executing.

For flexibility, OpenCL also defines an abstract memory model. It defines four different memory spaces as shown in Figure 2.3. Both the *Global* and *Constant* memory spaces are shared by every work-item of a kernel launch, while every work-group has its own *Local* memory. This is a programmer managed memory space, commonly used as a scratchpad for fast collaboration

and data sharing at the work-group level. Each work-item has its own *Private* memory. The mapping of memory spaces to actual hardware is implementation dependent.

2.1.2. Task-based programming languages

Writing parallel code is a challenging task, as it requires to orchestrate the execution of several different devices. This includes managing the distribution of data to the devices as necessary, and also deciding what parts of the application can be run in parallel and when to launch them. The resulting applications are often complex and hard to maintain. Task-based programming languages strive to make the programming of this kind of systems simpler, by keeping it closer to traditional, sequential programming. Their take on parallelism is based on the *task* concept, which is often a function that may be executed in parallel. The programmer is then in charge of defining tasks and the data they use, leaving the management of their execution to a runtime, and thus making programming easier.

OmpSs is a programming model, created and maintained by the Barcelona Supercomputing Center, for the development of parallel applications. It was originally based on OpenMP [ope15] and StarSs [PBL08] but supports heterogeneous systems, both using CUDA and OpenCL, through an extension [DAB+11, F. 14].

OmpSs proposes a data-flow driven parallel programming model, in which tasks are asynchronously executed in parallel. Tasks are declared using compiler directives in the code of the application. The data dependencies between tasks are also declared through pragmas, so OmpSs can build a dependence graph and automatically execute tasks when their input data are ready. This is obtained thanks to the two main pieces that build OmpSs:

- *Mercurium*, which is a source-to-source compiler that processes certain directives (pragmas) and generates the actual code of the parallel application that will be executed [Mer].
- *Nanos++* , which is a runtime library that provides the necessary functionality for the application generated by Mercurium to be able to run. These include the creation of

```
#pragma omp task in([n]a) out([n]b)
void task1(int n, int * a, int * b)

#pragma omp task in([n]c) out([n]d)
void task2(int n, int * c,int * d)

#pragma omp task in([n]e) out([n]f)
void task3(int n, int * e, int * f)

#pragma omp task in([n]g, [n]h)
int task4(int n, int * g, int * h)
```

Figure 2.4: Headers for the tasks.

tasks, the generation and management of the dependence graph, the management of data and the launch of tasks when their data is available [Nan].

When a parallel program is executed, a thread pool is created with only one thread, the master, set as active. This thread uses the capabilities of Nanos++ to create tasks as specified in the pragmas and translated by Mercurium. Figure 2.4 shows the definition of four sample OmpSs tasks. Tasks are internally identified by *work descriptors* and added to the dependence graph, taking into account the information on data provided in the *in* and *out* clauses. In our example, function `task1` is defined as a task using `a` as input and `b` as output (`task2-4` are equivalently defined). Figure 2.5 shows an example launch of the tasks defined in Figure 2.4 and the dependence graph that it would generate. The master thread then schedules tasks to the available threads as soon as their input data is ready or, in OmpSs terminology, when their dependencies are satisfied. In the example, tasks 2 and 3 will be launched as soon as task 1 finishes, while task 4 will have to wait for the termination of tasks 2 and 3 to be executed.

2.2. Heterogeneous system simulation in gem5

Gem5 [BBB+11] is a modular simulation platform for computer-system architecture research, encompassing system-level architecture as well as processor microarchitecture. It provides several interchangeable CPU models, ranging from a simple processor to a detailed out-of-order CPU model, and four different architectures: Alpha, ARM, SPARC and x86. For the memory

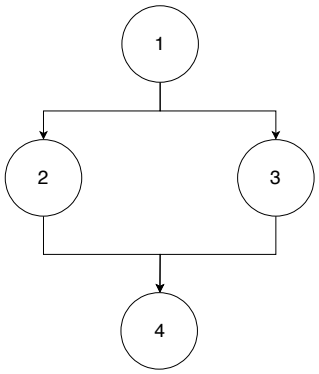
```
//Initializations
task1(n, v1, v2);

task2(n, v2, v3);

task3(n, v2, v4);

task4(n, v3, v4);
#pragma omp taskwait
//Free resources
```

(a) Launch of the tasks.



(b) Generated DAG.

Figure 2.5: Code for the launch of the tasks and generated OmpSs dependence graph.

subsystem, gem5 features Ruby, an event-driven simulator that models caches, crossbars, snoop filters and the DRAM. Gem5 supports two execution modes: *syscall emulation*, in which only the binary of an application is executed in the simulated system, and *full-system*, in which the complete OS is executed. All these capabilities make Gem5 the *de facto* standard architecture simulation tool in academic research. Regarding heterogeneous systems, the stable version of Gem5 includes a GPU model that represents an integrated SoC with support for shared virtual memory with the host CPU. This feature and its OpenCL backend, which are specially interesting for the study of integrated heterogeneous systems, will be explained next.

2.2.1. The GPU model

To simulate integrated heterogeneous systems, gem5 offers the GPU model. Proposed in [BMB15, GBD+18], it is modelled after the architecture of GCN AMD GPUs [AMD12]. In GCN, the GPU is formed by a set of *compute units*, an example of which is depicted in Figure 2.6. Each compute unit holds 4 SIMD units that can process 16 work-items in parallel and a private L1 data cache. Each group of 4 compute units shares an L1 instruction cache, while the L2 is distributed and shared among all the compute units. L1 caches are write-through, so they are kept coherent with the L2 cache, which is write-back. Both are virtually addressed, so the memory hierarchy is integrated with x86 microprocessors and the memory space shared. Coherence with main memory is kept through a protocol called Viper. All the compute units also share a front-end, which is in charge of storing the information of kernels and dispatching

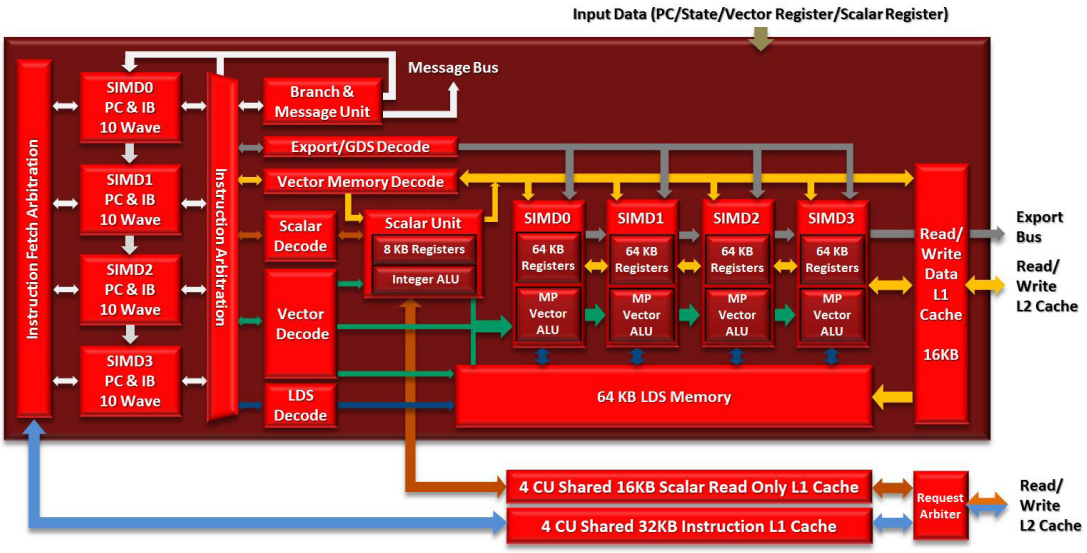


Figure 2.6: Representation of the GCN compute unit. Taken from: White Paper. AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE [AMD12]. Copyright AMD

work-groups to compute units. Regarding the complete GCN heterogeneous system, the CPU and GPU are two independent devices that can communicate and share the main memory.

2.2.2. OpenCL support

As explained in Section 2.1.1, OpenCL devices require a driver to communicate the runtime and the hardware through the OS. Adequating the GPU model to the working of a real driver would be very complex, as would be implementing a new, tailor-made driver. The implementation of the gem5 GPU model chooses to avoid the OS altogether, by implementing its own OpenCL backend and executing in syscall emulation mode. Consequently, OpenCL applications need to be compiled using a modified library that calls the necessary specific functions of the backend. Kernels also require a special treatment. They need to be compiled using CLOC, an offline compiler that generates an HSA object, which is then interpreted by the gem5 GPU model.

Two important limitations are worth noting. First, the GPU Model does not implement the complete GCN ISA. As a result, certain valid OpenCL kernels are rejected by the OpenCL backend and generate an error when the application is executed. This has proven an issue when running benchmarks on the simulator. Second, the backend and modified OpenCL library have no support for the execution of OpenCL on CPUs. This limitation will be addressed in the

implementation of hardware support for co-execution.

2.3. Related Work

Heterogeneous architectures have quickly permeated every computing system, ranging from large scale supercomputing servers to portable devices. This proliferation has given rise to an interest in how to adequately use the extra computing power and efficiency that these architectures offer [KC18]. This is a multi-faceted problem, as it involves hiding both the difficulties of handling several devices and the complexity of load balancing, which may be deeply influenced by the available devices and the workload to co-execute itself.

2.3.1. System abstraction

To abstract the programmer from the management of the resources available in the heterogeneous system, a framework is usually offered. It acts as an intermediate layer, separating the programmer from the underlying heterogeneous programming framework, such as OpenCL, and providing access to a virtual device that represents the whole system. This is achieved by using a combination of runtimes [LSPM15, YWTC15, PG14, HCY+14, APnBF16, DMSD16, BBK17, GLMR13, LSM15, ZH13, KSL+12, LSPM13, CGS+15], compilers [KKLL11, KSL+12], kernel code modifications [HCB16, LSM15, LSPM13, LSPM15], libraries [dlLTBR12] and hardware support [CGS+15]. Out of these works, [LSPM15, PG14, KKLL11, HCB16, dlLTBR12, LSPM13] target co-execution, whereas the rest focus on task-parallel approaches. These orchestrate the execution of different tasks or kernels and manage data dependencies, in a similar manner to OmpSs. On the contrary, co-execution poses the special challenge of keeping the memory of the devices consistent, as they all collaborate on the computation of a single kernel. To do so, these frameworks either keep track of data differences to merge the results in the GPU at the end of the execution [PG14], or perform complex buffer access analysis to identify patterns and split data accordingly among each of the available devices [KKLL11, LSPM15, HCB16, LSPM13]. These are often costly operations that represent a significant overhead.

2.3.2. Load balancing

The key to efficiently using a heterogeneous system is an adequate distribution of the workload among the available resources. Task-Parallel approaches often take load balancing decisions based on the history of prior executions. Several works have followed this path, either by building a regression model [YWTC15], using greedy algorithms [HCY+14], implementing a kernel and device profiler [APnBF16] or identifying history-based energy efficient work schedules [DMSD16]. Another common approach is the traditional technique of work-stealing [BBK17, GLMR13]. To completely use the available resources, other works propose to concurrently launch a second independent kernel to a device, if the first one is detected to not fully use the hardware capabilities [LSM15, ZH13].

The problem of co-execution poses different challenges from those of task-parallelism, as the potentially varying behavior of kernels and devices needs to be accounted for. Several publications have tackled co-execution via different kinds of static approaches [KKLL11, LSPM13, dlLTBR12, KGCF13, ZRL15, LSPM15, HCB16]. These works are remarkable efforts towards the efficient use of heterogeneous systems, but lack the necessary adaptiveness to represent an all-around solution to the load balancing problem. Other authors propose training-based load balancing schemes, either using previous execution information only [SMV10] or based on calculating a performance model [LHK09]. These works lack adaptiveness too, as they only perform well for known applications and do not address irregularity.

To be able to adapt to the singularities of every workload and device configuration, dynamic approaches, that are capable of making decisions at runtime, are necessary. These usually involve two steps: tracking the behavior of the kernel to identify the computing capabilities of the devices at runtime and modifying package sizes accordingly, so imbalance and overheads are avoided. To try to discover the ideal package size, some works monitor the observed throughput by using increasing package sizes until no performance improvement is detected [PG14], while others use throughput to tune CPU package sizes to avoid imbalance, while using a fixed size for the GPU [NVCA14]. However, these works fail to properly address the importance of irregularity. Others use the first portions of the workload as "probes" to get information to

schedule the rest of the workload statically [KBS+14, BSCJ13] using a predicted computing speed. Another work found that, for regular applications, the GPU throughput follows a logarithmic curve with respect to the package size. To address this behavior, [BBG13] replaces the static distribution of [KBS+14, BSCJ13] with a weighted self-scheduling scheme. However, these techniques do not sufficiently address applications with irregular behavior as they assume that the probe packages launched in the initial phase are representative of the whole load.

The observation that GPU throughput follows a logarithmic curve has also been used to model the behavior of irregular applications. The hypothesis is that irregular applications show different local logarithmic curves depending on the behavior of each execution region. This has been used to create a mathematical model that predicts the throughput of the next package based on a logarithmic approximation of the throughput of the previous packages [VAN+15].

Some authors propose load balancing techniques that take both performance and energy into account. For instance, GreenGPU dynamically distributes work to GPU and CPU, minimizing the energy wasted on idling and waiting for the slower device [MLC+12]. To maximize energy savings while allowing marginal performance degradation, it dynamically throttles the frequencies of CPU, GPU and memory, based on their utilization. Frequency scaling and inter-processor work distribution are also used in [WR10] to minimize energy consumption under a given execution time constraint.

2.3.3. Integrated heterogeneous systems and hardware support

Architectures that integrate CPU cores and GPU compute units in a single chip are one of the options to reach the exascale, as represented by the EHP design introduced in [VEL+17]. Nevertheless, the orchestration of CPU and GPU poses challenges of its own, as the effects of sharing certain hardware elements is yet unknown. This is the reason why this kind of systems are an active research field. One of the topics that has generated the most debate is the sharing of the memory between CPU and GPU, which has been studied in [GGG+16, HKW14, HKW15]. Another hot topic in heterogeneous systems is maintaining the memories of the CPU and the GPU coherent. A proposal to do so was presented in [PBG+13]. However, this kind

of techniques may sometimes represent a significant performance degradation. That is why [ANE+16] uses methods to selectively cache the data needed by the GPU to keep the coherence protocol from generating inefficiencies. This technique was designed for discrete systems, but it could also be interesting in integrated environments. Some authors propose to use the CPU to guide the execution of the GPU. Such is the case of [YXMZ12], in which the CPU executes a simplified version of the code of the GPU, which effectively acts as a prefetcher.

Parallel systems can also be more efficiently used by providing hardware support to capabilities offered by the software. This sometimes results in significant performance increases with small hardware changes. The work presented in [CMC+16] enables the hardware to reconfigure itself through DVFS using task criticallity information. TDM [CAM+18] tracks task dependence information in hardware, significantly reducing the overheads of task-based programming models.

Chapter 3

Load balancing algorithms for co-execution

One of the keys for the successful use of a parallel system is deciding how the workload is distributed. Consequently, load balancing is a classic computing problem, that has always caused interest in the research community. Nowadays, systems have gone heterogeneous, integrating more and more devices and adding more complexity to the task of efficiently using their capabilities. This has opened up a new field of investigation on load balancing. A fair amount of effort has been devoted to devising techniques to adequately distribute tasks among the devices of a heterogeneous system, so it is efficiently used. However, the problem of splitting a single workload among all the devices has not been sufficiently addressed. Co-execution allows for a more efficient use of the system, as its parallelism is not limited by the number of available tasks. Moreover, it makes the programming of heterogeneous systems easier, offering an approach closer to sequential programming, similar to OpenMP. This chapter presents *HGuided* and *Sigmoid* two novel load balancing algorithms specially designed for heterogeneous co-execution. For their adaptiveness, these algorithms succeed at adequately balancing both regular and irregular workloads, and by eliminating the need for user parameters, they represent a significant step towards effortless heterogeneous co-execution.

3.1. Requirements for heterogeneous co-execution

As introduced in Section 1.2.2, an accurate load partition will minimize the time that the devices spend idling, so both performance and energy efficiency are improved. This is a complex decision, as it requires the behavior of the co-executed kernel, the computing capabilities of the devices and the communication overheads to be taken into account. Regular kernels benefit from dividing the workload in fewer portions, best results often being achieved when a single, appropriately sized chunk, is generated per device, so communication overheads are minimized. On the contrary, irregular kernels require finer grained load balancing, specially near the end of the execution, to achieve the necessary adaptiveness to avoid imbalances. As a result, for an algorithm to successfully balance both regular and irregular workloads, it has to preserve enough adaptiveness without excessively penalizing regular workloads that do not require it.

This chapter proposes two new load balancing algorithms that address these requirements. The *Heterogeneous Guided* (HGuided) algorithm does it by using decreasing package sizes, tuned to the computing speed of the executing device. This algorithm achieves excellent results both for regular and irregular kernels, but still does not represent a truly effortless option for co-execution, as it is an informed algorithm, requiring certain parameters from the programmer. Accurately setting the values for these parameters requires certain effort and expertise, and failing to do so sometimes represents significant performance degradation. For this reason, this dissertation proposes a second novel load balancing algorithm that does achieve effortless co-execution: the *Sigmoid*, which not only is uninformed, but also delivers even slightly better performance than HGuided. It does so by monitoring the kernel execution and making load balancing decisions at runtime, to better adapt to the behavior of the kernel that is being executed. The following sections are focused on providing a deeper insight on HGuided and Sigmoid. The algorithms have been implemented as part of Maat, a load balancing and system abstraction library presented in Chapter 4, which also evaluates them.

3.2. Problem Definition

Before diving into the algorithms themselves, lets first define the load balancing problem in terms of the information available to the algorithms, which will be used in their description. Lets define a heterogeneous system $H = \{D, S\}$, where D is a set of N devices $\{d_1, ..., d_N\}$ and $S = \{s_1, ..., s_N\}$ their corresponding computing speeds, expressed in work-groups per second, and a kernel K that launches W work-items. The value for s_i depends on its associated device d_i, but also on K, as devices do not show the same speed for every kernel. Work-items are grouped in G work-groups, each of fixed size $L_s = \frac{W}{G}$, and G_r is the number of work-groups remaining to be scheduled. Since inter-work-group communication is undefined in OpenCL, it makes sense to choose the work-group as the unit of distribution.

In general, the response time of d_i running W_i work-items will be $T_{d_i} = \frac{W_i}{s_i}$, and that of H running K will depend on the last device to finish its work $T_H = max_i^N T_{d_i}$. As a result, a load balancing algorithm will strive to find a distribution of work to the devices so that $T_{d_i} \approx T_{d_{i+1}} \forall i \in [1, N]$. This is, a balanced work partition. Also, since the whole system is capable of executing W work-items in T_H, it follows that its total computing speed for this partition is $s_H = \frac{W}{T_H}$. Note that it also can be computed as the sum of the individual speeds of the devices.

$$s_H = \frac{W}{T_H} = \sum_{i=1}^{N} s_i$$

The goal of a load balancing algorithm is to determine the number of work-groups to assign to each device, so that all the devices finish their work at the same time. This means finding a tuple $\{\alpha_1, ...\alpha_n\}$, where α_i is the number of work-groups assigned to the device i, such that:

$$T_H = T_i d_1 = \cdots = T_{d_N} \Leftrightarrow \frac{L_s \alpha_1}{s_1} = \cdots = \frac{L_s \alpha_N}{s_N}$$

This set of equations can be generalized and solved as follows:

$$T_H = \frac{L_s \alpha_i}{s_i} \Leftrightarrow \alpha_i = \frac{T_H s_i}{L_s} = \frac{T_H s_i G}{W} = \frac{s_i G}{\sum_{i=1}^{N} s_i}$$

Since α_i is the number of work-groups, its value must be an integer. For this reason:

$$\alpha_i = \left\lfloor \frac{s_i G}{\sum_{i=1}^{N} s_i} \right\rfloor$$

If there is not an exact solution with integers then $\sum_{i=1}^{N} \alpha_i < G$. In this case, the remaining work-groups are assigned to the fastest device.

To try to find the values for α_i, load balancing algorithms often group work-groups into a number np of *packages*, either equally or differently sized, that will be correspondingly offloaded to the devices. However, achieving load balance is not the only figure of merit. A dynamic work distribution of a regular workload that uses $np = G$ is likely to achieve almost perfect load balance, but also poor performance, due to inefficient device use and excessive overheads. Therefore, a good load balancing algorithm will attempt to obtain very close execution times for a balanced work distribution and a small np to avoid overheads and appropriately use the devices.

3.3. The HGuided Algorithm

3.3.1. Overview

Considering that different applications show different behaviors, for a single load balancing algorithm to successfully adapt to different computing patterns, it needs to be dynamic, distributing the kernel at runtime. This is, it must split the workload into relatively small packages, whose number is larger than the amount of devices in the system. Packages are then scheduled at runtime on demand, achieving a certain degree of adaptiveness, determined by np, to avoid imbalances near the end of the execution. Applied to a heterogeneous environment, following a

host-device approach to parallelism, a thread running on the host will be in charge of assigning packages to the different devices, following the strategy depicted in algorithm 1.

Algorithm 1: Sample dynamic algorithm

Input: The number of work-groups G, a set of N devices with s_j default computing speeds

$G_r \leftarrow G$ *(Number of work-groups)*
for $j \leftarrow 1$ ***to*** N **do**
  $x \leftarrow Gr$
  $c \leftarrow package\_size(d_j, x)$
  Schedule c work-groups on device d_j
  $Gr \leftarrow Gr - c$
end
while $G_r > 0$ **do**
  $(d_j, c, t) \leftarrow$ Wait for any device
  $x \leftarrow G_r$
  $c \leftarrow package\_size(d_j, x)$
  Schedule c work-groups on device d_j
end

The decision of what package size to use has been intentionally left out of the previous algorithm outline. A common approach for a basic dynamic algorithm is to split the workload into small, fixed size packages. Such an algorithm adapts to the irregular behaviour of some applications. However, each completed package represents an interaction between the device and the host, in which data is exchanged and a new package is launched. This overhead has a noticeable impact on performance. Therefore, *np* needs to be kept as low as possible without compromising adaptiveness. Moreover, when considering heterogeneous systems, in which devices usually show significant computing speed differences, a fixed package size dynamic algorithm is not the best choice. This is because big package sizes are likely to produce imbalances, by scheduling the last package to the slowest device, while small ones will increase overheads and potentially inefficiently use fast devices.

The Heterogeneous Guided algorithm (or HGuided) is a novel take on load balancing that strives to better address the particularities of heterogeneous systems and irregular workloads. It follows a dynamic scheme and uses diminishing package sizes, in a similar fashion to the OpenMP Guided algorithm [ope15], to keep *np* under control, while fine grained adaptiveness is preserved near the end of the execution. This is, when load balancing decisions may be determining.

HGuided also considers the computing speed of the device that will receive the package, in order to properly use the computing capabilities of all the devices and avoid imbalances. The package size for device d_i is calculated as follows:

$$package\_size_i = \left\lfloor \frac{G_r}{CN} \cdot \frac{s_i}{\sum_{j=1}^{N} s_j} \right\rfloor$$

Note that the first term gives decreasing package sizes, as a function of the number of pending work-groups G_r, the number of devices N and a constant C. G_r will vary throughout the execution of the kernel, starting at W and decreasing as packages are scheduled, until 0 is reached. C is introduced to control the size of the first package, which will be the biggest generated by the algorithm, so it does not represent excessive computation in irregular workloads. This constant was empirically fixed to 2 in the experimental evaluation. This is a common value often used in OpenMP [ope15]. The second term adjusts the package size with the ratio of the computing speed of the device s_i to the total speed of the whole system. Therefore, the size of a package depends on the device that will execute it. Consequently, np will vary according to the order in which packages are launched to the devices. This can differ greatly between runs, especially under irregular workloads.

This function produces steadily decreasing package sizes as shown in the top graph of Figure 3.2. However, excessively small packages may generate overheads and inefficiently use the devices. To prevent these issues that might have a negative impact on performance, a *minimum package size* is defined. It acts as a lower bound for the size of the packages, which will not decrease beyond its value. This, together with the computing speed of the devices, are parameters required by the algorithm. The computing speed may be computed by performing a preliminary run of the kernel on each of the available devices.

The HGuided algorithm strikes a balance between adaptiveness and overheads. It does so by using large packages at the beginning of the execution, which reduces overheads, and decreasing their size near the end of the execution, improving the accuracy of the load balancing. In addition, this is done while keeping the utilization of the devices nearly constant throughout

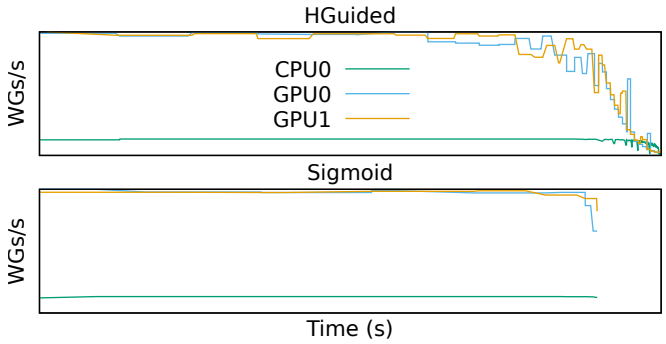


Figure 3.1: Computing speed comparison for Binomial.

the execution. This is because excessively small packages are avoided, which have a strong impact on performance. HGuided is evaluated in detail in Section 4. This algorithm has been used in other works such as EngineCL [GNT+19], in systems that hold FPGAs.

3.3.2. Limitations

HGuided generates fewer packages than a base dynamic approach, thus reducing overheads and improving performance. Nevertheless, it still has certain weaknesses. For instance, HGuided still does not fully leverage the capabilities of the devices. The top graph of Figure 3.1 shows the evolution of the computing speed, expressed in work-groups per second, when a system runs the Binomial kernel using the HGuided algorithm. Notice that, for a significant portion of the execution, the devices are going slower than they could.

This is because HGuided produces linearly decreasing package sizes, as shown in Figure 3.2 for the same benchmark. The result is a great deal of small packages near the end of the execution that can't exploit all of the computational resources of the most powerful devices. These may be particularly impactful for regular kernels, which do not require adaptiveness and benefit from lower amounts of packages.

The HGuided algorithm is also an informed algorithm, requiring the minimum package size and the computing speeds to be set as parameters from the programmer, which strongly condition the success of the load balancing. To be accurately set, they require costly, per-kernel and per system sweeps, as the best values may vary widely between kernels. Moreover, some kernels are

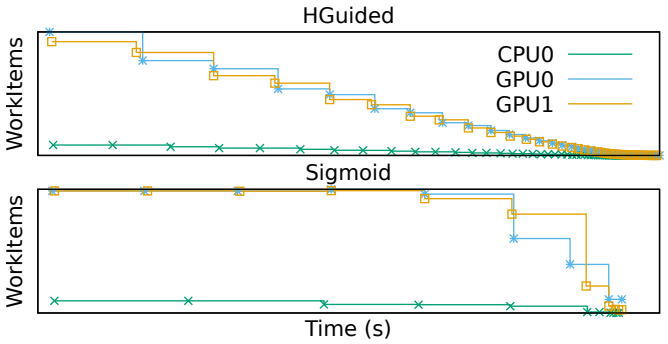


Figure 3.2: Package size comparison for Binomial.

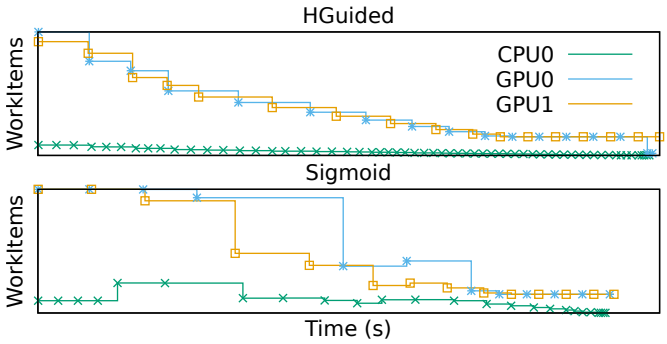


Figure 3.3: Package size comparison for BM3D.

very sensitive to the parameters, delivering highly degraded performance when using slightly off-key values. This results in dozens of tuning executions, which represent a waste of time, resources and energy. Furthermore, this is unfeasible in dynamic environments, such as a data center or cloud, where different applications may run on different systems, with an a priori unknown matching. The Sigmoid algorithm, which is presented next, solves these issues to constitute a truly effortless load balancing algorithm, requiring no intervention from the programmer to achieve near-optimal performance under both regular and irregular workloads.

3.4. The Sigmoid Algorithm

The Sigmoid algorithm was conceived in an effort to fulfill the requirements for a truly effortless uninformed load balancing scheme, which are listed below:

1. It should optimally divide a single massively data-parallel kernel between a set of heterogeneous devices.

2. It should evaluate the computational performance of the devices with a minimum overhead.

3. It should deliver good results with any type of kernel; regular or irregular.

4. It should not require any configuration from the programmer.

3.4.1. Overview

To accomplish these objectives, Sigmoid is a *dynamic* and *heterogeneous* algorithm, because it is able to distribute the workload among devices at runtime, proportionally to their computing power. By matching the package size to the computing speeds of the devices, excessively large packages are not assigned to slower devices, and the use of more powerful ones is maximized. It is also *adaptive*, since it is capable of modifying its operation to suit the type of kernel under co-execution. It will divide regular kernels in larger packages to reduce overhead, and use smaller ones for irregular kernels, as it is impossible to predict their execution time. To do this, it continually measures the performance of the devices and tunes a number of internal parameters accordingly. Unlike other proposals [BBG13, BSCJ13, NVCA14], this parameter tuning is performed transparently to the programmer and without significant loss of performance. Thus, Sigmoid behaves equally well with both regular and irregular applications. Finally, it is a *guided* algorithm, since package size, which is initially proportional to the computing power of each device, decreases towards the end of the kernel execution.

Since Sigmoid takes a dynamic approach to load balancing, it first launches an initial package to each of the available devices and then waits until any of them completes their execution. The packages are sized in accordance to an initial computing speed, based on the GFLOPs reported by the specs of the devices. When one finishes the execution, if there is pending work, a new package is generated and issued to the idle device. To improve the load balance, the size and response time of completed packages are analyzed to tune the internal parameters of the algorithm throughout the execution of the kernel.

How the package size evolves throughout the execution of a kernel is key to an efficient load balancing. This is because package size poses a dilemma: smaller packages garner greater adaptiveness, but also greater overhead. Moreover, computing speed is sometimes correlated with the amount of work launched to a device, so small packages often lead to suboptimal performance that is not representative of the actual capabilities of the hardware. The importance of this phenomenon has been already addressed in [BBG13]. Thus, a good load balancing algorithm will attempt to keep computing speeds and, consequently, package sizes as high as

possible, while not compromising adaptiveness.

To calculate successive package sizes, Sigmoid relies on a function that issues big packages for most of the execution and, gradually, smaller ones at the end, which reduces overheads while maintaining adaptiveness. The decrease rate of the package size is adjustable through an internal parameter, which varies depending on the behaviour of the kernel.

To calculate successive package sizes, Sigmoid relies on a function that issues big packages for most of the execution and, gradually, smaller ones at the end, which reduces overheads while maintaining adaptiveness and keeping device utilization high. This is depicted in Figures 3.1 and 3.2, which compare the computing speed and package size evolution when executing the Binomial benchmark with the HGuided and Sigmoid algorithms. Note how Sigmoid generates fewer and bigger packages, maintaining computing speed high for longer. The decrease rate of the package size is adjustable through an internal parameter, which varies depending on the behaviour of the kernel.

Sigmoid uses the size and response time of executed packages to detect if the kernel is irregular, and adjust the decrease rate to generate smaller packages, if more adaptiveness is required. The algorithm also automatically identifies an adequate minimum package size that strikes a balance between adaptiveness and performance, and calculates the computing speed of the devices to avoid imbalances. The result is an algorithm that adapts to the behavior of kernels. This is shown in Figure 3.3, which compares the package sizes generated by HGuided and Sigmoid for an irregular kernel. Again, HGuided generates linearly decreasing package sizes, although a certain distortion can be appreciated due to irregularity. Sigmoid, in turn, uses variable package sizes to adapt to the kernel. This can be seen in the humps near the end of the execution of GPU0 and GPU1, which account for computing speed variations associated to package workload differences. An exhaustive package size evolution analysis has been carried out for every evaluated application, however it has been left out due to space limitations. Figures 3.2 and 3.3 have been found representative of the behavior of regular and irregular kernels respectively. A high level description of the algorithm can be seen in Algorithm 2. The following sections explain the different internal parameters and functions of the algorithm.

Algorithm 2: Sigmoid algorithm

Input: The number of work-groups G, a set of N devices with s_j default computing speeds

$G_r \leftarrow G$ *(Number of work-groups)* $k \leftarrow k_r$ **for** $j \leftarrow 1$ ***to*** N **do**
  $oc_j \leftarrow$Occupancy lower bound for device d_j
  $x \leftarrow Gr$
  $c \leftarrow package\_size(d_j, x, k)$
  $c \leftarrow max(c, p \cdot t \cdot s_j, oc_j)$ Schedule c work-groups on device d_j
  $Gr \leftarrow Gr - c$
end
while $G_r > 0$ **do**
  $(d_j, c, t) \leftarrow$ Wait for any device
  $s_j \leftarrow$ Average of the last 3 computing speeds $(\frac{c}{t})$
  $\sigma_{s_j} \leftarrow$ Standard deviation of last 3 computing speeds
  if $\frac{\sigma_{s_j}}{s_j} > 0.2$ **then**
    $k \leftarrow k_i$
  end
  $x \leftarrow G_r$ $c \leftarrow package\_size(d_j, x, k)$
  $c \leftarrow max(c, p \cdot t \cdot s_j, oc_j)$ Schedule c work-groups on device d_j
  $Gr \leftarrow Gr - c$
end

3.4.2. The logistic function and the load balancing problem

The logistic function is used to model processes that can be observed in many fields, ranging from biology or medicine to machine learning [DOM02, Bis06]. This function, conveniently transformed, is the foundation of the Sigmoid load balancing algorithm. It is defined by the following equation and a graphical representation is shown in Figure 3.4.

$$logistic\_function(x) = \frac{L}{1 + e^{-k(x - x_0)}} \tag{3.1}$$

To apply the function to the load balancing problem, variable x will represent the amount of remaining work-groups. Consequently, it will be monotonically decreasing and take values between G, the total number of work-groups that have to be processed, and 0. As x will always be positive, parameter x_0 is eliminated. The maximum value of the function, L, will represent

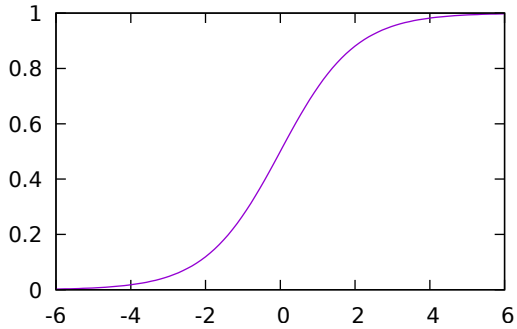


Figure 3.4: Representation of the logistic function for $L = 1$, $k = 1$ and $x_0 = 0$.

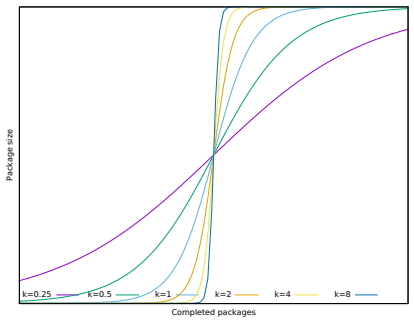


Figure 3.5: Evolution of the package size for different k values.

the size of the largest package. It can be seen in the figure that for $x = 6$ the function yields a value close to the asymptotic maximum. Since the scheduling unit is relatively coarse, this is considered enough, and Sigmoid will only use the function in the $[0, 6]$ interval. Variable k is the slope of the curve. The following lines show in detail how the internal Sigmoid function, represented in Figure 3.5, is derived from the logistic function.

Let G_r be the number of remaining work-groups. As the aim is to obtain a function f that produces decreasing package sizes as the execution of the kernel progresses, G_r will be used as the x in the logistic function. However it will be normalized to the total work-groups G and scaled to 6 to map it to the $[0, 6]$ interval.

$$f(G_r) = \frac{L}{1 + e^{-k\frac{6(G_r)}{G}}} \tag{3.2}$$

So far, since $0 \leq G_r \leq G$, then $f(G_r)$ returns values between $\frac{L}{2}$ and L. It is necessary to transform this to appropriate package size values, which should have their minimum at 0. First, the range of the function is mapped to the $[0, L]$ interval by multiplying by 2 and subtracting L.

$$f(G_r) = \frac{2L}{1 + e^{-k\frac{6(G_r)}{G}}} - L \tag{3.3}$$

And second, it is necessary to find L, which is the maximum value the function will take, and will be used as the size of the first package scheduled by the algorithm. The chosen value is

$\frac{G}{2N}$, which is equivalent to the initial package size commonly used by both the OpenMP Guided algorithm [ope15] or the HGuided using a C value of 2.

$$f(G_r) = \frac{2\frac{G}{2N}}{1 + e^{-k\frac{6(G_r)}{G}}} - \frac{G}{2N} \tag{3.4}$$

To account for the heterogeneity of the system, a correction based on the computing speed of the device is added. The speed s_i is defined as the number of work-groups that device d_i can compute per second. Similarly, the aggregated computing speed of the system is represented by s_H. And given the number of remaining work-groups G_r, the size of the next package for device d_i, using a slope k, is as follows.

$$Package\_size(i, G_r, k) = f(G_r)\frac{s_i}{s_T} = \left(\frac{2\frac{G}{2N}}{1 + e^{-k(6\frac{G_r}{G})}} - \frac{G}{2N}\right)\frac{s_i}{s_T} = \frac{1 - e^{-k(6\frac{G_r}{G})}}{1 + e^{-k(6\frac{G_r}{G})}}\frac{G}{2N}\frac{s_i}{s_T} \tag{3.5}$$

This function is used to obtain the size of the packages, but to avoid excessive overheads the size is not allowed to drop below two lower bounds. How Sigmoid automatically obtains these two values and the slope of the function is explained next.

3.4.3. Automatic parameter tuning

The above expression requires a series of parameters. Some of them are known beforehand, like the number of devices N, the total number of work-groups G or the number of remaining work-groups G_r. Others must be computed and updated as the kernel execution progresses. Such is the case of the computing speed of each device s_i or the slope of the Sigmoid curve. Moreover, a minimum package size has to be selected in order to avoid a large number of small packages at the end of the execution, as they would increase the host-device interaction

overhead and reduce the computing speed of the devices due to their small size. The automatic update of these parameters is what allows Sigmoid to autonomously adapt itself to different kernel behaviours. In this section we will explain how these parameters are obtained.

The computing speed of the devices is used to tailor the amount of work to be distributed to the capabilities of the receiving device. These values can be easily computed at runtime by monitoring the kernel execution. However, computing speeds are kernel dependent. Consequently, for the first packages of a kernel, speed information will not be available, so an approximation is necessary. As an estimation, the nominal GFLOP values reported by the hardware vendors of the devices, are initially used to calculate the relative speed of the devices. These values may not accurately represent the capabilities of the devices for the current kernel, but an approximate speed estimation at the beginning of the kernel execution does not have a large impact on performance. It is at the end of the execution when accurate speeds are required, and by then the algorithm will have refined these throughout the duration of the whole kernel. This is done by measuring the time that each package takes to execute and calculating its speed in work-groups per second. To reduce the influence of work bursts that may not be representative of the behaviour of the whole workload, the average speed of the last three packages scheduled to the device is used to update the speeds. Keeping track of a very long package history would harm adaptiveness. It was experimentally found that three packages strike a balance between adaptiveness and over-sensitivity.

The slope of the Sigmoid curve, represented by k, controls the rate at which the package size decreases and, ultimately, the degree of adaptiveness of the algorithm. As shown in Figure 3.5 a greater k produces a steeper curve, with fewer and bigger package sizes, that limit adaptiveness at the end of the execution. Consequently, regular applications, which do not require adaptiveness, will benefit from a bigger k, which will reduce overheads, while irregular ones will not. Sigmoid manages this by defining two different values for k, one for regular and another one for irregular kernels. To choose adequate values of k for each type of workload, a set of executions of all the kernels used in the evaluation (Section 4.6) has been done using different values. The results of this experiments showed that it is sufficient to use two different k values to achieve good results in both regular and irregular kernels. The values thus selected have

been, $k_i = 0.5$ for irregular applications and $k_r = 2$ for regular ones. The reason for this choice is that these values deliver good overall performance and belong to stable intervals, in which small k differences do not represent great performance variability. Figure 3.6 shows an example of this behaviour for two representative kernels: Mandelbrot as regular, and Ray as irregular. The chosen values for k_i and k_r are expected to provide good performance for other applications. These values were set using a subset of the selected benchmarks, and evaluated using all of them, consequently proving their validity. Nevertheless, for strictly optimum performance, a slight adjustment of these parameters might be necessary when executing other applications.

In order to apply the correct k value it is necessary to determine which type of kernel is being executed. When a kernel is launched, it is regarded as regular until proven otherwise. This avoids penalizing regular kernels and should not affect irregular ones, as adaptiveness is most necessary near the end of the execution. Consequently, packages are initially distributed using k_r. Irregularity is defined by a variability in the time taken to execute two equally-sized chunks of work. Therefore, to switch between k_r and k_i the variability of the computing speed for each device is analyzed. To do so, Sigmoid considers the standard deviation of the speed of the last three packages (σ_{s_i}) on the current device i. If the ratio between this value and the average speed s_i rises above a given threshold d, the kernel is deemed irregular and k_i is used. Note that once a kernel is considered irregular, k_i will be used for the remainder of its execution. This is due to the fact that some irregular kernels may have regions of regular behaviour, in which the standard deviation ratio might drop below the threshold. However, there is no guarantee that irregularity will not be present again near the end of the execution, with few opportunities to react. Therefore using k_r again in an irregular kernel would greatly harm performance. Nevertheless, to verify this hypothesis, tests were carried out using a version of Sigmoid that falls back to k_r if regularity is detected. This version caused an average performance loss of close to 10%.

To adequately set the value for the above mentioned threshold b, an analysis of performance variability in regular and irregular kernels was necessary, as even regular kernels present certain performance differences due to several factors, such as cache effects or contention. To decide the threshold b, the value of σ_{s_i} for each package executed on all the evaluated kernels has been

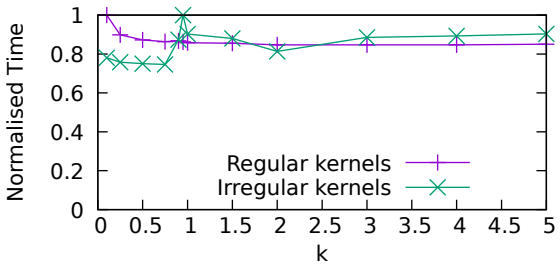


Figure 3.6: Influence of k on regular and irregular kernels.

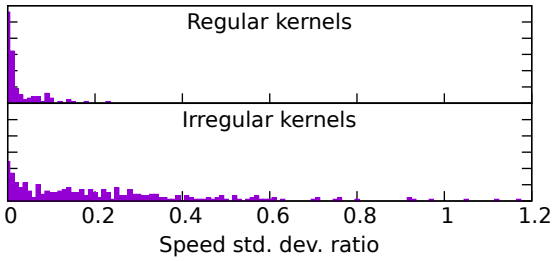


Figure 3.7: Histogram of the number of packages for regular and irregular kernels with respect to the speed variability percentage.

studied. Figure 3.7 shows histograms of the standard deviation ratio $\frac{\sigma_{s_i}}{s_i}$ for regular and irregular kernels. As can be seen, the packages obtained in the regular kernels present a maximum performance variability of around 20%, while differences are much greater for irregular ones. Considering this, b has been set to 0.2 to avoid misidentifying regular applications.

The purpose of applying lower limits to the size of the packages generated by the Sigmoid is twofold. First, it strives to contain the excessive overhead that is inherent in small packages. Second, it guarantees that package sizes do not decrease to a point in which the resources of the devices are not fully used. However, these factors should not be at odds with adaptiveness or induce imbalance in order to keep utilization high. The Sigmoid algorithm uses two lower bounds for the package size, choosing the highest value among the two bounds and the result of equation 3.5 as the size of the package.

Targeting the first mentioned purpose implies a risk: avoiding overheads by increasing package size might generate imbalances, arising from the time difference among the terminations of the last package scheduled to each device. In a worst case scenario, this imbalance might represent the whole execution time associated to the last package. Accounting for this, a maximum imbalance coefficient p is defined. This represents the maximum imbalance that will be generated by the Sigmoid algorithm in the aforementioned worst case scenario. Then, p is used in the following equation, together with the current execution time t and the average device speed s_i, to obtain a minimum package size that limits overheads but does not generate significant imbalance.

$$Minimum\_package\_size = p \cdot t \cdot s_i$$

Guided by the benchmarks used in the evaluation of the algorithms, $p = 0.05$ has been chosen, which does not cause excessive overheads and avoids imbalance. Conceptually, this means that, at the current speed, the execution of a package launched to device i, with a size calculated using the equation, will represent 5% of the current total execution time. Equivalently, in a worst case scenario, at most 5% of the current runtime will be spent in an imbalanced execution, with only one device computing and the rest idling.

The second lower bound for the package size guarantees that the devices are fully used. For GPUs, the algorithm implements the equations of the CUDA Occupancy Calculator, which is part of the CUDA Toolkit since version 4.1. These, take the number of registers and the amount of shared memory required by a kernel, which are values that can be obtained from the OpenCL compiler, and calculate its maximum occupancy and the number of work-groups per multiprocessor required to reach it. The latter value multiplied by the number of multiprocessors in the GPU, which can also be queried from OpenCL, is the minimum number of work-groups to achieve maximum occupancy. CPUs usually show a much more regular performance on the number of work-groups than GPUs, so on CPUs this lower bound is set to one work-group per CPU core. The Sigmoid algorithm is evaluated in detail in Section 4.

Chapter 4

Co-Execution & system abstraction

The focus of this dissertation is on transparent heterogeneous co-execution of massively data-parallel applications. That is, the orchestration of several computing devices, cooperatively working on the same task, with minimum programming effort. As substantiated in Chapter 2, the advent of heterogeneous architectures has been followed by the development of technologies to adequately leverage the outstanding capabilities of these systems. However, these technologies also have certain limitations that turn co-execution into a complex endeavour, in which programmers have to be aware of both the underlying architecture and the nuances of the applications. Most prominently, load balancing lies at the center of the efficient use of a parallel system, and its support in current *de facto* standard heterogeneous programming frameworks is almost non-existent. This chapter introduces these limitations to then present Maat, a new system abstraction and load balancing library focused on co-execution, that provides high level abstractions to manage a complete heterogeneous environment. Maat implements *HGuided* and *Sigmoid*, the two novel load balancing algorithms proposed in Chapter 3, which will be shown to deliver excellent performance and energy efficiency for both regular and irregular workloads.

4.1. Motivation

A computing system is usually defined as heterogeneous if it comprises processing devices of different types. In this dissertation, we will focus on heterogeneous systems comprising CPUs and accelerators. Considering the breadth of this definition and the recent explosion of new accelerators, a wide a array of different system configurations may be considered heterogeneous. Consequently, for a heterogeneous co-execution scheme to be truly useful, it has to be portable, so a change in the underlying system configuration does not represent major modifications on the applications. Moreover, performance should also be portable. This means that an application that delivers competitive performance on a system, should not require significant tuning or modifications to achieve equivalent performance upon migration. This imposes certain restrictions on the approach to the management of the heterogeneous environment:

- *Restriction 1*: All the devices need to execute the same code. Otherwise, several versions of the same algorithm would have to be developed, which is hard to maintain, and new versions would potentially be necessary if new devices were present.
- *Restriction 2*: Devices need to be transparently managed. This prevents the addition/elimination of devices or a system migration from implying changes to the application to account for the new hardware.
- *Restriction 3*: Data has to be distributed among the devices in a transparent manner. This is because memory spaces are very often considered as separated, and several representations of the same data, one for each device, may be necessary. Such complexity should be hidden from the programmer. This is specially true considering that most current heterogeneous systems use discrete accelerators, and those that share memory often incur significant performance degradation.
- *Restriction 4*: Kernels have to be distributed automatically, proportionally balancing their load among all the available devices and minimizing idle times. This enables hardware changes to be acknowledged and new work partitions to be accordingly chosen. Moreover, different kernels have different behaviors, which might even be influenced by

the executing hardware. Load balancing decisions must be abstracted from the programmer for greater portability and improved performance and efficiency.

The most commonly used heterogeneous programming frameworks, namely OpenCL [GHK+11] and CUDA [KH10a], fail to correctly address some of these restrictions. CUDA, being proprietary software, only supports the execution of kernels on NVIDIA GPUs, so not even Restriction 1 is met. OpenCL was designed with a focus on portability, so kernels can be executed on any device, assuming an adequate OpenCL driver is provided by the hardware vendor. Nevertheless, OpenCL does not meet restrictions 2 through 4. It is a powerful framework that offers all the necessary tools to manage a heterogeneous system, but it does not provide the programmer with any assistance to face the task. Devices are managed individually, so the programmer needs to tailor the code of the application to the available hardware, and both data transfers and kernel launches have to be explicitly specified for each of the available devices. Moreover, the programmer is also held accountable for making load balancing decisions, choosing the portions of the workload to be computed by each device and their size, which will ultimately determine the success of co-execution.

OpenCL indeed offers the basic capabilities for co-execution, but it is certainly neither transparent, nor effortless, producing hard to maintain applications and requiring a significant level of expertise from the programmer. For this reason, this dissertation presents Maat, a library, built upon OpenCL, that enables effortless co-execution. It lifts Restrictions 2 and 3 via higher level abstractions, and Restriction 4 by providing two original load balancing algorithms designed for heterogeneous co-execution. The latter will be introduced next, to then delve into Maat and the abstractions that conform it.

4.2. Overview of Maat

Maat is a new OpenCL-based load balancing and system abstraction library, designed with co-execution in mind. Its name comes from that of the Egyptian goddess of justice and balance, as its main goal is enabling the transparent, efficient use of all the resources available in a

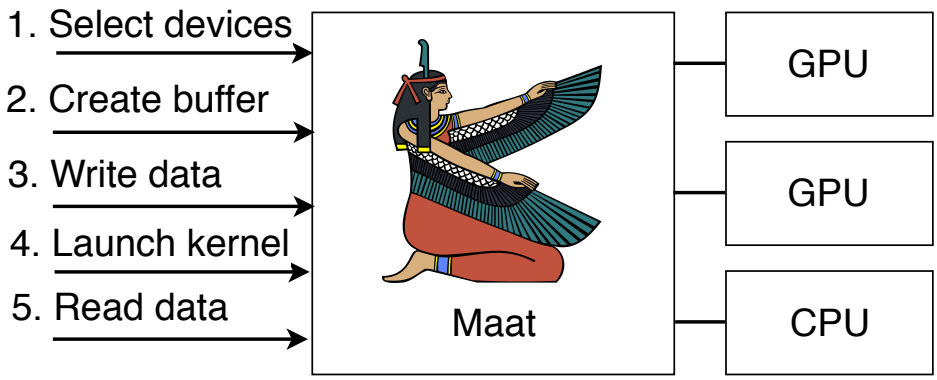


Figure 4.1: High level description of the operation of Maat.

heterogeneous environment. A very high level outline of how Maat is used is shown in Figure 4.1. Maat acts as an intermediate layer between the programmer and the devices, providing an abstract view of the complete system. As a result, the whole aggregated computing power of all the available hardware can be accessed using a single interface, with similar complexity to using a single device. To achieve it, Maat extends the definitions of OpenCL data structures and functions, so the basic flow of an OpenCL program is preserved, but the hassle of dealing with multiple, device-dependent, data structures is eliminated. The resulting code is completely independent from the underlying system, as all the available resources are managed as if a single all-encompassing device was available, making programming easier. This includes the most important part of co-execution, which is load balancing. The launch of a kernel using Maat, results in its transparent distribution among all the available devices, requiring no programmer action and efficiently using the system. Workload partitioning is automatically performed using one of the load balancing algorithms offered by Maat, which have been specially designed for heterogeneous co-execution.

4.3. Design of Maat

The development of a library for the effortless management of a complete heterogeneous system, using an OpenCL-like syntax, involves extending some of the basic concepts proposed by OpenCL. This requires certain design principles to be followed in order to properly address the needs of heterogeneous architectures.

- *Ease of adoption.* For an easier programming, the new library should adhere to the structure and philosophy of an OpenCL program as much as possible. This is to ensure that porting OpenCL applications to the new library does not represent important coding efforts.

- *Transparency.* The management of the system should be handled in a manner that prevents the programmer from having to be aware of the underlying details of the architecture, which should be hidden by higher level abstractions.

- *Portability.* Once transparency frees the programmer from the hardware, it also enables applications to be portable, allowing for easy migration.

- *Performance, energy consumption and efficiency.* The introduced abstractions should not be at odds with excellent performance. This is achieved by adequately managing the resources, accurately balancing the load and *overlapping* computation and communication as much as possible. This allows devices to spend as much time as possible performing useful work.

To fulfill these principles, Maat relies on three definitions that allow the resources of an entire heterogeneous system to be collectively handled through a single interface, regardless of the underlying hardware. This entails redefining how the programmer deals with the available devices, how work is launched and how data transfers are performed. This Section will be devoted to these three areas, introducing the new concepts that build Maat.

4.3.1. Device Abstraction

The most important concept that OpenCL hardware abstraction is built upon is the context (please, refer to Section 2.1.1 for basic OpenCL concepts). It is the main tool to manage the heterogeneous system, as the majority of the rest of data structures required to transfer data to the devices or offload work to them, depend on the context. However, it is also a limiting concept when co-execution comes into play. Contexts can only hold devices belonging to a

single platform and, as introduced in Chapter 2.1.1, a system may have an arbitrary number of platforms. To account for this limitation, we define the *SuperContext*.

Definition 4.1 *A SuperContext is a combination of devices available in the system, either belonging to a platform or to several.*

Using this abstraction, the programmer can transparently manage any combination of devices, improving portability, as no code change will be necessary to account for hardware changes. Figure 4.2 shows a sample system, the contexts that OpenCL would provide and some of those offered by Maat. The system comprises 3 devices, a CPU from a vendor and 2 GPUs from another, 2 platforms being consequently available. In this example, OpenCL would not be able to provide access to a context comprising the CPU and any of the GPUs. However, SuperContexts offer a single means to handle any combination of the available devices. Please, note that Maat can also provide SuperContexts holding devices from a single platform, such as one holding just the two GPUs in the example, so they can benefit from the advantages of transparent co-execution. This kind of SuperContexts have been omitted from the figure for clarity.

OpenCL also defines command queues as the data structure used to communicate with independent devices. However, the focus of Maat is on the transparent orchestration of several devices, so the programmer does not need to worry about the underlying system. Consequently, Maat internally manages the necessary command queues for good performance and communication-computation overlapping, as required by the executing hardware configuration.

4.3.2. Execution Abstraction

The work that will be offloaded to the devices in OpenCL is expressed in the form of kernels, which in turn are represented in the host by data structures also called kernels. However, these need to be compiled for each device that will execute them. Thus, they are not independent, as several instances of a kernel data structure, that ultimately stand for the same algorithm, may be necessary for co-execution. To avoid this, Maat defines the *SuperKernel*.

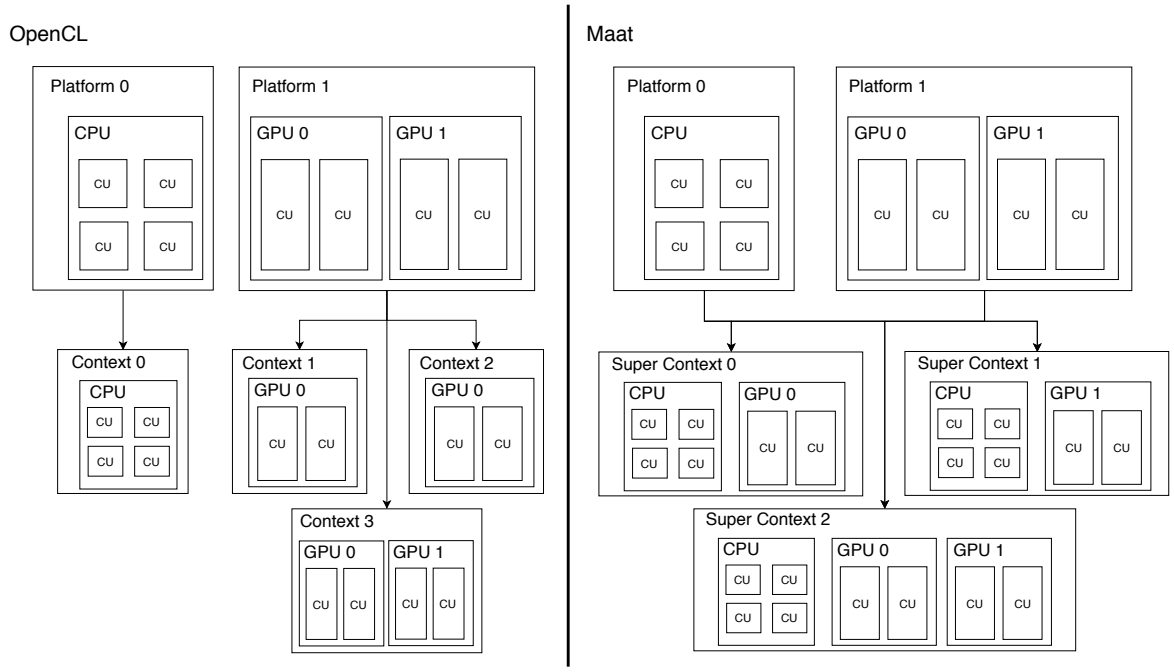


Figure 4.2: Possible contexts and SuperContexts in a sample system. Redundant SuperContexts omitted.

Definition 4.2 *A SuperKernel is a data structure representing the code of a kernel, that is valid across all the devices encompassed in a SuperContext.*

That is, valid across all the devices of a heterogeneous system if desired. To ease programming, upon creation of a SuperKernel, the kernel code it represents is automatically compiled for all the devices available in the SuperContext. This compilation is performed online, as the actual underlying hardware needs to be known to perform it.

For a greater ease of adoption, SuperKernels are launched using functions that follow a similar format to OpenCL kernel offload functions. They receive the number of work-items to be launched, the size of each work-group and the offset as parameters. However, they transparently use all the devices available in the associated SuperContext via co-execution, implementing several different load balancing algorithms and internally managing the scheduling of work. Chapter 3 presents a detailed description of the load balancing algorithms provided by Maat.

4.3.3. Memory Abstraction

OpenCL buffers guarantee portability by abstracting the data management from actual devices. Nevertheless, they just represent valid references on the memory of a certain device, offering no support for co-execution. To truly enable the transparent management of a heterogeneous system capable of co-execution, Maat offers the *SuperBuffer*.

Definition 4.3 *A SuperBuffer is a reference to a piece of data that is valid across all the devices belonging to the SuperContext.*

The creation of a SuperBuffer represents the transparent creation of as may buffers as required to operate with the available devices. To manage the distribution and gathering of results, buffers can be defined as *In* or *Out* buffers.

To communicate data to the devices, Maat offers mechanisms to write and read SuperBuffers, while the actual transfers to/from the individual underlying buffers are automatically performed and overlapped with computation when possible. Writes can be performed using two functions. Function `clWriteSuperBuffer` replicates the data of the SuperBuffer to all the devices. This function can be used with every kernel, but may perform unnecessary transfers. On the contrary, `clDelayedWriteSuperBuffer` avoids this by assuming that each device will only require a certain part of data. This mechanism has smaller overheads, but cannot be used for every kernel. It is the responsibility of the programmer to decide when it should be used. For example, every work-item of an application such as NBody will require an undetermined amount of data for its computation, because the number of neighbouring particles that will have an impact on its computation is unknown. Consequently, NBody would require all the data to be replicated. On the other hand, a blocking implementation of the matrix multiplication would allow for the use of the `clDelayedWriteSuperBuffer` for one of the matrices.

Regarding data reads, they are also automatically managed by Maat, so Out buffers are kept consistent at SuperKernel boundaries. The programmer just has to specify the location of the data in the memory of the host and Maat will handle the gathering of the results correctly.

To alleviate the overheads of data transfers and improve performance, these are automatically performed asynchronously. As a consequence, transfers and computation are overlapped as much as possible. This effectively hides communication times, so the devices are working on the computation associated to kernels for a greater fraction of the execution time. Moreover, Maat is also capable of handling the automatic redistribution of data for kernels that are called iteratively. That is, results are not only transparently transferred to the Host, but also to the other devices, so they are ready for the next kernel execution as soon as the current one finishes. The implementation of this features is explained in detail in Section 4.4.2.

4.3.4. Summary

By means of the abstractions explained above, Maat hides the complexity of managing heterogeneous systems from the programmer and offers the necessary capabilities for effortless co-execution. Table 4.1 shows the functions provided by Maat and their OpenCL equivalents. There is a close correlation between Maat and OpenCL functions. This is to retain an OpenCL-like approach to heterogeneous programming and increase the ease of adoption. By using Maat, a complete heterogeneous system can be used for co-execution, without the hassle of managing redundant, device-dependent, data structures. The result is a code that is portable easier to maintain, not requiring modifications if the system configuration changes. This is shown in Figure 4.3, which is a simplified representation of the data structures necessary to execute a simple kernel that uses two arrays. The arrows represent the relations of belonging between the different layers. That is, the OpenCL data structures managed by a Maat abstraction or their correspondence to actual devices. Using Maat, a SuperContext (SC1) is enough to manage every device, a single SuperBuffer is necessary for each array (SB1 and SB2) and just one SuperKernel (SK) is used to represent the implemented algorithm. On the contrary, OpenCL requires two contexts (C1 and C2) and several, device-dependent buffers and kernels. Certain OpenCL data structures, such as command queues and OpenCL programs have been omitted for clarity. These are transparently managed through the abstractions provided by Maat. No Maat data structure has been omitted. The simple co-execution shown in Figure 4.3, using

Maat	OpenCL
clCreateSuperContext	clCreateDevice clCreateSubDevices clCreateCommandQueue clCreateContext
clCreateSuperBufferIn clCreateSuperBufferOut	clCreateBuffer
clWriteSuperBuffer clDelayedWriteSuperBuffer	clEnqueueWriteBuffer
clReadSuperBuffer	clEnqueueReadBuffer
clEnableResultAutoDistribution	
clCreateSuperKernel	clCreateProgram clBuildProgram clCreateKernel
clSetSuperKernelArg clSetSuperKernelArgSuperBuffer	clSetKernelArg
clEnqueueSuperKernelHeterogeneousGuidedBalancing clEnqueueSuperKernelSigmoiddBalancing	clEnqueueNDRange
clReleaseSuperContext	clReleaseDevice clReleaseCommandQueue clReleaseContext
clReleaseSuperBuffer	clReleaseBuffer
clReleaseSuperKernel	clReleaseKernel

Table 4.1: Maat and OpenCL functions

OpenCL, would require around 800 lines of code. The same example using Maat requires only 500 lines, which, moreover, are independent from the underlying hardware configuration. This is because, if programming pure OpenCL, the Maat layer would not be present, which would leave the programmer alone with the responsibility of managing several representations of the same elements of the program logic. Maat keeps all this complexity under the hood.

4.4. Implementation

OpenCL is a portable heterogeneous programming framework. For this reason, it has been used as a basis for the abstractions introduced in Section 4.3. However, an operation in Maat usually represents multiple OpenCL operations that are internally performed. This section explains how Maat was implemented, focusing on how the different data structures are managed and how work and data are automatically distributed among the available devices at runtime.

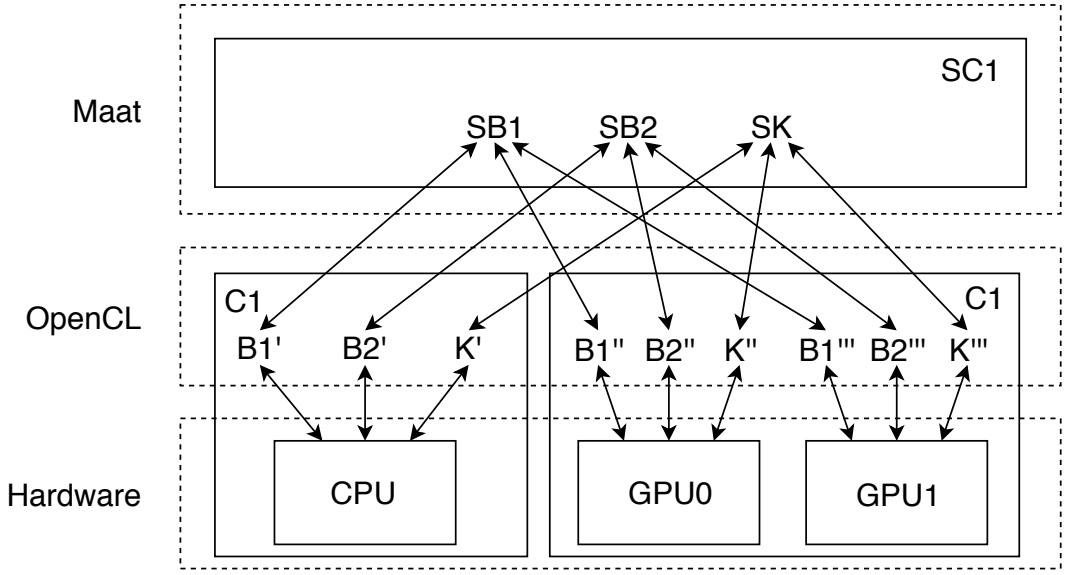


Figure 4.3: Underlying OpenCL data structures transparently managed by Maat in a sample system.

4.4.1. Data structure management

The SuperContext, SuperBuffer and SuperKernel are defined as abstractions that extend the definitions of the basic OpenCL data structures, which they are built upon. For instance, the creation of a SuperContext represents the internal creation of as many OpenCL contexts and command queues as necessary to manage the specified combination of devices. Any operation using a SuperContext will target the underlying contexts and command queues to enable co-execution, keeping all the complexity under the hood. The creation of a SuperKernel will also translate in the creation of the necessary kernels to account for the available devices. These are individually targeted correspondingly when parameters are associated to the SuperKernel. Likewise, when a SuperKernel is launched, it will generate as many kernel launches to their individual devices as required by the selected load balancing algorithm, which are managed at runtime, as explained in Chapter 3.

Lastly, the creation of a SuperBuffer triggers the creation of an individual buffer for each device too. Theoretically, OpenCL supports buffer sharing among devices belonging to the same platform. However, this was tested in a preliminary version of Maat and significant performance degradation was detected when sharing buffers. Writing a SuperBuffer represents buffer writes to the individual OpenCL buffers managed by Maat. The size of these writes

will depend on the chosen write function. The `clWriteSuperBuffer` function will copy all the data associated to a SuperBuffer to each of the internal buffers. On the contrary, the `clDelayedWriteSuperBuffer` function will only write to each buffer the data required by its owning device. To do so, the data is stored, and actual writes are performed when packages are launched, assuming that each device will require the portions of data linearly identified by the indices of the work-groups of the package. Consequently, this function should only be used for kernels known to fit this computing pattern.

Buffer reads have been implemented as a dummy function. This is because Maat does not keep track of the device that computes for each part of the work, but performs transfers as soon as a device completes a portion of the results. Consequently, all the results are available once a SuperKernel completes its execution, which also partially hides the overhead of data transfers. The completion of a package will trigger a read operation to the corresponding device. Reads assume that each work-group will produce the part of the results identified by its index, which is, by far, the most common approach to heterogeneous programming. This accounts for the fact that predicting the work performed by each work-group is an open research topic, with known general solutions being costly and ultimately impractical due to their overheads. How reads are automatically performed, hiding overheads, is explained in detail in Section 4.4.2.

4.4.2. Runtime capabilities

For the operation of its load balancing algorithms, Maat relies on the concept of a package, which is a set of consecutive work-groups belonging to a kernel. To dynamically launch them and orchestrate the co-execution of several different devices in general, Maat uses on OpenCL *callbacks*. These are functions defined by the programmer that are executed on the occurrence of a particular event, such as the termination of a kernel or the completion of a data transfer. According to the OpenCL specification [Ope18], callback functions must return promptly, avoid expensive functions or system routines and be thread-safe, as OpenCL uses a single thread for management and the callback may be interrupted. Figure 4.4 depicts the threads and callback functions used in Maat, which mainly handle the distribution of the load represented by a

SuperKernel and the management of the data held in SuperBuffers.

When a super kernel is executed, a package is launched to each of the devices, and a special *management thread* is spawned. This is depicted in the *Launch super kernel* portion of Figure 4.4. The thread will deal with the extra work related to the movement of data and the launching of additional packages. A thread-safe *Completion callback* function is also registered to the event signaling the completion of each of the packages. This function just stores the information of the packages that were completed and notifies the management thread.

The *Management thread* is in charge of:

- Updating the parameters of the load balancing algorithms if necessary.
- Reading the results of the computation of the finished packages and updating the host memory.
- Launching new packages, if more work has to be done, and writing their input data if needed. This step is necessary for dynamic load balancing schemes. A static one would skip it.
- Notifying super kernel completion.

Using a single thread is sufficient to perform these tasks, as only a small number of devices will be managed (tens at most). Moreover, the implementation of the management thread uses non-blocking operations exclusively, so the work that has to be done per completed package is not time consuming.

Support for non-blocking communication

Non-blocking calls also allow for computation and data transfers to be performed in parallel, which effectively hides the impact of communication and improves performance. This is possible because of the approach to computing favored by OpenCL. This standard considers work-groups as independent execution units, forbidding communication and synchronization between them.

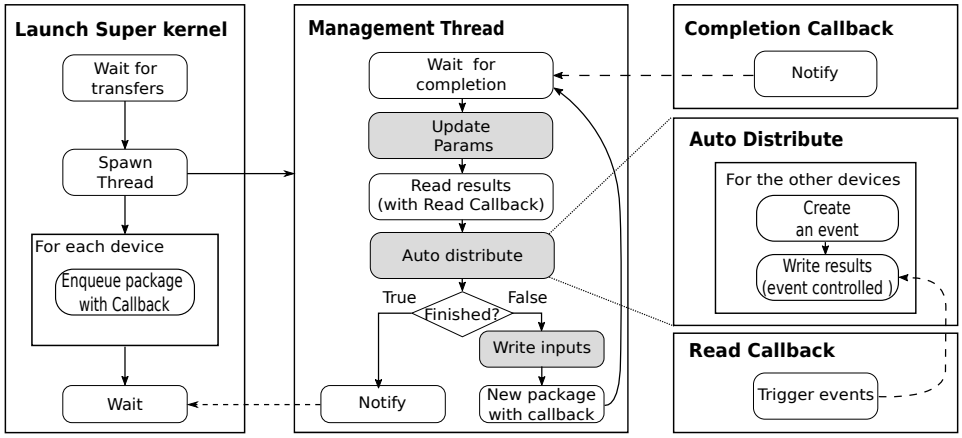


Figure 4.4: High level representation of the operation of Maat. Grey boxes represent optional operations.

Therefore, the output data written by a work-group, will necessarily be independent from that written by others. Consequently, once a package has finished executing, the results associated to its work-groups can be safely read, as they will not be modified.

However, the definition of command queues in the OpenCL standard has to be considered in order to take advantage of computation/communication overlapping. There are two types of queues: in order and out of order. The former force commands to wait until all the previous ones have completed, while commands do not necessarily have to wait for each other in the latter. Consequently, overlapping is only possible when using an out of order queue. However, they are not supported in every OpenCL implementation. To enable overlapping communication and computation regardless of how the underlying drivers implement the standard, Maat internally manages two independent in-order command queues per device: the general queue and the result queue. The management thread enqueues result reads on the result queue, while new package launches use the general queue. Thus, the overhead is hidden and communication does not block the progress of computation. To guarantee that a device does not start working until its input data is ready, input writes are enqueued to the general queue, so any later kernel launch has to wait for its completion. An example of the operation of the queues is shown in Figure 4.5, which shows the steps that are followed when the computation for package X finishes. Note how reads do not block the progress of packages. As a result of this dual queue implementation, the management thread has to check for three conditions to notify the termination of a SuperKernel: that there are no more packages to launch and that both, the

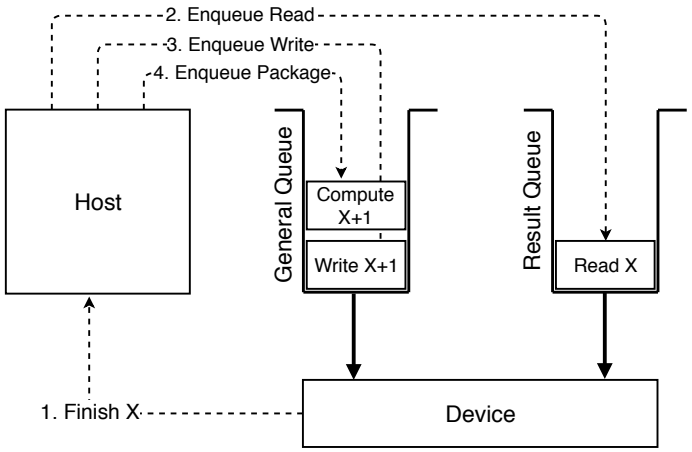


Figure 4.5: Representation of the dual queue operation of Maat.

general and result queues, are empty.

Support for iteratively launched kernels

Some common parallel programming algorithms use iterative methods. When applied to heterogeneous systems, this programming pattern often encapsulates a whole iteration into a single kernel call, which is repeatedly enqueued. As a result, the outputs of the launch of a kernel are also the inputs of a later execution. Such algorithms usually rely on the data already being stored in the device memory, so transfers are not required between iterations. However, when working with several devices, each of them will only produce a part of the output data, which may be required by the rest in the next iteration. Considering that most current heterogeneous systems have independent memory spaces, extra steps need to be taken to ensure that every device has the input data it requires. This situation is shown in Figure 4.6, which depicts how results may be split among several devices and need to be gathered before the next iteration begins.

One approach would be to preserve locality and avoid copies by launching the packages corresponding to the data already held in the memory of each of the devices. However, this is not possible for two reasons. First, the parts of the input data required by each work-group cannot be predicted in general. And second, the irregularity of an application is defined by the input data it uses, so the behaviour between different launches of the same kernel may differ

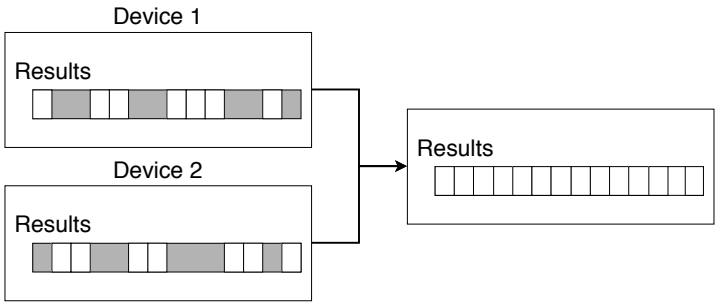


Figure 4.6: Representation of the partial results produced by two devices at the completion of a kernel.

greatly, rendering a work distribution based on the data obtained by each device in the previous iteration unprofitable.

The overhead of distributing the results of a previous SuperKernel launch among the devices is thus unavoidable. Nevertheless, to mitigate its impact and keep the illusion of a single, all-encompassing heterogeneous device, Maat performs the distribution of data automatically. When a package completes its execution, the management thread, apart from reading its results, copies them to the memories of the rest of the devices. To keep the management thread running and improve performance, these transfers are also non-blocking and performed using the data command queues.

However, OpenCL only supports direct transfers between devices that belong to the same platform. Considering that devices will often belong to different platforms, the transfer through the host is unavoidable. Maat transparently handles these transfers in the management thread. First, a non-blocking read from the producing device to the host is enqueued, followed by non-blocking writes to the Result queues of the rest of the devices. To guarantee that the writes do not start until the data is available in the host, these are controlled by OpenCL events as shown in the portion *Auto Distribute* of Figure 4.4. These events will be triggered, enabling the execution of the corresponding write, by the *Read callback*, which was associated to the aforementioned read. Automatic result distribution is activated using a function offered by the library. This whole process is overlapped with computation for every package but the last, so minimum communication overhead is introduced.

4.5. Methodology

The purpose of this section is introducing the aspects of the methodology particular to this chapter that were not presented in Section 1.5. This includes load balancing algorithms implemented for comparison with HGuided and Sigmoid and the applications used in the evaluation.

4.5.1. Reference algorithms

In order to evaluate the HGuided and Sigmoid algorithms, the two well-known load balancing algorithms were considered in the experiments. An uninformed version of the HGuided, using fixed parameters, was also implemented to proof the advantages of the Sigmoid algorithm at a comparable effort level.

Static algorithm [PBB16] This classic algorithm divides the kernel in as many packages as devices are available in the system. The size of each package is proportional to the relative computing speed of the device that will execute it. This algorithm minimizes the overhead, since only one package is sent to each device. For this reason, it is the a priori best choice for regular kernels. However, it is an informed algorithm, that requires the computing speed of the devices as input parameters, and it performs badly for irregular kernels.

Adaptive algorithm [BSCJ13] This algorithm was proposed for a two device scenario as a dynamic technique that requires no training and responds automatically to performance variability. The implementation used in this dissertation is an extension of the original algorithm for an arbitrary number of devices introduced in [PBB16]. The Adaptive algorithm proceeds by first dynamically launching γ small probe packages to each device, starting with a size representing β percent of the total workload and using a growth rate of δ, and then using the obtained execution times and the following formula to predict an ideal static work partitioning for the remaining work.

$$W_i = \frac{\prod_{\substack{j=1 \\ j\neq i}}^{N} \Omega_j \cdot W_r - \sum_{\substack{j=1 \\ j\neq i}}^{N} \left[(\lambda_i - \lambda_j) \cdot \prod_{\substack{k=1 \\ k\neq i \\ k\neq j}}^{N} \Omega_k \right]}{\sum_{j=1}^{N} \prod_{\substack{k=1 \\ k\neq j}}^{N} \Omega_k}$$

W_i is the number of work-groups that will be scheduled to device i, Ω_j is the execution time of the last package completed by device j, divided by the number of work-groups in that package, λ_i is an estimation of the remaining time before the package currently running in device i finishes and W_r is the number of remaining work-groups. The estimation for λ_i is obtained by using the size of the package and the value of Ω_i. This is, it is estimated using the computing speed of the previous packages. The amount of probe packages per device, their size and growth rate are programmer defined parameters. Parameters γ, β and δ may be defined by the programmer to better suit the workload to co-execute. However, the authors of [BSCJ13] suggest $\beta = 7\%$, $\gamma = 2$ and $\delta = 1.5x$ as default values that deliver good performance. The implementation offered in Maat initially used these parameters. However, it was found that the suggested size for the first probe package of each device (β) is excessive when more than two devices are considered or when the devices show significant computing speed differences. Consequently, a value of $\beta = 2\%$ has been used. Similar algorithms in the bibliography are [NVCA14, VAN+15, BBG13].

Default HGuided algorithm [PBB16] This algorithm, labelled as *DefHG* represents the effortless usage of the HGuided algorithm. It is used for reference, to evaluate two aspects of the algorithms proposed in Section 3. First, the improvement that the HGuided algorithm achieves when a careful parameter sweep is performed to set the parameter values. Second, to illustrate the advantages of the Sigmoid algorithm at an equivalent effort level. To be fair, the same parameters initially used by Sigmoid have been selected for DefHG: the nominal values for the computing speeds, and the values reported by the CUDA occupancy calculator for the minimum package size.

Table 4.2: Parameters for each benchmark

Benchmark	Type	Problem size	Local work size	GPU Speed	# of packages	Min. size
Binomial	Regular	2048000	256	7.28	30	380
Gaussian	Regular	8000 × 8000 81 × 81	128	13.77	30	1000
Mandelbrot	Regular	20480 × 20480	256	5.88	30	400
NBody	Regular	51200	128	7.33	10	400
Taylor	Regular	800 × 800	128	2.06	35	280
Aho	Irregular	1536000	64	8.2	80	200
BM3D	Irregular	800 × 800	64	2.28	35	150
Rap	Irregular	1024 × 1024	64	4.26	80	400
Ray	Irregular	12000 × 12000	64	7.7	125	380

4.5.2. Applications

Nine kernels have been chosen for the experiments, five of which exhibit regular behaviour. Binomial generates binomial lattices, useful for option pricing in financial software. Mandelbrot implements a blocked algorithm to compute a Mandelbrot set. NBody simulates a dynamic system of particles, used in many physics applications. Gaussian calculates the Gaussian blur of an image, commonly found in image and video processing software. The last regular kernel is Taylor, which performs a bi-dimensional Taylor approximation for a set of points. The other four kernels are irregular. Aho is an implementation of the Parallel Failureless Aho-Corasick (PFAC) string matching algorithm, commonly used for protein sequencing [LLCC13]. BM3D implements one of the filters of the BM3D image denoising algorithm [DFKE07]. Rap is an implementation of the Resource Allocation Problem [ACBA10]. There is a certain pattern in the irregularity of RAP, because each successive package represents a bigger amount of work than the previous. Finally, Ray Tracing renders realistic images by calculating the light that reaches each pixel by modeling light rays. Two different scenes of similar complexity but with different object distribution, (Ray1 and Ray2), have been defined. It will be shown later (Section 4.6) that changing the input data, the behaviour of the application varies wildly.

Each kernel has been run using a problem size big enough to justify its distribution among all the available devices. The local work size has been set so the performance of the fastest device, namely the GPU, is maximized. The reason for this is that almost no performance difference was detected when varying the local work size for the CPU. The selected values for each kernel

are shown in Table 4.2. This table also shows the parameters of the Static, Dynamic and BestHG algorithms used in the experimentation.

4.6. Evaluation

This section presents the experimental results obtained on the test systems when running the different benchmarks, as described in Section 4.5. These experiments aim to answer the following questions:

- How well does Sigmoid balance the workload across different heterogeneous devices?
- What is the performance of Sigmoid for both regular and irregular kernels?
- Is well-balanced co-execution capable of improving the energy consumption of a heterogeneous system?
- How does Sigmoid scale when the number of devices increases?

Each of the following sections answer one of the aforementioned questions, comparing the results achieved for Sigmoid with other load balancing algorithms.

4.6.1. Load Balance

The first metric considered in this analysis is the *Load Balance* which is shown in Figure 4.7, for *Batel*. For a given execution, it is defined as the ratio of the response time of the first device to conclude its work and that of the last. The ideal value for this metric is one, meaning that all devices finished simultaneously and the maximum utilization of the system was reached.

Sigmoid reaches perfect load balance in six out of the ten benchmarks. Compared to the other algorithms, it obtains the best load balance, except in Rap and NBody where it is slightly worse than BestHG and DefHG. Looking at the geometric mean, the Sigmoid algorithm boasts almost perfect load balance (0.97) closely followed by BestHG (0.94). Recall that the parameters of

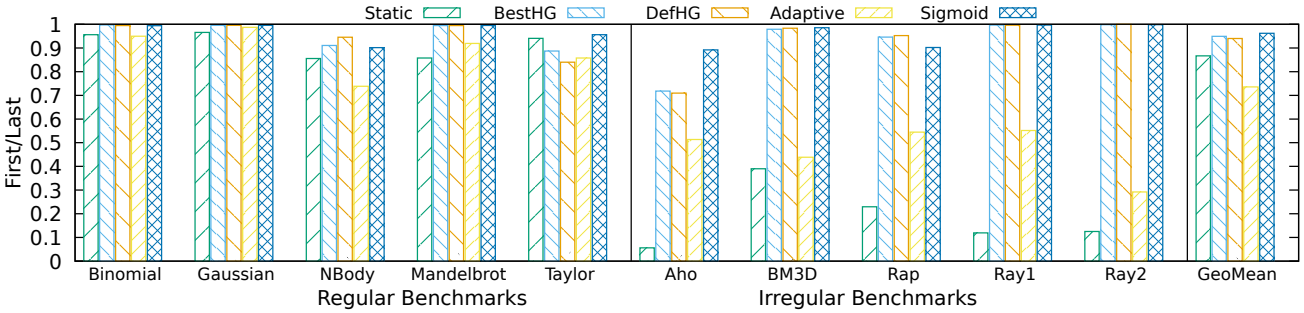


Figure 4.7: Load balance of each device for all algorithms and benchmarks in the heterogeneous system.

BestHG are optimal, obtained from a time-consuming sweep. Regarding the other algorithms, Static performs well in regular benchmarks but, as is expected, performs poorly in irregular ones. This is a consequence of the nature of these kernels, that make it very difficult to devise a fair load distribution before the actual execution. In the same way, Adaptive shows good results for regular kernels (except NBody) but no so satisfactory for irregular ones.

4.6.2. Performance

To give an idea of performance, Figure 4.8 shows the speedups reached by the different benchmarks in the *Batel* system, compared to the baseline scenario that only uses one GPU. The test system is composed by $N = 3$ devices, but since they do not have the same computing power, the speedup is never going to reach 3. Table 4.3 summarizes the maximum speedup S_{max} that can be obtained with each benchmark. These values were derived from the response time T_i of each device as shown in Equation 4.1, and are also represented in Figure 4.8 as a horizontal lines above the bars of each benchmark.

$$S_{max} = \frac{1}{max_{i=1}^{N}\{T_i\}} \sum_{i=1}^{N} T_i \tag{4.1}$$

Table 4.3: Maximum speedup for the different benchmarks

Benchmark	Binomial	Gaussian	Mandelbrot	Nbody	Taylor	Aho	BM3D	RAP	RAY
Max. Speedup	2.14	2.07	2.17	2.13	2.48	2.12	2.44	2.23	2.13

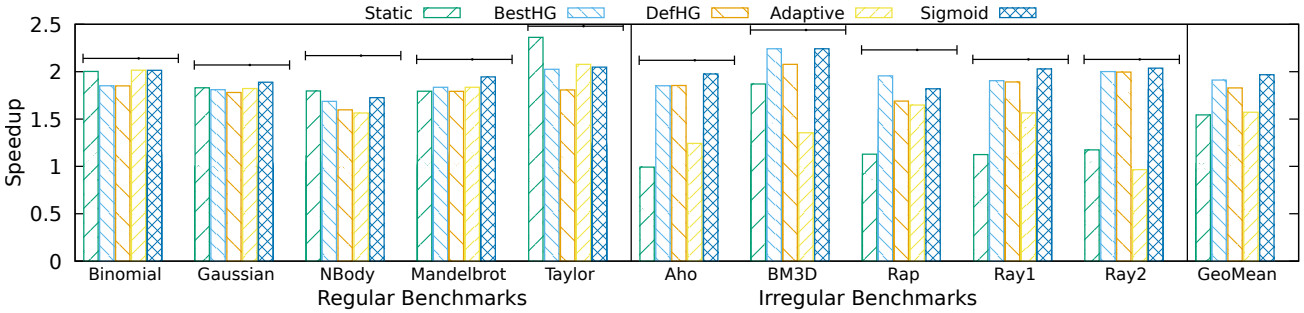


Figure 4.8: Speedups of the benchmarks with the different algorithms in the heterogeneous system.

Looking at the geometric mean of the speedups shown in Figure 4.8, it can be seen that the Sigmoid algorithm gives the best performance. It is 22% better than Static and 3% better than BestHG. Regarding the mean for regular and irregular kernels separately (not depicted), Sigmoid obtains the same performance (99%) as the Static for regular benchmarks and is even slightly better than BestHG for irregular. In short Sigmoid delivers the best overall performance and also equals the performance of the best alternative for both regular and irregular workloads. When compared to the speedup of the other effortless algorithms, Sigmoid also excels. It is 20% better than Adaptive and 7% better than DefHG.

Regarding each benchmark individually, Sigmoid gives the best performance in all except NBody, Taylor and Rap. Despite that Sigmoid attains the best load balance results in NBody and Taylor, using the optimal parameters with Static obtains a better speedup. Since these benchmarks have a very low computation-communication ratio, the overhead increases when the workload is subdivided in more packages than devices. With Rap, Sigmoid delivers the second best performance. This is because the minimum package size that guarantees efficient device use, generates a slight imbalance at the end of the execution. Regarding the effortless algorithms, Sigmoid delivers the best performance in all the applications but Taylor, in which it is only marginally surpassed by Adaptive.

The gap between the measured and the theoretical maximum values is a consequence of the extra communication overhead that comes from having more than one device. This is more notorious in applications in which the data can not be divided and must be replicated (NBody) or when the ratio between the computation and communication times is small (Mandelbrot,

RAP).

As discussed above, one of the advantages of Sigmoid is that it tries to minimize the number of packages, as each implies interaction between the host and a device, while maintaining adaptiveness. This can be seen in Table 4.4, which depicts the number of packages generated by each algorithm excluding Static, which would always generate as many packages as devices. Adaptive produces almost the same amount of packages for all benchmarks. This translates into good results in very regular benchmarks, as overheads are reduced. However, it fails in irregular ones, to which it cannot adapt. As for the HGuided algorithms, both versions generate a huge amount of packages, many more than the rest, although slightly less in BestHG thanks to the tuning of the parameters. This occurs even in regular benchmarks, like Binomial, which do not take advantage of adaptiveness. This causes two damaging effects. On the one hand, it notably increases overheads. On the other hand, a large number of packages are excessively small and do not fully take advantage of the capacity of GPUs. Finally, it should be noted that Sigmoid generates a much smaller number of packages, thus reducing the overhead with respect to HGuided. At the same time, it maintains adaptiveness according to the needs of each benchmark, surpassing Adaptive in this regard.

Table 4.4: Number of Packages generated by each load balancing algorithm and benchmark

Benchmark	Binomial	Gaussian	Mandelbrot	Nbody	Taylor	Aho	BM3D	RAP	Ray1	Ray2	Average
Adaptive	15	19	13	15	15	11	13	15	14	13	**14.30**
DefHG	445	105	25	175	51	828	92	79	389	291	**248**
BestHG	307	84	14	119	31	707	47	46	288	185	**182.80**
Sigmoid	28	31	17	27	20	13	28	20	40	32	**25.60**

4.6.3. Energy consumption

Nowadays, performance is not the only figure of merit used to evaluate computing systems. Their energy consumption and efficiency are also very important. Figure 5.10 gives an idea of the energy saving obtained by taking full advantage of all the compute devices in the *Batel* heterogeneous system. Contrasting with the baseline system that only uses one GPU, while the other devices are idle but still consuming. Therefore, the figure shows, for each benchmark, the energy consumption of each algorithm normalized to the baseline consumption. In this graph,

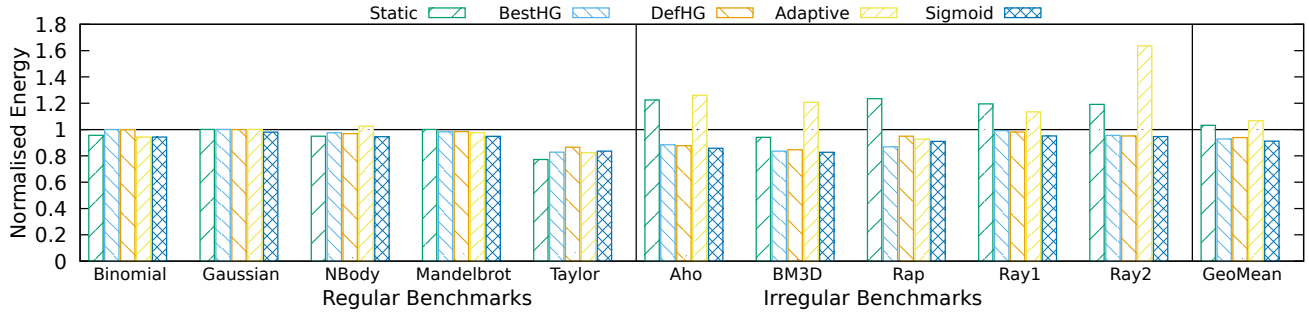


Figure 4.9: Energy consumption of the benchmarks with the different algorithms normalized to the baseline in the heterogeneous system.

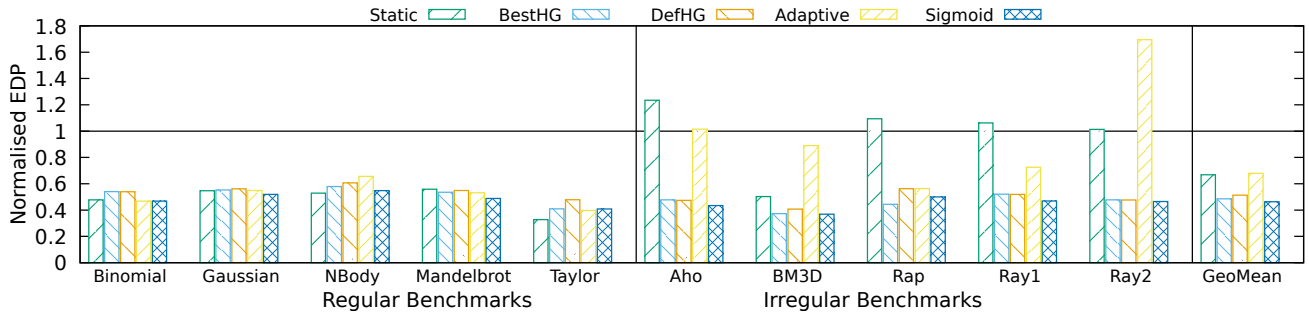


Figure 4.10: EDP of the benchmarks with the different algorithms normalized to the baseline in the heterogeneous system.

less is better, and bars over one indicate that the whole heterogeneous system consumes more energy than the baseline.

The energy measurements are strongly correlated to the performance of the algorithms. Observing the geometric mean, it can be seen that Sigmoid gives the best results, followed by BestHG, presenting energy savings of 9% and 7% respectively. Looking closely at some benchmarks, the other algorithms can consume significantly more energy than the baseline (Static and Adaptive in irregular benchmarks). Even DefHG and BestHG do not reach any improvement in Binomial and Gaussian. Interestingly, the only algorithm that always improves the baseline consumption is Sigmoid. The use of more devices logically increases the instantaneous power at any time. But, since the total execution time is reduced, the total energy consumption is also less. This saving is further improved by the fact that idle devices still consume energy, so making all the devices contribute work is beneficial. Notice that, of the effortless algorithms, Sigmoid attains the lowest energy consumption while Adaptive presents and overall energy consumption greater than the baseline.

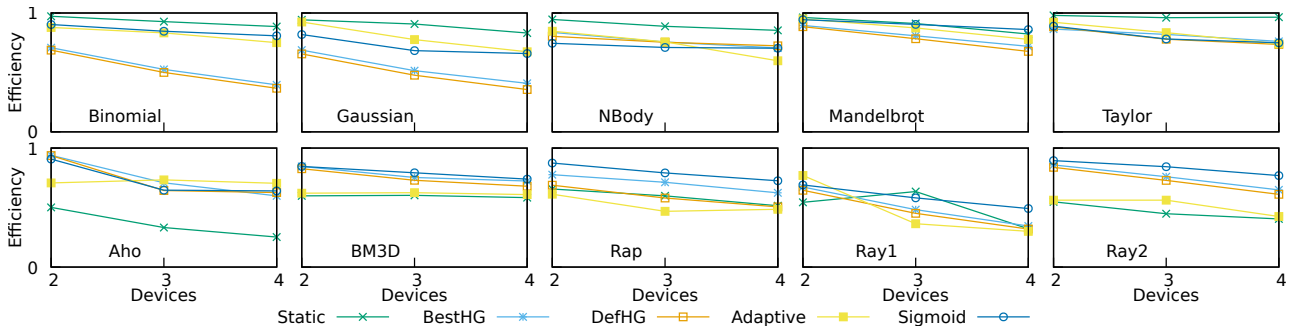


Figure 4.11: Efficiency of the different algorithms executing the benchmarks on a homogeneous system.

Another interesting metric is the energy efficiency, which combines performance with energy consumption. Figure 5.11 shows the Energy Delay Product (EDP) [CCBB15], of the algorithms normalized to that of the baseline. Since this is a combination of the two above metrics, the relative advantage of the different algorithms is maintained. The geometric mean shows that with this metric all algorithms are advantageous, Sigmoid giving the best results with a 54% improvement. BestHG also gives good results (52%) since its parameters have been optimized. These two algorithms give good results in all the benchmarks, while the remaining algorithms exhibit a strong variability, in some cases even with normalized EDP values over one.

In summary, these results prove that co-execution improves the energy consumption of heterogeneous systems, in addition to their performance, as shown in the previous section.

4.6.4. Scalability

The last experiment was developed in *Hydra*, a homogeneous system with four GPUs. This experiment evaluates the *strong scalability* of the load balancing algorithms. Therefore, the same problem size has been used for all the experiments, while the number of devices increases from 2 to 4. For a better comparison, the metric used to evaluate scalability is the efficiency. Perfect scalability means constant efficiency as the number of devices increases. This is usually not the case and, as shown in Figure 4.11, efficiency drops in all cases. The smaller the drop, the better the scalability of the algorithm.

These results show that Sigmoid is the only algorithm that scales well for both regular and

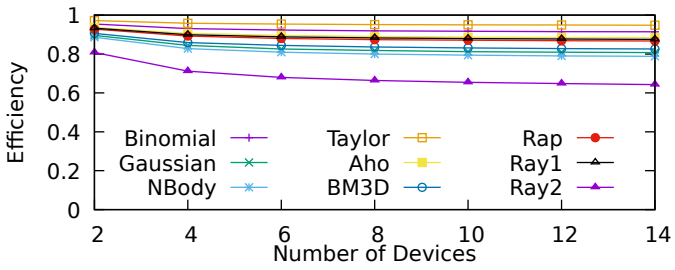


Figure 4.12: Estimation of the weak Scalability of Sigmoid algorithm using Gustafson Law.

irregular benchmarks. Regarding the rest of the algorithms, Static scales very well in regular algorithms, but it has serious problems in irregular ones, such as Aho and Ray1, where it scales well between 2 and 3 devices, but drops strongly with 4 devices. Adaptive also scales well on regular benchmarks, excluding NBody with 4 devices, but behaves very poorly on irregular ones. Finally, both BestHG and DefHG, have an erratic behavior in regular benchmarks, with good scalability for NBody, Mandelbrot and Taylor, but scaling badly for Binomial and Gaussian. Both versions of HGuided also show the same behavior for irregular benchmarks: good scalability for Aho, BM3D and RAP, but very poor for Ray1 and Ray2. In sum, out of the evaluated algorithms, only Sigmoid delivers uniform scalability results, regardless of the behavior of the workload: regular or irregular.

The number of devices available in future systems will surely grow beyond four. For this reason, and based on the data obtained in these experiments, an estimate of the *weak scalability* of Sigmoid has been made with up to 16 devices using Gustafson's Law [Gus88]. Evaluating strong scaling for such a high number of devices would require problem sizes that cannot be executed on just four devices due to memory constraints. Figure 4.12 shows how efficiency evolves by increasing both the number of devices and the size of the problem, so that the workload per device is always constant. As depicted, weak scaling for Sigmoid is almost perfect, according to Gustafson's Law estimates.

In conclusion, Sigmoid achieves almost perfect load balancing, delivering excellent energy and performance results. Moreover, it delivers better overall performance than the load balancing algorithms that require parameters and it also equals the results of the best algorithms for regular and irregular kernels individually. Finally, it is the only algorithm with good scalability in both regular and irregular applications.

4.7. Conclusions

OpenCL is a powerful tool that grants access to the enormous potential of heterogeneous systems. However, it leaves the programmer alone with the manual management of the available resources, which can turn into a complex task, hindering development and difficulting maintainability. This is especially true for co-execution, when all the devices collaborate on the computation of a single kernel. In such case, the programmer needs to manually initialize each device, distribute the data and make important load balancing decisions, which require significant expertise and effort.

Maat works as an abstraction layer that eases the co-execution of OpenCL kernels. It abstracts the programmer from the hardware, enabling the management of the whole heterogeneous system through a single interface, regardless of the available devices. By separating the management from the underlying resources, Maat makes the maintenance of the code easier, as there is no need to perform changes to the applications to account for new hardware. The abstraction layer deals with the data structures associated to the new devices, the required data transfers and the work distribution.

Maat also assists the programmer with the most crucial part of co-execution: load balancing. It implements HGuided and Sigmoid, two novel algorithms that are capable of adapting to different kernel behaviors. Both obtain close to ideal performance under regular and irregular workloads, but Sigmoid shines specially for being an uninformed algorithm, requiring no parameters from the user. By monitoring the execution of kernels and adequately tuning package sizes using its internal function, Sigmoid represents an all-around option for excellent performance and energy efficiency, equalling and, on occasion, surpassing, informed algorithms, and scaling remarkably well. This, together with the abstraction layer offered by Maat, succeed at the goal of attaining effortless, transparent co-execution. Maat has been extended to support other kinds of accelerators such as Xeon Phi in [GNT+19].

Chapter 5

Co-execution support in task-based programming models

HPC systems become more powerful by the year. However, this growth in computing capability has also been coupled with an increase in the complexity of the systems, turning parallel programming into an ever harder endeavour. This has led to different attempts from both academia and industry to aid the programmer, by offering high level interfaces that hide the complexity of the actual system. Task-based data-flow programming models are an example of such an effort. This kind of programming models internally handle much of the complexity of dealing with parallel systems, while obtaining remarkable performance. Regarding heterogeneous computing, most task-based programming models added support for the traditional Host-Device approach to heterogeneity, in which devices are treated as independent entities to which kernels are offloaded. However, they lack when a single task is to be co-executed, leaving the programming and the responsibility of data management and load balancing to the programmer. This chapter presents and evaluates an extension to OmpSs, as an example of a well-known task-based programming language, to enable an effortless and efficient co-execution. This is achieved by automatically dividing a single task and distributing it among the available devices, so they cooperatively execute it. This new capability frees the programmer from the burden of explicitly splitting a task if co-execution is desired, and achieves a degree of

abstraction in tune with what this kind of programming models offer.

5.1. Motivation

A common strategy to easing the programming of parallel systems is increasing the degree of abstraction, hiding low level details while still obtaining competitive performance. This is obtained by expressing parallelism via higher level constructions, using a more comfortable syntax, closer to that of sequential programming. Task-based programming models are an example of this approach. They represent the execution of a parallel program as a set of tasks that have dependencies among themselves (more details in Section 2.1.2). The programmer is then in charge of writing sequential code with certain annotations that define tasks and the data they require and produce. This information is used by a runtime to build a Task Dependence Graph that controls when tasks are scheduled to the available resources. To account for heterogeneity, extensions were added to this models, that enable the programmer to specify that certain tasks are to be executed in a heterogeneous device [F. 14, TV14, MMG16]. An instance of this is the OmpSs `target` directive, shown in Figure 5.1. In the example, the `binomial_options` task will be executed in an OpenCL-capable device. The `ndrange` part of the directive is used to define, in an OpenCL manner, the number of work-items that need to be spawned. In the example, `sample*(numSteps+1)` work-items will be launched along `1` dimension, grouped in work-groups of `numSteps+1` work-items each.

This is a very significant improvement over base OpenCL, as some of the complexity is kept under the hood. However, heterogeneous devices are still treated as completely independent entities, forcing the programmer to explicitly partition tasks between the available devices if any degree of cooperation is desired. This is depicted in Figure 5.2, which shows the necessary changes to co-execute the workload of the simple task of Figure 5.1 using two devices. First, the task itself needs to be modified, adding arguments `start` and `end`. These serve to identify the portion of the total workload to be performed in the execution of the task, and are used to calculate the number of work-items to spawn, defined in the `ndrange` clause. Consequently, an

```
#pragma omp target device(opencl) copy_deps  \\
        ndrange(1,samples*(numSteps+1), \\
                numSteps+1)
#pragma omp task in([samples]randArray)  \\
                 out([samples]output)
__kernel void binomial_options(int numSteps,
         int samples, const __global float4*
         randArray, __global float4* output);
```

```
//Initializations
binomial_options(NUM_STEPS, SAMPLES,randArray, output);
#pragma omp taskwait
//Free resources
```

Figure 5.1: Header file for a sample task using the heterogeneous system extension.

execution of `binomial_options_split` using `start=0` and `end=SAMPLES` would be equivalent to the execution shown in Figure 5.1. Second, the new task needs to be called twice using the new arguments as corresponding. Note that not only the header of the task would need to be modified, but also its body. This is because each of the two task launches are completely independent, using work-item ids that start in `0`. As a result, `start` and `end` need to be used as offsets for each work-item to access the data it requires. This involves analyzing the access pattern of the original task to adapt its data usage for co-execution. Also note that in this example the task is divided into two *a priori* equally sized portions, completely disregarding load balancing. As substantiated throughout this dissertation, load balancing decisions are central to the success of co-execution and far from trivial, depending both on the task at hand and the available hardware. For irregular workloads, dynamic load balancing techniques are necessary to achieve proper performance, requiring the workload to be split into several packages, sometimes with varying sizes. This would represent a significant amount of work for the programmer, which would have to decide the number and size of the subtasks that have to be launched, and manage their data accordingly. Once again, the programmer is effectively left unaided regarding load balancing, which is central to efficiently using a^{o} heterogeneous system. The addition of support for effortless co-execution to an already high level parallel programming framework represents another step towards bridging the gap between programmers and parallel systems.

```
#pragma omp target device(opencl) copy_deps  \\
        ndrange(1,end-start*(numSteps+1), \\
                numSteps+1)
#pragma omp task in([samples]randArray)  \\
                 out([samples]output)
__kernel void binomial_options_split(int numSteps,
         int samples, const __global float4*
         randArray, __global float4* output,
          int start, int end);
```

```
//Initializations
binomial_options_split(NUM_STEPS, SAMPLES, randArray, output, 0,
   SAMPLES/2);
binomial_options_split(NUM_STEPS, SAMPLES, randArray, output, SAMPLES
   /2, SAMPLES);
#pragma omp taskwait
//Free resources
```

Figure 5.2: Header and code for the manual co-execution of a sample task .

5.2. Kernel co-execution in OmpSs

This chapter proposes to add native support for heterogeneous co-execution into task-based programming models, so the cooperative execution of a single task, using all the resources available in the system, does not represent any extra work. This involves automatically handling the division and distribution of the workload represented by the task among all the available devices, so no modifications to its code are required. Splitting the workload implies deciding how much work is scheduled to each device. This is a complex choice, as it requires expertise and determines the achieved performance and energy efficiency. For co-execution support to be useful, the programmer should be spared workload distribution decisions, leaving them to load balancing algorithms that specially target heterogeneous systems. Co-executing also means moving data between the devices involved. This movement should also be kept from the programmer, so he does not have to worry about the data that will be accessed by each work-item or the results it will produce. Finally, all this should be achieved with minimum impact on the programming model itself. This is to preserve its original approach to programming, and ease the use of the new co-execution capabilities and the adaptation of pre-existing applications.

In short, the goal is to enable the programmer to co-execute a task using a code as similar as

possible to the one shown in Figure 5.1, requiring no modification to the task itself. OmpSs has been chosen as a sample task-based programming model, but this chapter and its conclusions can be generalized to other programming models.

5.3. Load Balancing Algorithms

To account for the original philosophy of OmpSs, the classic OpenMP load balancing algorithms have been adapted to a heterogeneous environment and implemented in the new scheduler. These are the *Static*, *Dynamic* and *Guided* algorithms. The latter has been renamed to *HGuided* as it is an implementation of the HGuided algorithm presented in Section 3.3. The overall behavior of these algorithms is illustrated in Figure 5.3. It shows the ideal case in which in the execution of a regular application all devices finish simultaneously, thus achieving perfect load balance. An uninformed version of the HGuided algorithm, called *Auto-Tune*, has also been implemented to offer an effortless co-execution algorithm that does not completely depart from the traditional OpenMP approach to load balancing. The following Sections provide a deeper insight on these algorithms.

5.3.1. The Static algorithm

Static algorithms operate before a task starts its execution, dividing the workload in as many *packages* as devices are available in the system. OpenMP has traditionally targeted homogeneous systems, so coming up with a balanced work distribution is relatively simple. The workload is just split in equal packages and, provided that it is regular, a balanced distribution is likely to be found. Certain subtle effects may have an impact the success of the load balancing though. One such is memory access time variability in NUMA systems. To control affinity, the OpenMP Static algorithm provides a parameter to define a block size that determines how the packages are built. Each device will be statically assigned a set of blocks of contiguous threads, that are assigned in round-robin. For the next section that is distributed, if the same block size

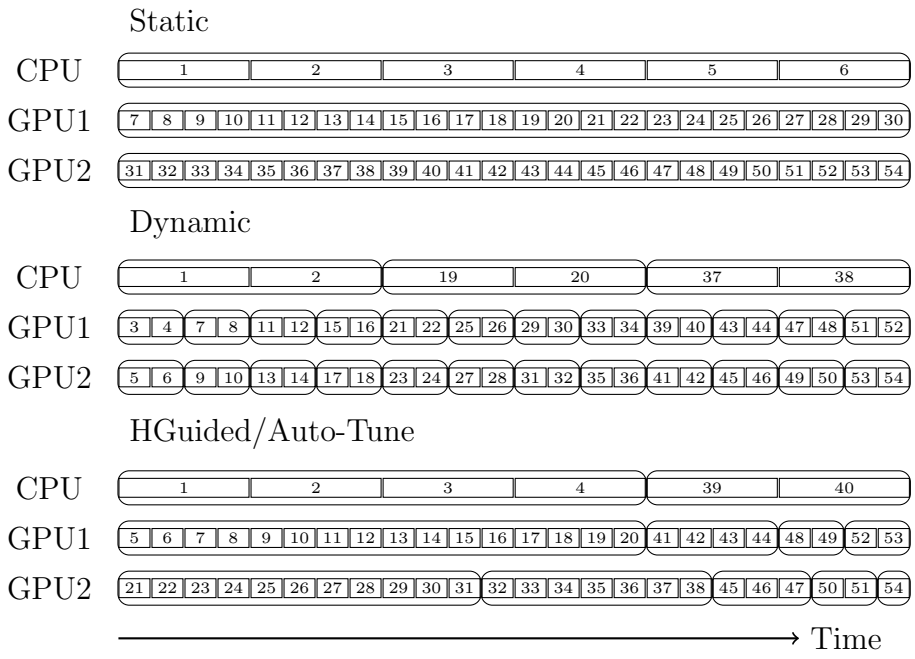


Figure 5.3: Depiction of how the four algorithms perform the data division among three devices. The work groups assigned to each device, identified by numbers, are joined in packages shown as larger rounded boxes. Note that the execution time of work groups in the CPU is four times larger than in the GPUs.

is used, the devices are guaranteed to receive the same set of block of threads, which favours affinity.

However, heterogeneous systems are affected by a not so subtle effect that, if not adequately addressed, can render a work distribution completely useless: the devices potentially have different computing speeds. Consequently, a straightforward implementation of the OpenMP Static algorithm, applied to heterogeneous systems, will be virtually useless. Scheduling equal amounts of work to devices with different computing speeds is unreasonable, as execution times will also differ and some devices will idle while waiting for others.

To account for this requirement, the Static algorithm proposed for heterogeneous co-execution in OmpSs receives the computing speeds of the available devices s_i as parameters. The algorithm performs an *a priori* division of the workload, so the computing speeds also need to be known *a priori*. This can be easily computed by performing an execution of the kernel using each of the available devices individually. The results presented in Section 5.7.1 will show that inaccurately setting this parameter sometimes has a strong impact on performance. The Static algorithm

then proceeds by splitting the workload into as many children work descriptors as devices, each with a number of work-groups proportional to the computing speed of the receiving device in relation to the aggregate computing speed of the heterogeneous system $\frac{W \cdot s_i}{\sum_{i=1}^{N} s_i}$.

This behavior is depicted in Figure 5.3. Considering the significant overhead of transferring data to accelerators, it might seem fitting to implement a scheme to preserve affinity in a similar fashion to OpenMP. However, this cannot be easily translated to heterogeneous systems. This is because computing speed depends not only on the devices, but also on the kernel to be co-executed. Consequently, the small gains that can be achieved through affinity are likely to be lost in an imbalanced work distribution.

The main advantage of the Static algorithm is that it minimizes the number of host-device interactions. Each device computes for the workload of a single work descriptor, while the host just waits for their completion to gather the data. Consequently, the Static algorithm performs well when facing regular workloads, which do not require any degree of adaptiveness, so the lesser the overhead, the better the obtained performance. However, the Static algorithm does not adapt, so its performance is not as good for irregular loads, which require dynamic techniques to be able to conform to their unpredictable behavior.

5.3.2. The Dynamic algorithm

Some applications are irregular, so they do not present a constant load during their executions. To adapt to their irregularities, dynamic algorithms divide the workload into small packages of equal size. The number of packages is well above the number of devices in the system. During the execution of a task, the host will be in charge of assigning packages to the different devices on demand. The OpenMP dynamic algorithm is an example of this approach to load balancing, taking the desired size for the packages as a parameter.

This algorithm can be easily adapted to OmpSs heterogeneous co-execution. The runtime splits the G work-groups in work descriptors, each with the package size specified by the user in an environment variable. This number must be a multiple of the work-group size. If the number of

work-items is not divisible by the package size, the last work descriptor will be smaller. Then, the runtime assigns one work descriptor to each device and waits for the completion of any of them. When device d_i completes the execution of the work represented by a work descriptor it will get scheduled a new one.

This behaviour is illustrated in Figure 5.3. The workload is divided in small, fixed size packages and the devices process them achieving equal execution time. As a consequence, this algorithm adapts to the irregular behaviour of some applications. However, each completed package represents an interaction between the host and a device. This overhead has a noticeable impact on performance, specially in OmpSs as runtime operation is costly.

5.3.3. The HGuided algorithm

Accounting for the high overheads that pure dynamic approaches often produce, the OpenMP Guided algorithm tries to reduce the number of packages generated while remaining adaptive. It does so by using decreasing package sizes as the execution of a task progresses. The adaptation of this algorithm for OmpSs heterogeneous co-execution is equivalent to the one for HGuided presented in Section 3.3. The computing speed of the devices and the minimum package size have to be specified specified by the programmer, and the package size is calculated as follows $package\_size_i = \left\lfloor \frac{G_r}{CN} \cdot \frac{s_i}{\sum_{j=1}^{N} s_j} \right\rfloor$. Note that unlike the previous algorithm, Guided does not allow for all the children work descriptors to be created when a task is launched and later scheduled. This is because package sizes depend on the actual execution and the device that will compute for each package, so children work descriptors need to be created on demand.

The HGuided algorithm strikes a balance between adaptiveness and overheads, which makes it a good all-around solution that adequately distributes the workload for both regular and irregular applications. However, it is still an informed algorithm, requiring two parameters to be provided by the programmer: the computing speed and the minimum package size. These have a key impact on the success of co-execution and are dependent on both the executed application and the devices themselves. Determining the best value for the minimum package size is specially complicated, especially for GPUs, because it is essential to do a sweep to obtain a value that

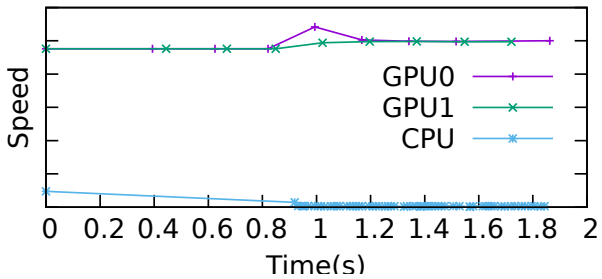


Figure 5.4: Evolution of the computing speed per device.

gives good results. Moreover, HGuided is quite sensitive to these parameters, so choosing an adequate value for them is sometimes a demanding task that requires a thorough experimental analysis. For this, reason an uninformed version of the HGuided algorithm, called Auto-Tune, has been implemented. It delivers similar performance and energy efficiency to HGuided, but does not require any parameter from the programmer.

5.3.4. The Auto-Tune algorithm

Overview

For co-execution to be truly effortless, load balancing should not require any intervention from the programmer. However, all the aforementioned algorithms require parameters that condition the success of co-execution and require a significant effort to be set. To quantify the impact of using inaccurate parameter values, the sensitivity of the Static, Dynamic and HGuided algorithms to their parameters is evaluated in 5.7.1. The conclusion is that all the algorithms, but especially HGuided, may deliver significantly degraded performance when using slightly off-key parameter values.

The *Auto-Tune* algorithm is an evolution of HGuided that strives to eliminate the need for parameters while retaining near optimal performance and efficiency, for both regular and irregular loads. It uses the same formula as HGuided to calculate the package size, but initially uses nominal parameter values that are then adjusted at runtime. Auto-Tune also handles the minimum package size differently depending on the device type that each package will be sent to.

Computing speed

As explained in the previous section, this algorithm starts using default parameter values that are later tuned throughout the execution. The use of default values at the beginning of the execution has negligible impact on performance, as accurate values are only necessary near the end of the execution, which is one imbalances may happen. By then, the algorithm will already have calculated adequate values for the kernel. The computing speed for the first package launched at each device is calculated using the theoretical GFLOPs of the hardware. These can be obtained at the installation of OmpSs either by querying the available devices or by running a simple compute intensive benchmark. For the successive packages, the speed is updated taking into account the computing speed displayed by each device. This is calculated as the average number of work-items processed per second for the last three packages launched to each device. By using the average speed of the last packages, a gradual adaptiveness is attained that keeps the algorithm resistant to bursts of irregularity that would not be representative of the actual speed for the next packages. Three packages have been found to strike a balance between adaptiveness and accounting for the history of the execution. Figure 5.4 depicts the evolution of the computing speed during the execution of one of the applications used for experimentation. The nominal computing speeds are used at the beginning of the execution until all the devices have finished at least one package. Then, the computing speeds are updated at runtime. In the figure, the nominal speed for the CPU was higher than the actual one for the application. Note that the use of the nominal speeds for the initial packages does not disturb the load balancing, as all the devices are kept busy and do not delay the completion of the benchmark. This operation is similar to that of the Sigmoid algorithm presented in Section 3.4.

Minimum package size

Package size also has an influence on the computing speed of throughput based architectures, such as GPUs. Consequently, package size must be kept relatively high to prevent an inefficient use of the hardware and overheads. However, this is also a potential source for imbalance. If

the computing speed of the devices differs greatly, a high minimum package size that reduces overheads is likely to be too big for slow devices, namely, CPUs, which would cause delays. To prevent this, the Auto-Tune HGuided algorithm uses different minimum values for CPUs and GPUs.

The value selected for the CPU is one work-group per CPU core, so no hardware is left unused and imbalance is avoided. This is because the CPU is not a throughput device, so its computing speed is usually much less sensitive to package size than the GPUs. Moreover, CPUs are often the slowest device of the system, so using a small minimum package size with them will improve the load balancing. Moreover, considering that data is already held in main memory, using a small package size for CPUs does not have an impact on communication overheads.

Two values are considered for the GPU minimum package size. First, the equations implemented in the CUDA Occupancy Calculator are used to obtain the minimum number of work-groups that will achieve maximum occupancy for the current kernel and GPU. The CUDA Occupancy Calculator is part of the CUDA Toolkit since version 4.1. This value is a lower bound for the minimum package size, but might be too low if the application launches a large amount of work-items, producing too many packages and high overheads. To prevent this, the number of work-items is also analyzed and the final minimum package size is set to the maximum between the value obtained by the Occupancy Calculator and 5% of the work-items. This percentage has been experimentally set to keep the number of packages low and avoid performance degradation in the GPU.

These enhancements give forth an uninformed algorithm with improved adaptiveness, that delivers comparable performance to HGuided for a fraction of the effort. It completely eliminates the need to provide any parameter and saves a great deal of pre-processing time per application and system, as will be shown in Section 5.7.1.

5.4. Design

OmpSs has evolved throughout the years to provide high level abstractions for parallelism at an outstanding performance. It effectively does most of the work for the programmer while hiding the complexity of managing parallel systems. However, a framework that makes hard things easy will necessarily be complex underneath. For this reason, decisions on how to implement new functionality need to be carefully made, as a mistake may compromise the ease of implementation, maintainability or success of the solution.

The main design principle in the implementation of co-execution support, has been respecting the approach to parallel programming of OmpSs. It offers well defined abstractions that favour certain programming strategies, that should be preserved for co-execution to be seamlessly integrated in the framework and become a useful solution, with equivalent effort to using just a single device. For this reason, the already present `target` directive has been used to implement co-execution.

As presented in Section 2.1.2 the features offered by OmpSs have their foundation in the combination of two main building blocks: Mercurium, which is a source-to-source compiler, and Nanos++, which is a runtime capable of managing tasks, their data and the *Task Dependence Graph (TDG)* their dependencies generate. As a consequence, the first design decision when incorporating new functionality into OmpSs is determining whether it will be implemented as part of the compiler, of the runtime or arise from the cooperation of both. This decision has an impact on the programming of the framework, its maintainability and even its performance and functionality.

For an educated decision on where to implement co-execution, lets first go back to its definition and correlate it with the OmpSs infrastructure. When we talk about co-execution, we refer to the orchestration of several devices, cooperating on the computing associated to a single task, in a data-parallel fashion. OmpSs extracts parallelism from the execution of tasks. Consequently, the transparent co-execution of a task in OmpSs may be implemented as the automatic generation of several children tasks that collectively represent the same computation as their parent.

This will be accordingly scheduled to the available devices. OmpSs tasks are defined as common functions, appropriately characterized using a directive. Any call to a function defined as a task will be translated by Mercurium into code that uses the Nanos++ runtime to generate and manage the required tasks. Considering the above, co-execution effectively lies at the border between runtime and compiler. However, a defining factor sets apart the runtime as the most adequate element to implement it: adaptiveness. To achieve proper load balancing, it is necessary to react to the varying behavior of both applications and heterogeneous systems. For example, finer grained child tasks may be necessary if the parent task shows an irregular behavior. This kind of decisions cannot be made beforehand, which definitely turns Mercurium into a poor choice to implement co-execution.

The next design decision is how to integrate co-execution in the runtime. The main purpose of Nanos++ is the research of parallel systems. Consequently, it follows a modular design, that can be easily extended by means of plugins. These are selected for each execution using runtime options, usually via environment variables. For instance, OmpSs offers several modules as plugins, implementing different throttling policies, instrumentation schemes or schedulers. The latter are in charge of implementing different policies to distribute tasks among the available computing elements. Therefore, it seems fitting to implement co-execution support as a scheduler plugin that transparently distributes the workload associated to a single task among all the available devices. Such an approach has negligible impact on programming, as the code of a task or that associated to its execution require no modification. Only the proposed scheduler and the desired load balancing algorithm, together with its parameters, if needed, have to be specified via environment variables. Figure 5.5 compares the environment variables necessary to use a single device to those used for co-execution. This is the only modification necessary to use all the available hardware. Consequently, this design achieves truly effortless co-execution, enabling code originally designed for a single device to make the most of all the available hardware by just adequately setting certain environment variables.

```
# Single device
NX_OPENCL_DEVICE_TYPE=GPU
NX_SCHEDULE=bf
```

```
# HGuided load balancing
NX_SCHEDULE=co-execute
NX_ALGORITHM=hguided
NX_GPU_SPEED=34.0
NX_MIN_PACKAGE_SIZE=115200
```

Figure 5.5: Comparison of the environment variables to use for single device execution and co-execution.

5.5. Implementation

In OmpSs tasks are represented by *Work Descriptors*, which hold the information of the computation that is to be performed. This includes the executable for the task, its input data and the location where outputs should be written to, which are stored in a *CopyData* data structure. Mercurium is in charge of translating a call to a function that has been defined as a `#pragma omp task` into calls to the necessary runtime functions to create the corresponding work descriptor. The scheduler in turn is responsible of managing tasks and how and when they are executed. A simplified example of the launch of an OmpSs task and an excerpt of the generated Mercurium code is shown in Figure 5.6a .

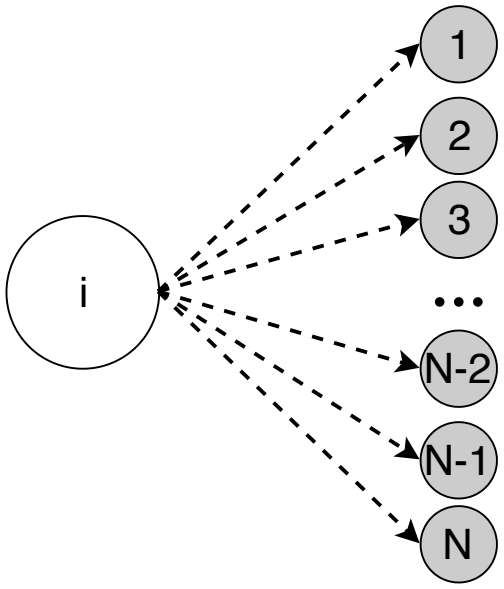
As substantiated in Section 5.4, co-execution has been implemented as a new scheduler. When function `nanos_submit` is called to enqueue the new work descriptor into the runtime, it is stored in the scheduler, and new children work descriptors are created and added to the task pool. These collectively represent the workload associated to their parent. This operation is depicted in Figure 5.6b, which depicts the children work descriptors generated by the new scheduler. The created children work descriptors are shown in grey, while the one expressed in the code of Figure 5.6a is white. Children work descriptors are an OmpSs notion. They are defined as work descriptors spawned by their parent, not considering the latter as finished until all its children have completed their execution. This enables the implementation of co-execution to preserve the logic of the original DAG. No task following the one in co-execution will start until all the children work descriptors, and consequently their parent, have finished.

Each of the children work descriptors is identical to its parent except for two key differences. First, they have different OpenCL parameters, namely `global_work_size` and `offset`, express-

```
//Initializations
binomial_options(NUM_STEPS,
   SAMPLES, randArray, output)
   ;
#pragma omp taskwait
//Free resources
```

```
//Simplified Mercurium code
create_wd_compact_opencl(...);
nanos_submit(...);
```

(a) Sample code for the launch of a task and the corresponding code generated by Mercurium. Header of the task in Figure 5.1



(b) Depiction of the created children work descriptors.

Figure 5.6: OmpSs code and depiction of the children work descriptors generated by the co-execution extension.

ing the workload that the work descriptor represents. For example, lets imagine a task that launches 1000 work-items. If it was evenly split into two children work descriptors, each one would have a `global_work_size` of 500, but the first one would have an offset of 0, while the second one would have 500. Second, their output data is just a portion of that of their parent, which is conveniently offset so the results are written adequately. This is represented by an independent *CopyData* object, holding the start address and size that the package will have to work on. As a result, coherence problems are avoided in the OmpSs directory. Apart from the aforementioned details, data transfer relies on the well tested and documented methods used by standard OmpSs. To perform the correspondence between work descriptors and output data, an assumption is made: each OpenCL work-item will produce the result for the position of the output buffers indexed by its identifier. This may seem a strong requirement, but it is met by most kernels widely used in the industry and research.

However, a limitation was found in the original implementation of OmpSs that required addressing. The information for the `global_work_size` is handled by Mercurium and hardcoded into the function that ultimately launches kernels. This issue effectively makes runtime supported co-execution impossible. To solve it, a slight change to Mercurium was necessary to

make OpenCL kernel configuration parameters available to Nanos++. Consequently, a new Mercurium work descriptor creation function has been implemented, which behaves like the original but includes these parameters in the data held by the work descriptor, so the runtime can access and modify them. The creation of the children work descriptors is performed by a modified version of the `duplicateWD` function that does this extra work. This function is also responsible for making the OpenCL parameters of the divided work descriptors available to the Mercurium code, which will trigger the actual kernel launches.

Once the submission of the parent work descriptor is completed, the `done` function is called. This is a Nanos++ function that is used to signal the completion of a work descriptor. It also waits for the completion of the children of the calling work descriptor. In this way, no task dependent on the divided one will be run until all the children resulting from the work distribution are completed, so the dependencies of the task graph are maintained.

The actual decisions on when to launch the children tasks, which involve detecting when devices are idle, are performed by the original Nanos++ implementation, and required no modification. This also holds true for the movement of data from and to the devices. This is because co-execution has been implemented making the most of OmpSs strengths and striving to minimize the impact on its original implementation.

5.6. Methodology

This section completes the information on methodology provided in Section 1.5 with the details particular to the experimentation in OmpSs. It presents the applications chosen for evaluation and their maximum achievable speedup considering the performance the available devices show executing them. Experiments have been carried out using the *Batel* system, presented in Section 1.5.1.

Six applications have been chosen for the experimentation. Three of them: *NBody*, *Krist* and *Perlin* are part of the OmpSs examples offered by BSC, and the other three: *Binomial*, *Sparse Matrix and Vector product (SpMV)* and *Rap* have been specifically adapted to OmpSs from

Table 5.1: Parameters for each application

Benchmark	Type	Problem size	Local work size	GPU Speed	Package size	Min. size
NBody	Regular	8192000	128	1.67	1024	8192
Krist	Regular	800000, 2048000	128	4.47	4096	4608
Binomial	Regular	8192000	256	34.00	1152000	11520000
Perlin	Regular	32768	128	22.00	1024	1664
SpMV	Irregular	1024000, 327680000	128	22.00	1024	8192
Rap	Irregular	1024×1024	64	6.14	16384	65536

existing OpenCL applications. The first four (NBody, Krist, Binomial and Perlin) are regular, meaning that all the work-groups represent a similar amount of work. On the contrary, SpMV and Rap are irregular, which implies that each work-group represents a different amount of work. The parameters associated to each of the load balancing algorithms have been set to maximize performance. The computing speed for a device/application pair has been obtained as the relative performance of the device, with respect to that of the fastest device for the application.

Perlin implements an algorithm that generates noise pixels to improve the realism of moving graphics. Krist is used on crystallography to find the exact shape of a molecule using Röntgen diffraction on single crystals or powders. Rap is an implementation of the Resource Allocation Problem. It has a certain pattern in its irregularity, because each successive package represents an amount of work larger than the previous. The parameters used for each of the applications are shown in Table 5.1. Note that the computing speed represents how many times the GPU is faster than the CPU.

To give an idea of how successful co-execution has been, the speedup over the baseline obtained for each benchmark will be compared to the maximum achievable speedup. This is obtained using Equation 1.1, presented in Section 1.5.2. Note that, for the employed benchmarks, the CPU is much slower than the GPUs, and is in charge of running the OmpSs runtime, which is quite heavy. Then, considering that *Batel* was used in the experiments, which holds 2 GPUs and a CPU, the maximum achievable speedup using the three devices will not be 3, but a fraction over 2 which depends on the computing speed of the CPU for the application. The speedup for each application using a perfectly balanced work distribution is shown in Table 5.2. These values give an idea of the advantage of using the complete system.

Table 5.2: Maximum achievable speedup per application.

Application	NBody	Krist	Binomial	Perlin	SpMV	RAP
Max. Speedup	2.61	2.2	2.03	2.04	2.05	2.16

5.7. Evaluation

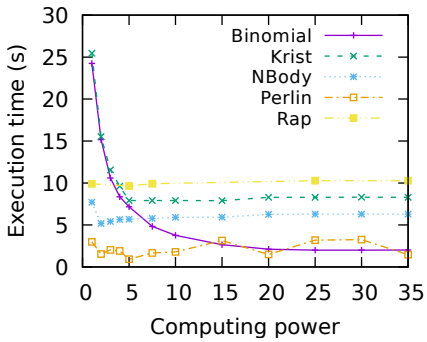
This Section evaluates the proposed strategy for heterogeneous co-execution in task-based parallel programming languages as implemented in OmpSs. These experiments aim to answer the following questions:

- How sensitive are the proposed load balancing algorithms to their parameter values, considering that OmpSs might generate greater overheads than other lighter weight heterogeneous programming frameworks?
- Do the proposed load balancing algorithms adequately distribute the load?
- Does Auto-Tune manage to successfully balance the workloads regardless of their behavior?
- Does heterogeneous co-execution make sense performance-wise and energy-wise, apart from the increase it represents in abstraction and ease of programming?

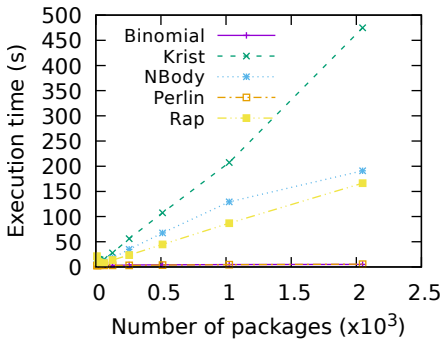
5.7.1. Parameter sensitivity

As explained in Section 5.3, the Static, Dynamic and HGuided algorithms require different parameters for their operation. These have to be provided by the programmer and are one of the key factors for a successful load balancing. However, determining the most adequate values for a workload is not trivial, as they may differ greatly between applications and device configurations. Consequently, the selection of parameters is often a work intensive process, usually based on experimentation.

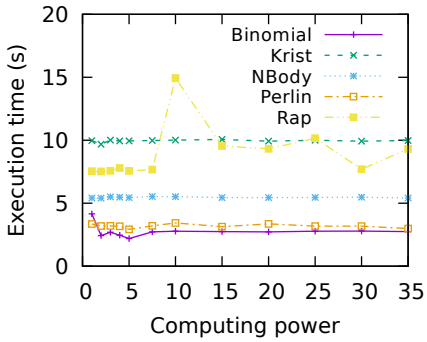
The importance of adequately choosing the parameter values is illustrated in Figure 5.7, which displays the execution time for the applications when varying the parameters for each of the



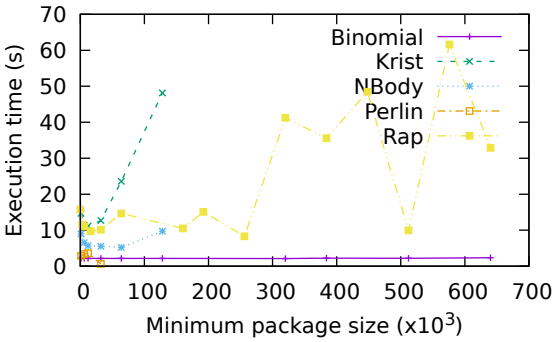
(a) Execution time with different computing speeds for the Static algorithm.



(b) Execution time with different numbers of packages for the Dynamic algorithm.



(c) Execution time with different computing speeds for the HGuided algorithm.



(d) Execution time with different minimum package sizes for the HGuided algorithm.

Figure 5.7: Parameter sensitivity analysis

algorithms. Note that for the HGuided algorithm, when one of the parameters is modified, the other is set to the identified optimal value. As shown in the figure, for every of the parameters, the applications show very different behaviors, ranging from near insensitivity to delivering greatly degraded performance, sometimes even lacking a clear relation with the parameter value, as is for example the case of Rap for the minimum package size. Moreover, the applications are not affected equally by the parameters. For example, Binomial is highly sensitive to the computing speed in the Static algorithm and moderately sensitive to almost insensitive to the rest of parameters, while Rap behaves just the opposite: it is insensitive to the Static computing speed and tremendously sensitive to the other parameters.

Lets analyze each parameter separately. First, regarding the computing speed for the static, the performance of Binomial, Krist and NBody improves as bigger values are used, until a minimum is reached. Going beyond this value either has no effect or represents a slight loss of performance. This can be seen in NBody. Binomial and Krist are specially affected by this parameter, while Rap is almost insensitive and Perlin highly irregular. This parameter behaves

very differently for the HGuided. In the case of this algorithm, regular kernels are almost insensitive, excluding Binomial, in which the CPU is much slower than the GPU. In light of this, it can be concluded that the use of an accurate minimum package size can compensate for an inaccurate computing speed. However, the same cannot be said for irregular kernels. Due to their very nature, their performance is highly variable when the computing speed is modified. With respect to the number of packages generated by the dynamic algorithm, kernels behave as expected. Increasing their amount of packages improves performance until an optimum is reached. Beyond that point performance degrades due to overheads. This behavior is common to all the benchmarks. Krist and NBody are shown to be specially sensitive to overheads. This is because each of their work-items perform a small amount of work. The kernels show a similar behavior when modifying the minimum package size. However the performance of Rap is very variable when using big package sizes. This is because big minimum package sizes overheads the capability to adapt to irregular workloads, generating an unpredictable behavior. Note that certain applications, such as NBody, spawn fewer work-items than others. For these, only smaller minimum package sizes could be tested.

Considering these results, it is obvious that, in order to achieve an accurate load balancing, an experimental tuning of the algorithm parameters is often a must. This also shows the importance of uninformed algorithms that free the programmer from the burden of having to tune the parameters for each individual kernel and hardware configuration. The Auto-Tune algorithm, by automatically adjusting the parameters, is such an algorithm. It does not require any parameters while it matches and even surpasses the performance of HGuided in certain cases.

5.7.2. Experimental results

The experiments presented in this section have been developed using the optimal values for the parameters required by each algorithm, obtained in the previous section. This implies that the results for the Static, Dynamic and HGuided algorithms are the best that can be achieved, but require a great effort to tune the parameters.

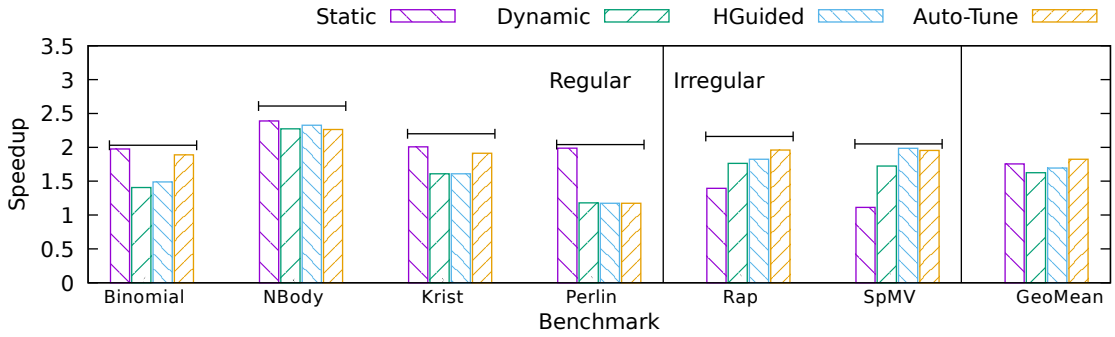


Figure 5.8: Speedup per application.

Performance

Figure 5.8 shows the speedup obtained for each application, calculated with respect to their execution time using the baseline system, as explained in Section 1.5. The values for the maximum achievable speedup are shown in the graph as horizontal lines above each benchmark. Additionally, the geometric mean is shown, which includes both four regular benchmarks and two irregular ones.

From the results of the geometric mean it can be seen that the best result is obtained by the Auto-Tune algorithm, closely followed by the Static, the HGuided and finally the Dynamic. However, it is worth noting that more regular than irregular kernels, which benefits Static. With more irregular ones, Auto-Tune would be ahead by a greater margin. Furthermore, it should be emphasized that the Auto-Tune algorithm is much easier to use, because it does not require finding optimal values for any parameter.

A detailed analysis of the speedups reveals that the Static algorithm is the best option for regular applications, with an average speedup for these four kernels of 2.08. This is because they require no adaptiveness, so they benefit from the minimum overhead introduced by the Static algorithm. However, the Auto-Tune algorithm achieves very similar results to Static with less configuration effort in every benchmark but Perlin, which is very sensitive to overheads as can be seen in the results for all the algorithms but Static. This is because it has a shorter execution time and generates fewer work-items than the rest of applications. The result is a greater impact of the overheads. The other two algorithms achieve good results, but suffer from a problem that reduces performance. If one of the last packages is assigned to the slowest

device it is likely to delay the execution of the whole application. This problem could be avoided by increasing the number of packages, but in that case overheads come into play, which also degrade performance.

For irregular applications, the best results are obtained by Auto-Tune and HGuided algorithms, with an average speedup of 1.96 and 1.91 respectively for these kernels. Their adaptive behaviour favours load balancing in these applications, where the workload of each work-group is completely unknown and unpredictable. On the other hand, the reduction in host-device interactions with respect to Dynamic reduces the runtime overhead, which is inherent to this type of algorithms. This is the reason why the HGuided and Auto-Tune algorithms deliver better performance than the simpler Dynamic algorithm. Finally, the Static algorithm fails to balance the load because it cannot cope with the unpredictability of these applications.

Overall Auto-Tune achieves the best performance, followed by Static. However, this is deceptive, as there are more regular than irregular kernels in the evaluation. If addressed separately, Static achieves an average speedup of 2.08 for the Static kernels and of 1.25. Regarding Auto-Tune, it achieves speedups of 1.76 and 1.96. In light of this, Auto-Tune is the best option to face a kernel with unknown behavior. Moreover, it does not require any parameters

Load balance gives an idea of the degree of utilization of the system and, consequently, of how well a load is balanced. A value of one represents that all the devices have been working all the time, thus achieving the maximum speedup. In Figure 5.9 the geometric mean efficiencies show that the best result is achieved by Auto-Tune with an efficiency around 0.85. In addition, there is at least one load balancing algorithm for every application that achieves an efficiency over 0.9 or even as high as 0.98, reached by Binomial and Perlin with the Static. This is true even for the irregular applications, in which obtaining a balanced work distribution is significantly harder.

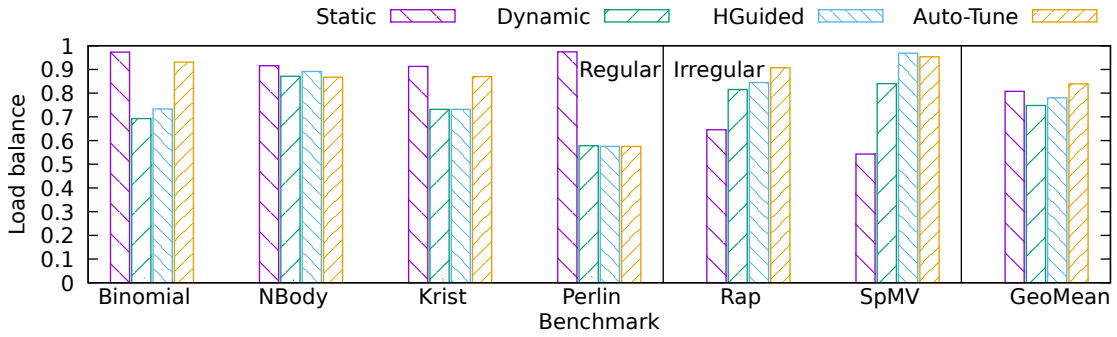


Figure 5.9: Load balance of the heterogeneous system.

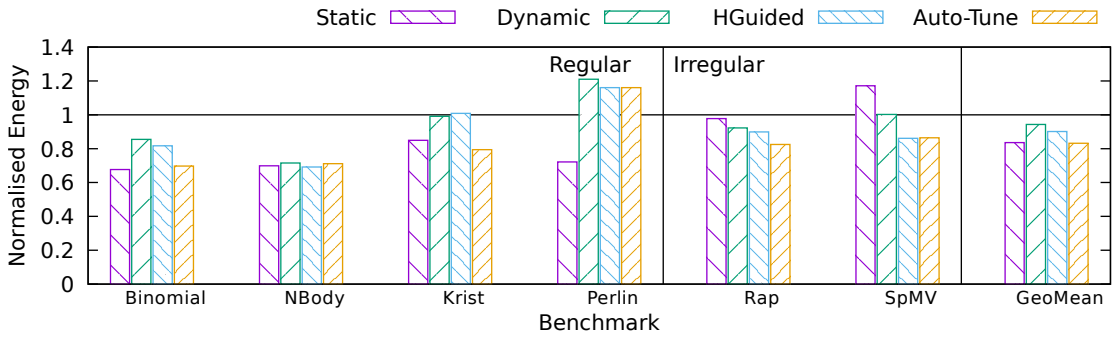


Figure 5.10: Normalized energy consumption per application.

Energy

Not only performance, but also energy consumption and efficiency are very important in the evaluation of a computing system. Figure 5.10 shows, for each benchmark, the energy consumption of each algorithm, normalized to the consumption of the baseline system, meaning that less is better. The baseline only uses one GPU while the other devices are idle and still consuming. This would be the case of a current HPC system, in which failing to use all the available resources may represent an energy waste.

The values of the geometric mean indicate that the algorithms that consume less energy are Static and Auto-Tune, with a saving of almost 20% compared to the baseline. Regarding the individual benchmarks, it is always possible to find an algorithm where the normalized energy is less than one. Moreover, all the algorithms reduce consumption, despite using the whole system. The use of more devices necessarily increases the instantaneous power at any time. But, since the total execution time is reduced, the total energy consumption is also less. Furthermore, since idle devices still consume energy, making all devices contribute work is beneficial.

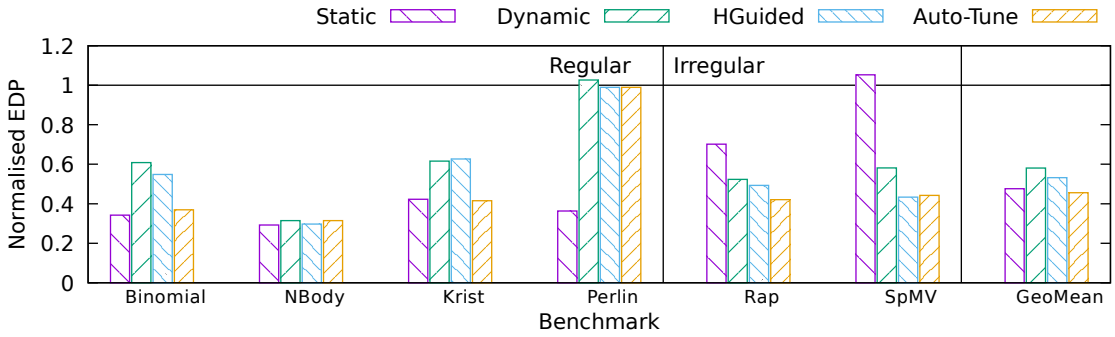


Figure 5.11: Normalized EDP per application.

The analysis of the algorithms shows a strong correlation between performance and energy saving. Consequently, the best algorithm for regular applications is also the Static, with an average saving of 26.5%. However, for irregular applications, it wastes 7.4% of energy. On the other hand, the Auto-Tune gives an average energy saving of 16%.

Regarding the results of individual benchmarks, it is interesting to comment Krist. The highest energy saving in this benchmark is provided by Auto-Tune, although it is not the best in performance. The reason for this is that the execution for Auto-Tune is less balanced that that for Static. The result is that the GPUs, which are more energy efficient, execute more work, so consumption is decreased even though the CPU is idle for a fraction of the time. Consequently, for this particular application, best energy is likely to be obtained when using the two GPUs exclusively. There are only two particular benchmarks where the use of the whole system employs more energy than the baseline. These are Perlin with Dynamic, HGuided and Auto-Tune, and SpMV with Static. This is because, in these cases, the gain in performance is too small (around 18% and 12% respectively) and cannot compensate for the increased power consumption involved in using the complete system.

Another interesting metric is the energy efficiency, which combines performance with consumption. With the dual goal of low energy and fast execution in mind, the *Energy Delay Product (EDP)* is the product of the consumed energy and the execution time of the application. Figure 5.11 shows the EDP of the algorithms normalized to the EDP of the baseline.

Since the EDP is a combination of the two above metrics, the previous results are further corroborated. Therefore Auto-Tune also achieves the best energy efficiency results on geometric

mean, followed by Static, HGuided and Dynamic. Attending to the individual algorithms, their relative advantages is also maintained. Although the Static algorithm on regular applications shows a significant reduction of the EDP of 65%, the same is not true on irregular ones, reducing only 12.4%. In contrast, the Auto-Tune is more reliable, as it achieves a similar reduction on both kinds of applications; 48% on regular and 57% on irregular. Moreover, HGuided and Auto-Tune are the only algorithms that always achieve an improvement over the baseline in terms of energy efficiency, with Dynamic representing a decrease in the efficiency of Perlin and Static of SpMV.

5.8. Conclusions

High level programming frameworks have traditionally eased the obtaining of excellent performance through abstractions that hide complex implementation details, which are internally managed. However, heterogeneous systems are often still treated as second class citizens. This is specially true regarding co-execution, which, if desired, usually has to be handled manually by the programmer, including complex tasks such as load balancing.

This chapter presents an extension to OmpSs, as an example of a task-based programming model, to support the effortless co-execution of massive data-parallel kernels on heterogeneous systems. Co-execution is offered as an OmpSs plugin, which has been implemented with the ease of use as a priority. Therefore, if the programmer desires a kernel to be co-executed, he will only have to load the corresponding plugin and select the desired load balancing algorithm, with no impact on the code of the application whatsoever. This is the result of a careful design that has leveraged the strengths of OmpSs and embraced the original design of the framework.

Regarding load balancing, the extension implements the classic OpenMP algorithms, adapted to heterogeneous co-execution. As expected, the Static algorithm is the best option for regular loads, while HGuided excels for irregular ones, achieving remarkable performance for regular ones too. The Dynamic algorithm has been found unable to adequately balance the load. This is due to OmpSs introducing a significant overhead due to its very operation as a complex data-

flow driven framework. This is an important conclusion, as for a load balancing algorithm to be successful in an environment similar to OmpSs, it will have to consider its sensitivity to overheads, keeping the number of generated packages low as HGuided does. A fourth load balancing algorithm is also presented, which is an uninformed version of HGuided called Auto-Tune. This algorithm proves to be the best all-around solution for truly effortless co-execution, delivering equal or better performance than HGuided, while requiring no action from the programmer.

Chapter 6

Hardware supported co-execution in heterogeneous systems

Accelerators were initially complex pieces of hardware, implemented in discrete cards and communicated with the CPU through slow interconnections. Nevertheless, heterogeneous systems are getting closer together. Now, thanks to miniaturization, a single SoC may hold CPU cores and a capable GPU, sharing the same memory space. As a result, integrated architectures have permeated computing systems, ranging from HPC to mobile devices. This is because they offer promising features that have enabled higher levels of abstraction. However, the accelerator of the system is still treated as a second class citizen, in a classic host-device kind of relationship. Despite the integration in a single chip, there is neither software nor hardware support for low-level co-execution in this kind of devices. This chapter proposes to fully use the potential capabilities of integrated systems to enable hardware supported co-execution, so a kernel launched to a heterogeneous system fully uses all its resources with no programmer effort. The new design has been evaluated through simulation using gem5, and focuses on OpenCL workloads, but it could be extended to other programming models.

6.1. Motivation

The history of computer architecture is a tale of integration, fuelled by an ever decreasing transistor feature size. This was known as Dennard scaling. However, since 2005 clock frequencies improve much more slowly as devices become smaller. As a result, computer architects had to devise more sophisticated ways to obtain performance using the extra transistors available. As the capacity for integration grew over time, chip designers had to face a dilemma: adding more cores or going heterogeneous, devoting part of the silicon to a GPU. The undeniable success of accelerators convinced every manufacturer to choose to design products along the latter lines [AMD12, Jun15, Dav16] in an effort to improve both the performance and energy efficiency of certain applications.

The integration of new, formerly independent resources into the chip, enables a tighter relation with the CPU, that allows for a different use of the hardware. This may enable both new capabilities or a more convenient and efficient support for already available ones. An instance of the latter would be, for example, how vector processors evolved into vector extensions which, with the aid of the compiler, manage to extract excellent performance requiring few to no programmer intervention.

Current SoCs hold both CPU and GPU cores that share the same memory [AMD12]. This enables the use of a unified memory space between the CPU and GPU and adds the possibility of system-wide atomics, among other important features that ease the programming of heterogeneous systems. Nevertheless, even though CPUs and GPUs are closer than ever, they are not fully integrated yet. GPUs are still treated as accelerators by the programming models, following a strict host-device philosophy. For a kernel to be run using all the available resources, the programmer needs to manually partition the workload; all the challenges of co-execution remaining unchanged.

Discrete systems require the intervention of drivers to communicate with the different devices, but such requirement should not be necessary in integrated architectures. The software tools presented in the previous chapters can be used for this kind of systems, but their integration

in a single chip stands the chance of supporting co-execution in hardware, avoiding software overheads. The programmer would only have to launch a kernel and leave the hardware in charge of co-execution and load balancing, with no complex software in between, that will generate an overhead no matter how optimized it is. This opens up new possibilities regarding how problems are solved, but new limitations arise too.

- Advantage: Hardware co-execution will have much smaller overheads, which enable a finer grained work distribution.
- Advantage: New information will be available that will enable to make more educated load balancing decisions.
- Advantage: Data management should be easier, as the memory space is shared, and all the devices use the same virtual addresses.
- Limitation: Simpler load balancing algorithms will be required, as their operation will be implemented in hardware.
- Limitation: The parallel operation of CPU and GPU cores using the same memory may have negative effects on performance. This is due to the different way in which CPU and GPU cores stress the memory subsystem and the interconnection network [BKA10, ZAM+15]. Examples of this effects will be shown in Section 6.5.

By leveraging these advantages while considering the limitations, hardware co-execution represents an opportunity to better use the resources of integrated architectures and ease their programming. It paves the way towards a more transparent use of the heterogeneous system, more in line with traditional programming, in which the interaction of compiler and hardware makes for excellent performance and efficiency, with almost no programming effort. Consequently, this chapter proposes to extend the architecture of integrated CPU-GPU systems to support hardware co-execution. This will be done by adding a new module, independent from the CPU and the GPU, capable of scheduling work to both devices. The proposed design has been implemented in the gem5 simulator for its evaluation.

6.2. Design

The addition of hardware supported co-execution requires not only to devise an architecture capable of transparently distributing work among heterogeneous resources. It also involves rethinking how the integrated heterogeneous system will be programmed. This section elaborates on the most adequate way to offer co-execution to the programmer, and the underlying architecture modifications required to support it.

6.2.1. Programming support

The goal of hardware co-execution is adequately using all the resources of the heterogeneous system, while minimizing programming effort. Therefore, compiler or software assisted techniques to automatically identify what to co-execute would be the best option, as they would require no programming. However, they are way beyond the scope of this dissertation. Moreover, it would be far-fetched to take such a disruptive path as a first approach to hardware co-execution.

To keep programming effort as low as possible, the next logical step is for the programmer to specify what will be co-executed. This would imply that the code to be co-executed can be run on all the involved devices. Therefore, OpenCL represents an ideal choice, as a kernel can be run on any kind of device provided an adequate driver. Then, to support co-execution, it will be necessary to extend OpenCL to provide the programmer with a means to specify that a kernel has to be co-executed. Having chosen OpenCL as the target language for co-execution, OpenCL terminology will be used throughout the rest of this chapter. However, the proposed design could be easily extended to other programming models with support for heterogeneous systems.

As explained in Section 2.1.1, OpenCL command queues are used to launch a kernel to a specific device. This kind of operation should be preserved to respect OpenCL philosophy and ease the adaptation of code to co-execution. Therefore, it will be necessary for the OpenCL driver to provide a data structure that represents all the available resources, so the programmer can

use it for co-execution as if it represented a single device. Devices belong to contexts, which are the main data structure to manage the heterogeneous system. These are representations of available device configurations provided by the OpenCL library in collaboration with the driver. To create a context, OpenCL provides functions that take the type of the devices to be selected as an input parameter. For example, when `CL_DEVICE_TYPE_GPU` is used, a context holding all the available GPUs will be created. Such a context would provide individual access to the GPU devices available in the system, but no co-execution capabilities, as explained in Section 2.1.1.

Considering that the goal is to obtain a single device data structure that encompasses every resource, a new device type `CL_DEVICE_TYPE_HETEROGENEOUS_SYSTEM` may be defined, to express that hardware co-execution is desired. If used, the resulting context will provide the programmer with a single logical device, representing all the resources in the system. The later use of this context and the device data structure it provides will be transparently handled for hardware co-execution. As a result, the programmer will effectively deal with a single device in a classic host-device manner, while all the complex details are managed by the library, the driver and, ultimately, the hardware. This design enables effortless access to the co-execution capabilities supported by the hardware.

6.2.2. Architecture and OS support

Nowadays, the most common integrated heterogeneous system holds a number of CPU cores and GPU compute units in the same SoC. For its prevalence, this is the configuration that will be considered in this chapter. The obtained conclusions may be extended to other kinds of heterogeneous systems, but it would be necessary to evaluate if the design applies, regarding the particularities of each architecture. The names of certain hardware structures have been intentionally kept generic for the sake of clarity. This is because the names used by manufacturers are often just marketing devices that change between brands or even hardware generations.

In order to devise a strategy to adequately distribute a kernel execution between cores and compute units, lets first analyze how the OpenCL runtime handles each kind of device. This will

provide useful information to come up with a low-impact design, that leverages the capabilities already present in the hardware and OpenCL runtime.

For CPUs, the runtime spawns one thread per available core. These threads will operate as a work pool to process kernels, receiving work in the form of work-groups, which will be distributed by a thread devoted to management. Each thread will process the work-items of a work-group sequentially before passing to the next work-group. Therefore, there will be parallelism between work-groups, but not between work-items within the a work-group. This is to reduce the cost of synchronization within the work-group and to avoid cache sharing issues that may arise if a work-group was split among several cores. Note that all the threads will execute exactly the same code, as the kernel function will be the same.

On the contrary, the OpenCL runtime operates very differently when dealing with GPUs. If for CPUs the management is prominently software-base, hardware will play a very important role for GPUs. When a kernel is launched, its information is stored in a hardware queue. This includes, but is not limited to, the number of work-groups and work-items to be spawned, a pointer to the first instruction to execute and a pointer to the input parameters. That is, all the necessary information to manage the distribution of work. These registers are part of a hardware dispatcher, which is in charge of the dynamic distribution of work-groups to compute units. Each compute unit will initially get as many work-groups as made possible by its resources. Then, when a compute unit completes a work-group, it will notify the dispatcher and request a new one. A depiction of a sample kernel K1 and its later distribution to the compute units is shown in Figure 6.1. This kind of operation has its origin in discrete GPUs. However, it holds true for integrated ones, even though they are tightly knit to the CPU. The interconnection communicating them will just be faster.

6.2.3. Proposed dispatcher design

To provide support for hardware co-execution in integrated systems, it is necessary to implement a single means to manage the distribution of work among all the heterogeneous devices. The GPU dispatcher seems like a fitting candidate for the task, as it already holds all the necessary

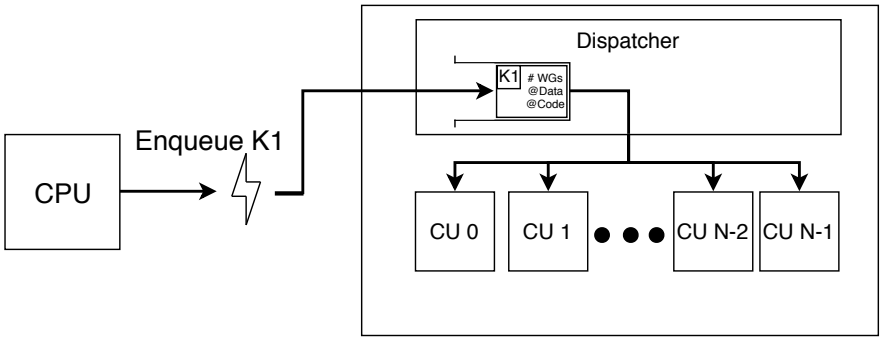


Figure 6.1: Representation of the proposed design for the support of hardware co-execution.

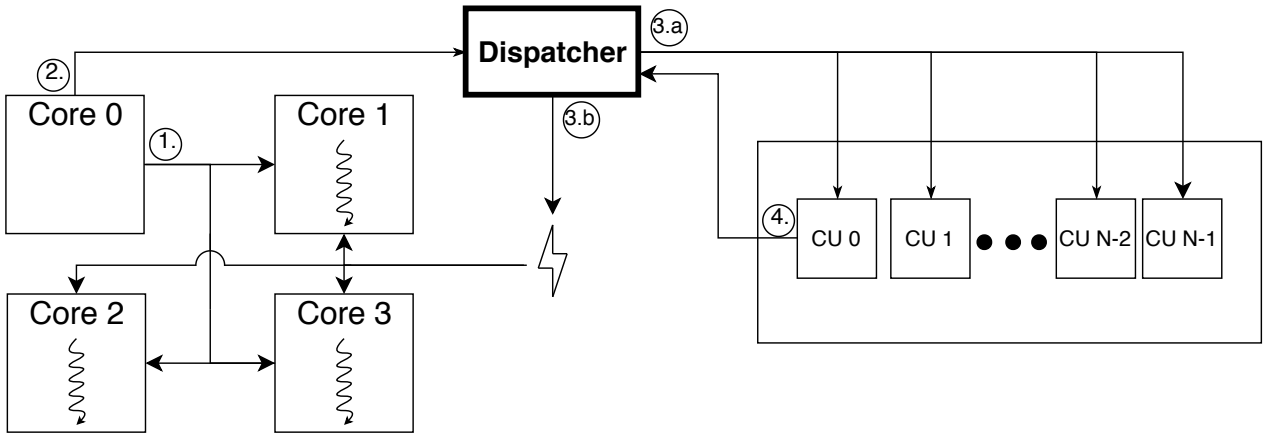


Figure 6.2: Representation of the proposed design for the support of hardware co-execution.

information for co-execution and load balancing. However, it has traditionally been part of the GPU, regarding compute units only.

Considering that integrated systems have reduced latencies and overheads and that their design is based on the sharing of certain hardware structures, it is not unreasonable to move the dispatcher away from the GPU. This new shared dispatcher design enables the transparent scheduling of work-groups to both CPU cores and compute units, while preserving the usual operation of the devices individually. Its operation is outlined next and depicted in Figure 6.2.

1. At a kernel launch, the OpenCL runtime will create one thread per core, excluding the one considered as host. This is similar to the default operation for CPUs, but no work assignation will be performed now.

2. The information of the kernel execution will be written to the dispatcher, similarly to how it is done for the GPU.

3. The dispatcher will start distributing work-groups on demand:

   a) For GPUs, work-groups are assigned to compute units until their resources are completely used. This is the default operation for the GPU.

   b) For CPUs, each thread gets assigned one work-group. In this case a system call will be necessary to specify the work-group scheduled to each core.

4. When a work-group is completed, the executing device will request more work. This will be done in hardware by the GPU compute units and through the runtime for the CPU cores.

This design strives to minimize the modifications necessary to support hardware co-execution, leveraging the resources already present in the architecture. The only important change is where the orchestration of the execution of a kernel is performed. This requires no new functionality from the devices themselves. They just have to operate as they did before, executing work-groups when required and requesting more work when idle. However, this design has an important weakness. If a work-group gets scheduled to the slowest device near the end of the execution, it will be likely to delay the completion of the whole kernel. To prevent this, a certain degree of intelligence needs to be added to the dispatcher, so it can decide whether it should schedule work-groups to a device or not. This will be explained next.

6.2.4. Automatic co-execution throttling

Depending on the co-executed kernel, the computing speed difference between the CPU and the GPU may be significant. As a consequence, a careless work-group distribution may result in the slowest device being assigned a work-group at the wrong time, effectively delaying the completion of the whole kernel. To prevent this, the dispatcher needs a means to evaluate the time that a device will take to execute a work-group, so it can make a decision on whether it is sensible to perform the dispatch. In general, in a two device scenario, a work-group should only be scheduled to a device if its execution time is smaller than that of running all the remaining workload on the other device. This is, a work-group should be dispatched to device d_1 if the

following expression is satisfied, where $T(d_i, x)$ represents the time to execute x work-groups in device d_i and G_r is the number of remaining work-groups:

$$T(d_1, 1) < T(d_2, G_r) \tag{6.1}$$

Considering that both the CPU and the GPU are parallel devices, T could be defined as the number of batches of work-groups that will be executed sequentially, multiplied by the time to execute a single work-group in device d_i. This can be expressed as:

$$T(d_i, x) = \lceil \frac{x}{pe_i \cdot MaxWg_i} \rceil \cdot T_{Wg_i} \tag{6.2}$$

Where each of the parameters will be defined as follows:

1. pe_i is the number of processing elements of device d_i.
2. $MaxWg_i$ is the number of work-groups that a processing element of device d_i can compute in parallel.
3. T_{Wg_i} is the time to execute one work-group in device d_i.

Parameter pe is straightforward and known to the dispatcher. The CPU will have as many processing elements as cores, and the GPU will have as many as compute units. $MaxWg$ is also known to the dispatcher. Each CPU core is capable of executing a single work-group. GPU compute units, in turn, can execute a certain number of work-groups in parallel, which depends on the amount of resources used by each particular kernel, such as registers or memory. This value may be discovered by the dispatcher, as it initially schedules work-groups to the compute units until they are filled. The execution time for a work-group and how it is handled by the dispatcher require a detailed, step by step, explanation, which is provided next.

Device execution time per work-group

The GPU dispatcher does not have the necessary resources to monitor the execution of work-groups. Consequently, to be able to predict the execution time of a work-group in the CPU or GPU, the dispatcher needs new hardware. This functionality will be implemented by extending the dispatcher with two registers, the *CPU Time per Work-group Register* (CTWR) and the *GPU Time per Work-group Register* (GTWR), which will hold the observed time to execute a work-group in the CPU and GPU respectively.

To update the values of the CTWR and GTWR, two hardware tables will be used. The *CPU Work-group Log* table (CWL) will have one entry per CPU core, which will hold the timestamp for the last work-group dispatched to the core. The *GPU Work-group Log* table (GWL) will be similarly defined, but it has to consider that GPUs may be executing more than one work-group simultaneously. For this table to monitor all the work-groups running in the GPU, it would need to have as many entries as the number of compute units, times the maximum amount of work-groups supported by a compute unit. Note that this value is likely to be greater than $MaxWgs$, as it is absolute maximum number of supported work-groups for a kernel that uses minimum resources. Current architectures support a maximum of 16 work-groups per compute unit, and contain total of 11 compute units. A table with 176 entries may be too big to be efficiently implemented in hardware. For this reason, the GWL will only monitor one of the compute units of the GPU. This does not represent a significant loss of information, as the GPU is usually the fastest device, so by monitoring one of its compute units there is little risk of scheduling excessive work to the rest. Moreover, work-groups are often scheduled in round-robin, so those assigned to any compute unit should be equally representative of the behavior of the whole kernel. As a result, the GWL will have 16 entries, each one holding a timestamp and a work-group identification. These tables will be updated each time a work-group gets dispatched.

When the dispatcher gets notified about the completion of a work-group it will use the CWL or GWL to update the CTWR or GTWR as corresponding. To do so, it will subtract the current timestamp from the one stored in the table entry corresponding to the device and work-group that has been completed. To further clarify this, Figure 6.3 depicts the dispatcher for a simple system with only two cores and compute units that can execute a maximum of four work-groups

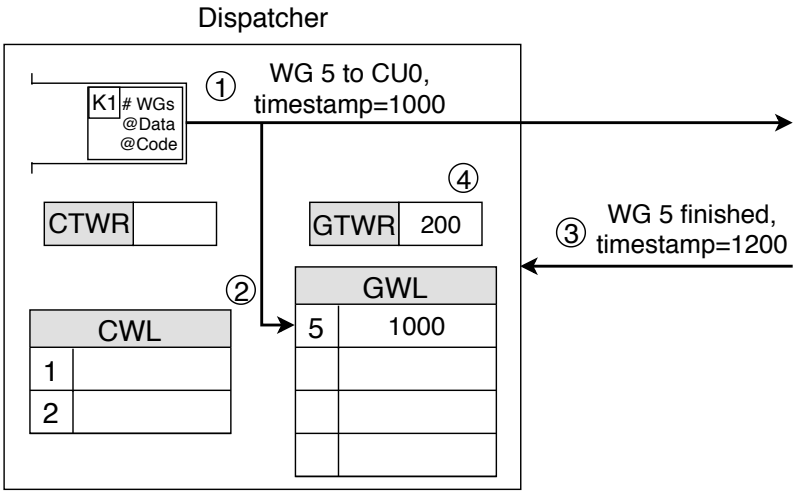


Figure 6.3: Example of the operation of the dispatcher in detail.

simultaneously. In the example, work-group 5 gets scheduled to compute unit 0, which is the one monitored by the GWL, at timestamp 1000, so a GWL table entry is written with this data. A while later, at timestamp 1200, the dispatcher is notified that work-group 5 has finished, so it updates the GWTR value to $1200 - 1000 = 200$.

By using the values of CTWR and GTWR and Equations 6.1 and 6.2 the dispatcher can decide whether it should schedule a work-group to a device or not. This keeps slow devices from limiting the performance obtained from co-execution.

6.3. Implementation

The design proposed in Section 6.2 has been implemented in the gem5 simulator [BBB+11] version 2.0 for its evaluation. This is a powerful event-driven simulation platform that can model integrated CPU-GPU architectures, as introduced in Section 2.2. The implementation of support for hardware co-execution required changes to the OpenCL library used by the simulator and to the architecture itself, which will be introduced in this section.

6.3.1. OpenCL library

The OpenCL implementation used by gem5 does not use a regular driver, but a backend executed in syscall emulation as explained in Section 2.2. Therefore, the application to be simulated has to be compiled using a special OpenCL library that accommodates to the needs of the backend. In order to extend OpenCL with the scheme for hardware co-execution presented in 6.2.1, it is necessary to study the library considering that every step in the outline of an OpenCL program may require modifications. This includes:

- The creation of contexts and command queues.
- The management of data.
- The compilation and launch of kernels.

However, these three items will be straightforward to adapt for co-execution. This is because of the simplified OpenCL library required by the gem5 backend.

The gem5 simulator can model systems that integrate CPU cores and a GPU, but only considers OpenCL execution on the GPU. Therefore, as the library is designed to strictly accommodate to the needs of the simulator, the functions to query for platforms, contexts or devices have a hardcoded return representing that only one GPU is available. The proposed design for co-execution uses a single device to represent all the available devices. It is just necessary to store that they are of type `CL_DEVICE_TYPE_HETEROGENEOUS_SYSTEM` for later use, when the user specifies so in the functions to create them.

With respect to data, the library disregards the type of device to create, read or write buffers, as it only targets the GPU. The simulated system has a shared address space between CPU and GPU, so the data management originally performed by the library for the GPU will also be valid for the CPU cores.

Lastly, the management of kernels does not require any device type dependent action from the library. It is just necessary to pass the information of the device type to the simulator.

To do so, the `HSAQueueEntry` data structure, has been extended to hold the device type. `HSAQueueEntries` are the gem5 representation of a workload to be computed by the GPU, including a pointer to its code, its arguments, number of work-groups and work-group size.

However, this ease of implementation is a double-edged sword. It comes from the library being tailored to the needs of gem5, which only targets the execution of OpenCL kernels on GPUs. This will prove a challenge regarding the participation of the CPU cores in co-execution, which will be explained in the next section.

6.3.2. OpenCL workloads on CPU cores

Gem5 does not support the execution of OpenCL on CPU cores. However, to achieve co-execution, the CPU has to be able to compute a portion of the kernel. Implementing full OpenCL support for CPUs in gem5 would be the best option. However, it would represent an effort way beyond the scope of this thesis. This is because gem5 x86 cores can only execute x86 binaries, and there is no compiler that generates a standalone executable from the code of a kernel. Consequently, for full OpenCL support, it would be necessary to devise a scheme to enable the cores to execute one of the formats generated by CLOC, the offline kernel compiler used with gem5. This is, either an HSA Object or hsail code, which is an assembly language for heterogeneous systems. Any of the options represents a daunting amount of work, potentially requiring to develop a new type of gem5 core from the ground up.

Therefore, it is necessary to find a workaround to enable heterogeneous co-execution in gem5. The solution will be to use x86 binaries for the cores, representing an equivalent program to the OpenCL kernel to co-execute. A program will be considered equivalent to a kernel if it meets the following two requirements:

1. It receives the same arguments as the kernel, plus extra ones to determine the work-group to execute.
2. The execution for a work-group produces the same output data that the kernel would produce.

To further clarify this concept, Figure 6.4a shows a simple kernel for the addition of two vectors and an equivalent C program. Both the program and the kernel receive vectors `a`, `b`, `c` and the number of elements of the vectors `numElem` as arguments. The C program also takes the id of the OpenCL work-group to execute and the size of the work-groups, in order to identify the part of the output vector that it has to compute for. For simplicity, the kernel of this example uses a 1D kernel, but more dimensions would be equivalently handled. Similarly, other OpenCL functionalities, such as local memory or barriers, can be easily addressed in the equivalent C program. Note that, for a given work-group, both the kernel and the program would produce exactly the same portion of the output vector `c`.

This approach enables the simulated CPU cores to emulate the execution of certain work-groups of an OpenCL kernel, as required for co-execution. The simulator will receive an OpenCL program, its associated kernel and the equivalent x86 binary, and transparently launch work-groups to GPU compute units and CPU cores. This will be thanks to the changes to gem5 detailed in the next section.

6.3.3. Architecture

As explained in Section 6.2.2, the main architecture element for co-execution will be the work-group dispatcher. Gem5 models it using the `GpuDispatcher` class, which holds a set of `Shader` objects that are the gem5 representation of the GPU compute units. This class has an `exec` function, which is executed every clock cycle of the simulated system and is in charge of distributing work-groups to those `Shader` objects that have free resources to accept them. The work-groups are taken from `HSAQueueEntry` objects that are directly written by the OpenCL library to an address known by the simulator.

In order to support co-execution, the `GpuDispatcher` class, now just named `Dispatcher`, needs to be extended. First, it needs to hold a reference to the available CPU cores in the system, which will get assigned work-groups, when idle, in a new `exec` function that considers not only compute units but also CPU cores. The latter will be assigned work by using the new `addWorkload` function implemented in the `FullO3CPU` class, which models an out of order

```
__kernel void addVectors(__global float *a,
                         __global float *b,
                    __global float *c,
                    int numElem)
{
    int gid = get_global_id(0);
    if(gid<numElem)
    {
        int z=gid*2;
        c[gid] = a[gid] + b[gid];
    }
}
```

(a) OpenCL vector addition kernel.

```
int main(int argc, char * argv[])
{
    if(argc==7)
    {
        int wgSizeX=atoi(argv[1]);
        int wgIdX=atoi(argv[2]);

        float *a=(float *) atol(argv[3]);
        float *b=(float *) atol(argv[4]);
        float *c=(float *) atol(argv[5]);
        int numElem=(int) atoi(argv[6]);

        int start=wgIdX*wgSizeX;
        int end=start+wgSizeX;
        if(end>numElem)
        {
            end=numElem;
        }

        int i;
        for(i=ini;i<fin;i++)
        {
            int gid=i;
            c[gid]=a[gid] + b[gid];
        }
    }
    else
    {
        printf("Wrong number of arguments\n")
    }
}
```

(b) Equivalent C program for the vector addition.

Figure 6.4: Code for a vector addition OpenCL kernel and equivalent C program.

CPU core. The dispatcher will use this function to specify the arguments that will be passed to the equivalent x86 binary and trigger its execution. Note that `HSAQueueEntry` objects hold all the necessary arguments, including the information on work-groups and the kernel arguments. Also note that gem5 models a shared address space between CPU and GPU, so the memory addresses used by the kernel can be directly passed to the equivalent x86 binary as arguments. The dispatcher can still choose between co-execution or regular GPU operation using the `HSAQueueEntry` object too, which was extended in Section 6.3.1 to contain the device type of the queue in which the kernel was launched.

Regarding CPU cores, one will operate as usual, executing the OpenCL application and not taking part in the distribution of work made by the dispatcher. The rest will be created with no associated workload. This is because assigning a workload to a core implies that it will start running immediately, which is not the desired behavior for the cores that will receive OpenCL work-groups. This implies that these cores will only be partially initialized, as the initialization of certain structures requires information on the workload. When the `addWorkload` function is called, the core will receive a workload and the remaining initialization steps will be performed. Then, the equivalent x86 binary will start executing using the arguments specified by the dispatcher. When it finishes, a new `notifyWgComplCPU` function, implemented in the dispatcher, will be called, to inform that the work-group has been completed. Subsequent calls to the `addWorkload` function for a core, result in the selective re-initialization of the structures that require so to execute a new work-group. This includes the reset of certain pipeline stages, free register lists and rename maps and the update of the page table for the binary.

The resulting structure for the simulator is outlined at a very high level in Figure 6.5, showing only the structures and functions relevant to co-execution. These changes to gem5 provide support for hardware heterogeneous co-execution, autonomously managing the distribution of the workload of a single OpenCL kernel among all the available devices of the simulated system.

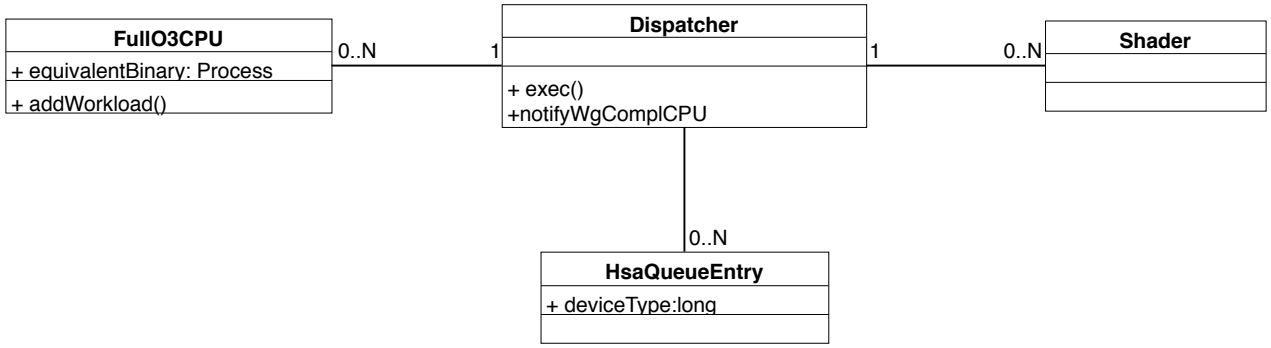


Figure 6.5: Simplified class diagram of the parts of gem5 directly involved in co-execution.

6.4. Methodology

The aim of this section is introducing the aspects of the methodology particular to this chapter that were not presented in Section 1.5. This includes a description of the system modelled in the simulator, the kernels used in the evaluation and the selected metrics.

6.4.1. Modelled architecture

To evaluate the proposed dispatcher, gem5 has been configured to model an architecture similar to that of the AMD Ryzen 5 2400G. This is an integrated SoC released in 2018 that holds 4 CPU cores and 11 GPU compute units. Each core runs at 3.6 GHz and is capable of executing up to 2 threads via SMT. Considering that gem5 does not support multithreading, 8 cores have been used for the evaluation to emulate having 8 threads in total. Compute units run at 1250 MHz. Regarding the memory hierarchy, cores and compute units only share the main memory. The cores have a private L1 Cache and shared L2 and L3 caches. Compute units have a private L1 cache and a shared L2 cache. The CPU and the GPU share the same memory space. Note that 1 of the 8 CPU cores will be kept busy executing the OpenCL application, so the dispatcher will distribute work-groups to the remaining 7 cores.

Table 6.1: Parameters for each benchmark

Benchmark	Type	Problem size	Local work size
Binomial	Regular	7500	128
Mandelbrot	Regular	2048×2048	128
Gaussian	Regular	1000×1000, 21×21	128
LavaMD	Regular	8	128
Rap	Irregular	182×182	64
SpMV	Irregular	256000, 512000	64

6.4.2. Selected applications

Six kernels have been chosen for the experiments, 3 of which exhibit regular behaviour. Binomial generates binomial lattices, useful for option pricing in financial software. Mandelbrot implements a blocked algorithm to compute a Mandelbrot set. Gaussian calculates the Gaussian blur of an image, commonly found in image and video processing software. The other three kernels are irregular. LavaMD calculates particle potential and relocation due to mutual forces between particles within a large 3D space. Rap is an implementation of the Resource Allocation Problem [ACBA10]. There is a certain pattern in the irregularity of RAP, because each successive package represents a bigger amount of work than the previous. Finally, SpMV performs a sparse matrix vector multiplication. Table 6.1 shows the parameters used for each of the applications. Note that, in general, smaller problem sizes have been used for this evaluation than in the previous chapters. This is to keep simulation times within reasonable margins. For each kernel, an equivalent C program was developed that is executed by the CPU cores.

6.4.3. Evaluated metrics

The selected metrics to study performance are based on the execution time of the kernels. Communication times have not been considered in this case because they should not change whether co-executing or using a single device. This is because the CPU and GPU share the same memory, so there is no need to perform any extra data transfer to the devices. Two metrics that use execution times have been evaluated: speedup and co-execution rate.

The speedup has been calculated considering an execution that only uses the GPU as the

baseline. This represents the most common use of OpenCL. The baseline is compared to the effortless use of the GPU compute units and CPU cores using hardware supported co-execution. Note that CPU cores are significantly slower than GPU compute units for the evaluated kernels, so speedups will just be as high as a fraction over one, even though seven new cores will be part of the co-execution.

The co-execution rate is defined as the ratio of the response time of the first device to conclude its work and that of the last. This metric does not consider the individual cores or compute units, but the CPU and GPU as a whole. This is because, due to co-execution throttling, some processing elements may stop getting scheduled work-groups earlier than others to avoid delays. However, as long as one core and one compute unit are computing simultaneously, co-execution is being used. The ideal value for this metric is one, meaning that co-execution was used for the whole duration of the kernel. Deviations from this value may be due to performance differences between CPU and GPU or kernel irregularity.

Additionally, current integrated systems are based on the sharing of certain computing resources. In the proposed architecture, the CPU and the GPU share the main memory. Consequently, three extra metrics have been used to try to better understand the impact of co-execution on this critical element: the variation of the average GPU latency to access the main memory, the variation of the GPU L2 miss rate and the variation of the total GPU L2 misses.

6.5. Evaluation

This section evaluates the proposed design for hardware supported heterogeneous co-execution in integrated systems. The aim of the experiments is to answer the following questions:

- Does the proposed design achieve co-execution?
- Does co-execution have a potential for performance gains in integrated systems?
- Do any undesired effects appear as a result of co-execution and the sharing of hardware structures?

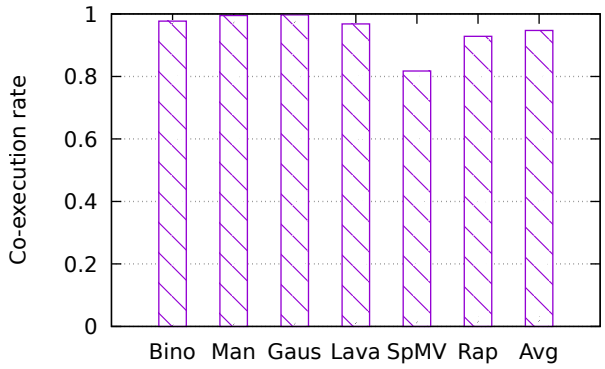


Figure 6.6: Load balance for the evaluated for the evaluated benchmarks using hardware supported co-execution.

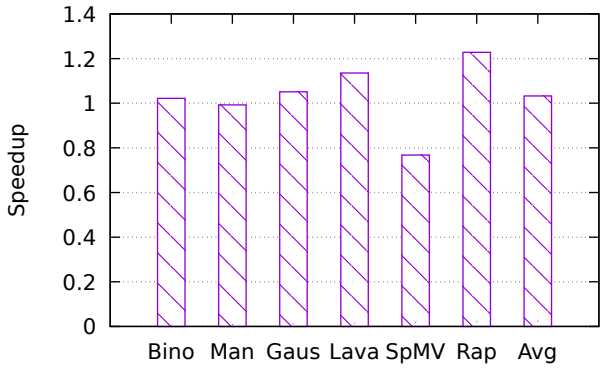


Figure 6.7: Speedup for the evaluated for the evaluated benchmarks using hardware supported co-execution.

Co-execution rate

To evaluate whether the proposed dispatcher achieves co-execution, Figure 6.6 depicts the co-execution rate of the evaluated benchmarks. These results show that co-execution is indeed achieved, with an average of 95% of the duration the kernels spent in co-execution. This is a very good result considering the computing speed difference between CPU and GPU. Values go as high as 99.5% for Mandelbrot and Gaussian, achieving a near perfect co-execution rate. SpMV achieves the lowest value, but co-execution is still leveraged for 82% of the duration of the kernel. This result is due to the irregularity of this application.

Performance

Performance has been evaluated by using the speedups obtained thanks to co-execution with respect to using only the GPU, which is the fastest device. They are displayed in Figure 6.7. The results show that co-execution is beneficial in 4 of the 6 evaluated benchmarks. Binomial and Gaussian obtain a speedup of roughly 1.02 and 1.05 respectively. Considering the co-execution rates for this benchmarks, shown in Figure 6.6, this means that the speedup obtained is close to the maximum achievable, as CPUs are contributing most of the time. However, the performance difference between CPU and GPU for these two kernels is significant. The speedups obtained for LavaMD and Rap are 1.14 and 1.23 respectively, as the CPU has a higher computing speed for these benchmarks, so it can contribute more work. This excellent performance is achieved even

though the co-execution rates are slightly lower, due to the irregularity of these applications. The dispatcher correctly identifies that CPU cores would represent a delay for the last part of the benchmark, so co-execution gets throttled.

Of the benchmarks in which co-execution represents a performance loss, Mandelbrot is a very interesting case, as it achieves the best co-execution rate, but performs slightly worse than the baseline, with a speedup of 0.99. Regarding SpMV, its poor performance can be traced to two factors. First, its irregularity results in an inaccurate estimation of the work-group execution time. Co-execution is not throttled and the CPU, which is the slowest device, finishes last, causing a delay. The second factor and the reason for the performance of Mandelbrot, are related to how these kernels access memory, which will be explained in the next section. Overall, co-execution represents an average speedup of 1.03.

Memory access analysis

Co-execution is based on the collaboration of several devices computing for the same workload. This implies that, in the proposed integrated architecture, more devices will be simultaneously stressing the memory system. This could lead to contention in the access to the main memory. To evaluate this, Figure 6.8 shows the variation of the GPU latency to main memory when co-executing, with respect to a GPU-only execution. Consequently, if co-execution generated no contention, a variation of 0 would be obtained. Results show that this is not the case. All the benchmarks but LavaMD show an increase in the GPU memory access latency, with an average of 29% and going as high as 120% for Gaussian. Regarding LavaMD, its latency reduction is due to a significant decrease in the total number of GPU L2 misses. This metric is depicted in Figure 6.9. Note that the two greatest miss reductions are obtained in LavaMD and Rap, which are also the two applications with the best results regarding speedup and memory access latency. This leads to the conclusion that the kernels that best co-execute are those that reduce the GPU L2 misses the most. However, performance gains are still achieved through the co-execution of applications, such as Gaussian, with small total miss variation and memory latency growths. This is because certain applications hide latencies better than others.

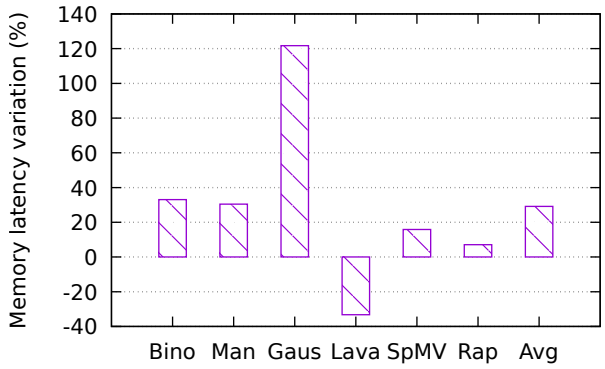


Figure 6.8: Memory access latency variation for the GPU when co-executing.

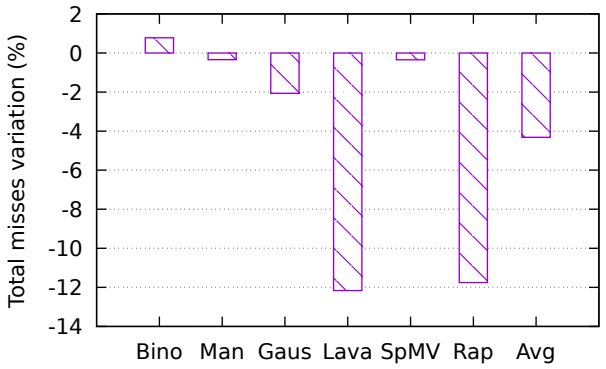


Figure 6.9: Total L2 miss variation for the GPU when co-executing.

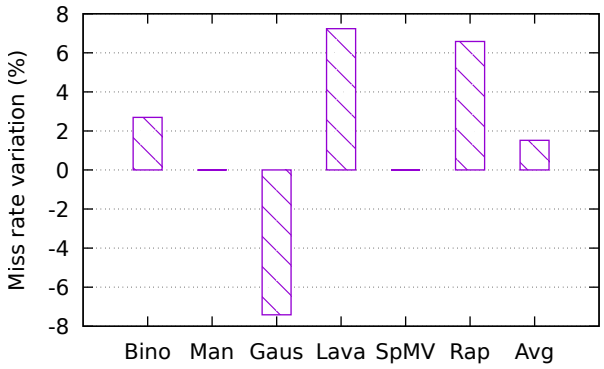


Figure 6.10: L2 miss rate variation for the GPU when co-executing.

To evaluate this, Figure 6.10 shows the GPU L2 miss rate variation when comparing co-execution to a GPU-only execution. This shows that co-execution has a variable impact on the L2 miss rate. The miss rate for Gaussian is reduced by 7.4%, which indicates that data reuse has improved. On the contrary, Rap, which achieves the greatest speedup, also shows a miss rate increase of 6.5%. In these two cases, memory latency is hidden through multi-threading, which is the foundation of GPU computing. Regarding SpMV and Mandelbrot, they show no miss rate variation. However, this is because they never hit in L2, not even in a GPU-only execution. This is because these kernels have limited data reuse, which is fully exploited in the L1. This, together with the low compute to memory access ratio of this kernels, is the reason why they do not benefit from co-execution: they are highly impacted by main memory latencies, which are not properly reduced by the L2. Lastly, the case of Binomial is very special. It shows a small increase in the L2 miss rate but, looking at the total misses in Figure 6.9, also an increase in the total misses with respect to a GPU-only execution. In other words,

when co-executing, the GPU is doing less work, but generating more misses overall. These extra misses can only be attributed to the interaction of the CPU and the GPU. Consequently, considering that they only share the main memory and that work-groups produce independent results, this behavior can only be caused by one factor: false-sharing. This is an effect caused by the coherence protocol when two pieces of data belonging to the same memory block are written by two different devices. Even though the data is not actually shared, the block is, so the writes generate invalidations and, ultimately, misses.

6.6. Conclusions

This chapter presents a novel design to provide hardware support for co-execution in integrated heterogeneous systems. It is based on a new dispatcher, independent from both the CPU and GPU, that is implemented in the SoC and distributes OpenCL work-groups to cores and compute units indistinctly. By tracking the execution of work-groups, the dispatcher is capable of throttling co-execution to keep a device from generating delays if it is deemed too slow.

The new design was implemented for its evaluation in the gem5 simulator. Experimental results show that the proposed design achieves co-execution. Significant performance gains, as high as 23%, are possible. However, the sharing of the memory between CPU and GPU also may produce undesired effects such as contention, which harm performance. Consequently, to benefit from co-execution, the workloads need to adequately hide the increased memory contention that may arise, either through multithreading or cache. Coherence protocol related issues such as false-sharing were observed. These results open up a new line of research to alleviate the pressure on memory associated to co-execution. A possibility would be to evaluate memory latencies at runtime, to throttle co-execution if they get excessively high. Other schemes could also be devised to improve memory usage and avoid this throttling.

Chapter 7

Conclusions and Future Work

7.1. Conclusions

The ever-growing need for parallelism and the boom of workloads that manage high volumes of data, such as deep learning, has made heterogeneous architectures prevalent due to their outstanding performance and energy consumption. However, while systems have grown, integrating more and more CPUs and accelerators, programming models have continued to consider devices as isolated entities, favouring host-device approaches and task parallelism. This has turned heterogeneous co-execution into a a second class citizen, forcing the programmer to manually handle it if desired. Nevertheless, accelerators base their success on data-parallelism, so the workloads they execute lend themselves to co-execution. For this reason, it represents a viable option to extract all the potential capabilities of the devices available in the system. However, to be useful, heterogeneous co-execution needs to be effortless, representing equivalent work to using a single device.

This dissertation proposes several techniques to enable effortless co-execution in heterogeneous systems. Two main concepts need to be addressed in order to achieve co-execution: abstraction and load balancing. The former refers to the relation of the programmer to the management of the heterogeneous system. For effortless co-execution, ways to manage the whole system through a single interface should be provided, disregarding the specific details of the concrete

underlying system. This eases programming and guarantees portability. The latter involves how the workload is split among the available devices. To achieve a successful co-execution, the response time of the devices should be equalized, so they all contribute useful work for the whole duration of the workload and idle times are minimized. This is a complex task that should not be left to the programmer, as it entails considering the behavior of the co-executed workload and the computing speed of the available devices. This dissertation tackles these two concepts in several ways, both from the hardware and the software point of view.

First, it introduces HGuided and Sigmoid, two novel load balancing algorithms specially designed for heterogeneous co-execution. Through its use of decreasing package sizes and its attention to computing speeds, HGuided manages to successfully balance both regular and irregular workloads, obtaining excellent performance and energy efficiency. However, it requires certain parameters from the user to operate. In an effort to make load balancing easier, Sigmoid is an uninformed algorithm that eliminates the need for user parameters by monitoring co-execution. It uses a function derived from the sigmoid to calculate package sizes and it tunes its internal parameters to adapt to the behavior of the kernels. The result is outstanding performance and efficiency requiring no action from the user.

Maat, an OpenCL-based co-execution library was presented in this dissertation. This library makes co-execution significantly simpler by providing the illusion of a single device that represents the aggregate computing power of all the available resources. This is achieved through abstractions built on OpenCL data structures, while their approach to parallel programming is preserved. This eases the adaptation of pre-existing OpenCL applications for co-execution. Maat also improves the portability of applications, as its abstractions guarantee that an application that uses all the devices in a system will also use them in a different one. Evaluation showed that Maat achieves close to ideal performance and significant energy savings for the evaluated benchmarks, best results being obtained using the Sigmoid algorithm, closely followed by HGuided.

Support for heterogeneous co-execution was added to a task-based programming model. This kind of programming models ease the development of applications by offering high level ab-

stractions that hide the complex details of parallel programming. However, they lack regarding co-execution. A module was added to OmpSs to support co-execution. This was done with low impact on the original OmpSs infrastructure and while preserving its approach to parallel programming. In the same vein, regarding load balancing, the implemented algorithms respect the approach of OpenMP, which lies at the heart of OmpSs. Static produces a single package per device, with a size proportional to its computing speed in relation to that of the whole heterogeneous system. Dynamic, on the contrary, generates many equally sized packages that are scheduled at runtime. HGuided works similarly, but uses decreasing package sizes to reduce overheads. Lastly, Auto-Tune is an evolution of HGuided that monitors execution and eliminates the need for programmer parameters. Evaluation shows that Auto-Tune achieves the best overall performance, almost equalling Static for regular workloads and successfully balancing irregular ones. Moreover it requires no parameters from the programmer. Co-execution proved to be beneficial for task-based programming, as it helps to effortlessly leverage all the computing power of the system through collaboration.

A new dispatcher was designed to provide hardware support for co-execution in integrated heterogeneous systems. After elaborating on the most adequate way to leverage hardware supported co-execution from the programming, the new dispatcher was presented. It operates by scheduling OpenCL work-groups to CPU cores and GPU compute units indistinctly and monitoring the time per work-group for each device. This enables the dispatcher to throttle co-execution if a device is likely to cause a delay. The evaluation, performed using the gem5 simulator, shows that the dispatcher achieves co-execution and that significant performance gains can be obtained. However certain challenges also arise, such as contention produced by the increased pressure on memory produced by co-execution.

7.2. Future Work

A dissertation not only represents an answer to a question, but also the opening of new research lines that build upon the work carried out for the duration of the PhD. Some interesting lines

of work are outlined next.

- *Extending Maat to work with multiple kernels:* Maat targets heterogeneous co-execution and data parallelism, completely disregarding task parallelism. It would be interesting to extend it to combine co-execution with the scheduling of other independent kernels. This would require new load balancing algorithms to consider the behavior of the different workloads in execution.

- *Load balancing for energy consumption:* Performance and energy consumption are often related. However, in certain instances, the work partition that maximizes performance does not minimize the energy consumed. All the load balancing techniques presented in this dissertation target performance. Algorithms that focus on energy would also be enriching, as it is currently one the limiting factors in the design of computing systems, ranging from supercomputers to mobile devices.

- *Preprocessing-based load balancing:* HGuided and Sigmoid are agnostic algorithms. They operate the same regardless of the co-executed workload. Better load balancing could be achieved by analyzing the workload to co-execute and taking different decisions based on its complexity or the kind of operations it uses. Machine learning based techniques could also be used to help in the decision making process.

- *A priori co-execution throttling:* The techniques proposed in this dissertation operate after a kernel is launched. However, if one of the devices is too slow for the workload, it will already have been scheduled work that will produce a delay. Methods to a priori decide not to use slow devices in co-execution would avoid this issue. These techniques could be implemented either as part of new load balancing algorithms or in hardware.

- *Extending the hardware dispatcher to better handle irregularity:* The proposed dispatcher uses co-execution throttling to prevent delays. However, kernels with varying work-group execution times may produce inaccurate throttles. A dispatcher capable of detecting irregularity and acting accordingly would improve the support for hardware co-execution.

- *Optimizations to reduce memory contention:* Hardware supported co-execution has shown to increase the average memory access latency. CPU cores and GPU compute units should be better orchestrated to prevent this effect, as it may significantly harm performance. This may include modifications to the operation of the dispatcher or to the memory hierarchy itself.

List of Publications

- B. Pérez, E. Stafford, J. L. Bosque, and R. Beivide. Sigmoid: an auto-tuned load balancing algorithm for heterogeneous systems. In *Transactions on Parallel and Distributed Systems*. Submitted pending review.

- B. Pérez, E. Stafford, J.L. Bosque, R. Beivide, S. Mateo, X. Teruel, X. Martorell, and E. Ayguadé. Auto-tuned opencl kernel co-execution in ompss for heterogeneous systems. *Journal of Parallel and Distributed Computing*, 125:45 – 57, 2019.

- Raul Nozal, Borja Pérez, José Luis Bosque and Ramón. Load balancing in a heterogeneous world: CPU-Xeon Phi co-execution of data-parallel kernels. *The Journal of Supercomputing*, The Journal of Supercomputing, published on-line.

- Borja Pérez, Esteban Stafford, José Luis Bosque, and Ramón Beivide. Energy efficiency of load balancing for data-parallel applications in heterogeneous systems. *The Journal of Supercomputing*, 73(1):330–342, Jan 2017.

- B. Pérez, E. Stafford, J. L. Bosque, R. Beivide, S. Mateo, X. Teruel, X. Martorell, and E. Ayguadé. Extending ompss for opencl kernel co-execution in heterogeneous systems. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 1–8, Oct 2017.

- E. Stafford, B. Pérez, J. L. Bosque, R. Beivide and M. Valero. To distribute or not to distribute: the question of load balancing for performance or energy. In *23th International European Conference on Parallel and Distributed Computing, Euro-Par 2017*, Santiago de Compostela, Spain, August, 2017.

- R. Nozal, B. Pérez and J. L. Bosque. Towards co-execution of massive data-parallel OpenCL kernels on CPU and Intel Xeon Phi. In *Proceedings of the 17th Internation al Conference on Mathematic al Methods in Science and Engineering*, (CMMSE), Cádiz, Spain, 2017.

- Borja Pérez, José Luis Bosque, and Ramón Beivide. Simplifying programming and load balancing of data parallel applications on heterogeneous systems. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, GPGPU '16, pages 42–51, New York, NY, USA, 2016. ACM.

- B. Pérez, E. Stafford, J. L. Bosque, and R. Beivide. Energy efficiency evaluation in heterogeneous computers. In *Emerging Technology Conference*, 2016.

- B. Pérez, E. Stafford, J. L. Bosque, and R. Beivide. An energy evaluation of data-parallel applications in heterogeneous systems. In *Proceedings of the 16th Internation al Conference on Mathematic al Methods in Science and Engineering*, Cádiz, Spain, 2016.

- G. Ortega, E.M. Garzón, B. Pérez, E. Stafford, J. L. Bosque, and R. Beivide. Evaluation of Load Balancing Methods for Irregular Data Parallel Applications on Heterogeneous Systems. In *Proceedings of the 15th Internation al Conference on Mathematic al Methods in Science and Engineering*, (CMMSE), Cádiz, Spain, 2015.

References

[ACBA10] Alejandro Acosta, Robert Corujo, Vicente Blanco, and Francisco Almeida. Dynamic load balancing on heterogeneous multicore/multiGPU systems. In Waleed W. Smari and John P. McIntire, editors, *HPCS*, pages 467–476. IEEE, 2010.

[AMD12] AMD. Amd graphics cores next (gcn) architecture, 2012. Last accessed April 2019.

[ANE+16] N. Agarwal, D. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler. Selective gpu caches to eliminate cpu-gpu hw cache coherence. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 494–506, March 2016.

[APnBF16] Ashwin M. Aji, Antonio J. Peña, Pavan Balaji, and Wu-chun Feng. Multicl. *Parallel Comput.*, 58(C):37–55, October 2016.

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[BBG13] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, January 2013.

[BBK17] T. Beri, S. Bansal, and S. Kumar. The unicorn runtime: Efficient distributed shared memory programming for hybrid cpu-gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1518–1534, May 2017.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[BKA10] Ali Bakhoda, John Kim, and Tor M. Aamodt. Throughput-effective on-chip networks for manycore accelerators. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 421–432, Washington, DC, USA, 2010. IEEE Computer Society.

[BMB15] A. Gutierrez B. M. Beckmann. The amd gem5 apu simulator: Modeling heterogeneous systems in gem5, 2015.

[BSCJ13] Michael Boyer, Kevin Skadron, Shuai Che, and Nuwan Jayasena. Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability. In *Proc. of the ACM International Conference on Computing Frontiers*, pages 21:1–21:10, New York, NY, USA, 2013. ACM.

[CAM+18] E. Castillo, L. Alvarez, M. Moreto, M. Casas, E. Vallejo, J. L. Bosque, R. Beivide, and M. Valero. Architectural support for task dependence management with flexible software scheduling. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 283–295, Feb 2018.

[CCBB15] Emilio Castillo, Cristóbal Camarero, Ana Borrego, and Jose Luis Bosque. Financial applications on multi-cpu and multi-gpu architectures. *J. Supercomput.*, 71(2):729–739, February 2015.

[CGS+15] J. Cabezas, I. Gelado, J. E. Stone, N. Navarro, D. B. Kirk, and W. m. Hwu. Runtime and architecture support for efficient data exchange in multi-accelerator applications. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1405–1418, May 2015.

[CMC+16] E. Castillo, M. Moreto, M. Casas, L. Alvarez, E. Vallejo, K. Chronaki, R. Badia, J. L. Bosque, R. Beivide, E. Ayguade, J. Labarta, and M. Valero. Cata: Criticality aware task acceleration for multicore processors. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 413–422, May 2016.

[DAB+11] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[Dav16] J. Davies. The bifrost gpu architecture and the arm mali-g71 gpu. In *2016 IEEE Hot Chips 28 Symposium (HCS)*, pages 1–31, Aug 2016.

[DFKE07] K. Dabov, A. Foi, V. Katkovnik, and K. Egiazarian. Image denoising by sparse 3-d transform-domain collaborative filtering. *IEEE Transactions on Image Processing*, 16(8):2080–2095, Aug 2007.

[DGNGT+19] Maria Angelica Davila Guzman, Raul Nozal, Ruben Gran Tejero, Maria Villarroya-Gaudo, Dario Suarez Garcia, and Jose Luis Bosque. Cooperative cpu, gpu and fpga heterogeneous execution with enginecl. *The Journal of Supercomputing*, 2019.

[dlLTBR12] Carlos S. de la Lama, Pablo Toharia, Jose Luis Bosque, and Oscar D. Robles. Static multi-device load balancing for opencl. In *Proc. of ISPA*, pages 675–682. IEEE Computer Society, 2012.

[DMSD16] Bryan Donyanavard, Tiago Mück, Santanu Sarma, and Nikil Dutt. Sparta: Runtime task allocation for energy efficient heterogeneous many-cores. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES '16, pages 27:1–27:10, New York, NY, USA, 2016. ACM.

[DOM02] Stephan Dreiseitl and Lucila Ohno-Machado. Logistic regression and artificial neural network classification models: a methodology review. *Journal of Biomedical Informatics*, 35(5):352 – 359, 2002.

[F. 14] F. Sainz and S. Mateo and V. Beltran and J.L. Bosque and X. Martorell and E. Ayguadé. Leveraging ompss to exploit hardware accelerators. In *Int. Symp. on Computer Architecture and High Performance Computing*, pages 112–119, Oct 2014.

[GBD+18] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers. Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 608–619, Feb 2018.

[GGG+16] V. García, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Pena. Evaluating the effect of last-level cache sharing on integrated gpu-cpu systems with heterogeneous applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, Sep. 2016.

[GHK+11] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[GLMR13] T. Gautier, J.V.F. Lima, N. Maillard, and B. Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Proc. of IPDPS*, pages 1299–1308, May 2013.

[GNT+19] María Angélica Dávila Guzmán, Raúl Nozal, Rubén Gran Tejero, María Villarroya-Gaudó, Darío Suárez Gracia, and Jose Luis Bosque. Cooperative cpu, gpu, and fpga heterogeneous execution with enginecl. *The Journal of Supercomputing*, page 1–15, 2019.

[Gus88] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[HCB16] Pierre Huchant, Marie-Christine Counilh, and Denis Barthou. Automatic OpenCL Task Adaptation for Heterogeneous Architectures. In *Euro-Par*, Euro-Par 2016: Parallel Processing, pages 684 – 696, Grenoble, France, August 2016.

[HCY+14] A Haidar, Chongxiao Cao, A Yarkhan, P. Luszczek, S. Tomov, K. Kabir, and J. Dongarra. Unified development for mixed multi-GPU and multi-coprocessor environments using a lightweight runtime environment. In *Proc. of IPDPS*, pages 491–500, May 2014.

[HKW14] J. Hestness, S. W. Keckler, and D. A. Wood. A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 150–160, Oct 2014.

[HKW15] J. Hestness, S. W. Keckler, and D. A. Wood. Gpu computing pipeline inefficiencies and optimization opportunities in heterogeneous cpu-gpu processors. In *2015 IEEE International Symposium on Workload Characterization*, pages 87–97, Oct 2015.

[Jun15] Junkins, Stephen. The compute architecture of intel processor graphics gen9, 2015. Last accessed April 2019.

[KBS+14] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. Adaptive heterogeneous scheduling for integrated GPUs. In *Proc. of PACT*, pages 151–162, New York, NY, USA, 2014. ACM.

[KC18] Raju K and Niranjan N. Chiplunkar. A survey on techniques for cooperative cpu-gpu computing. *Sustainable Computing: Informatics and Systems*, 19:72 – 85, 2018.

[KGCF13] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceed-*

ings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13, pages 149–160, New York, NY, USA, 2013. ACM.

[KH10a] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[KH10b] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[KKLL11] J. Kim, H. Kim, J.H. Lee, and J. Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *Proc. of the ACM PPoPP.*, pages 277–287. ACM, 2011.

[KMSZ15] David R. Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. *Heterogeneous Computing with OpenCL 2.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2015.

[KSL+12] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: An opencl framework for heterogeneous CPU/GPU clusters. In *Proceedings of the ACM ICS*, pages 341–352, New York, NY, USA, 2012. ACM.

[LHK09] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proc. of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[LLCC13] C. Lin, C. Liu, L. Chien, and S. Chang. Accelerating pattern matching using a novel parallel algorithm on gpus. *IEEE Transactions on Computers*, 62(10):1906–1916, Oct 2013.

[LSM15] J. Lee, M. Samadi, and S. Mahlke. Orchestrating multiple data-parallel kernels on multiple devices. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 355–366, Oct 2015.

[LSPM13] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. Transparent CPU-GPU Collaboration for Data-parallel Kernels on Heterogeneous Systems. In *Proc. of PACT*, pages 245–256, Piscataway, NJ, USA, 2013. IEEE Press.

[LSPM15] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. Skmd: Single kernel on multiple devices for transparent cpu-gpu collaboration. *ACM Trans. Comput. Syst.*, 33(3):9:1–9:27, August 2015.

[Mer] Mercurium C/C++/Fortran source-to-source compiler. Last accessed April 2019.

[MLC+12] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang. Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. In *2012 41st International Conference on Parallel Processing*, pages 48–57, Sept 2012.

[MMG16] M. Martineau, S. McIntosh-Smith, and W. Gaudin. Evaluating openmp 4.0's effectiveness as a heterogeneous parallel programming model. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 338–347, May 2016.

[Nan] Nanos++ Runtime Library. Last accessed April 2019.

[NVCA14] Angeles Navarro, Antonio Vilches, Francisco Corbera, and Rafael Asenjo. Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures. *J. Supercomput.*, 70(2):756–771, November 2014.

[NVI12] NVIDIA. Kepler GK110 whitepaper, 2012. Last accessed April 2019.

[NVI18] NVIDIA. NVIDIA Management Library (NVML), 2018. Last accessed April 2019.

[ope15] OpenMP Application Program Interface, Version 4.5, 2015. Last accessed April 2019.

[Ope18] Khronos Opencl. The opencl specification version: 2.2 document revision: 7, 2018.

[PBB16] Borja Pérez, José Luis Bosque, and Ramón Beivide. Simplifying programming and load balancing of data parallel applications on heterogeneous systems. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, GPGPU '16, pages 42–51, New York, NY, USA, 2016. ACM.

[PBG+13] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 457–467, New York, NY, USA, 2013. ACM.

[PBL08] J. M. Perez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *2008 IEEE International Conference on Cluster Computing*, pages 142–151, Sept 2008.

[PG14] Prasanna Pandit and R. Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM CGO*, pages 273:273–273:283, New York, NY, USA, 2014. ACM.

[Rah13] Rezaur Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers.* Apress, Berkely, CA, USA, 1st edition, 2013.

[RNR+11] Efraim Rotem, Alon Naveh, Doron Rajwan, Avinash Ananthakrishnan, and Eli Weissmann. Power management architecture of the 2nd generation Intel Core microarchitecture, formerly codenamed Sandy Bridge. In *IEEE Int. HotChips Symp. on High-Perf. Chips (HotChips˜2011)*, 2011.

[SGS10] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[SMV10] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. Maestro: Data orchestration and tuning for opencl devices. In *Proceedings of the 16th International*

Euro-Par Conference on Parallel Processing: Part II, Euro-Par'10, pages 275–286, Berlin, Heidelberg, 2010. Springer-Verlag.

[TV14] Vasanth Tovinkere and Michael Voss. Flow graph designer: A tool for designing and analyzing intel&; threading building blocks flow graphs. In *2014 43rd International Conference on Parallel Processing Workshops*. IEEE, sep 2014.

[VAN+15] Antonio Vilches, Rafael Asenjo, Angeles Navarro, Francisco Corbera, Rubén Gran, and María Garzarán. Adaptive partitioning for irregular applications on heterogeneous cpu-gpu chips. *Procedia Computer Science*, 51:140 – 149, 2015. International Conference On Computational Science, ICCS 2015.

[VEL+17] T. Vijayaraghavan, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan. Design and analysis of an apu for exascale computing. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 85–96, Feb 2017.

[WR10] G. Wang and X. Ren. Power-efficient work distribution method for cpu-gpu heterogeneous system. In *International Symposium on Parallel and Distributed Processing with Applications*, pages 122–129, Sept 2010.

[YWTC15] Yi-Ping You, Hen-Jung Wu, Yeh-Ning Tsai, and Yen-Ting Chao. Virtcl: A framework for opencl device abstraction and management. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 161–172, New York, NY, USA, 2015. ACM.

[YXMZ12] Y. Yang, P. Xiang, M. Mantor, and H. Zhou. Cpu-assisted gpgpu on fused cpu-gpu architectures. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, Feb 2012.

[ZAM+15] Amir Kavyan Ziabari, José L. Abellán, Yenai Ma, Ajay Joshi, and David Kaeli. Asymmetric noc architectures for gpu systems. In *Proceedings of the 9th In-*

ternational Symposium on Networks-on-Chip, NOCS '15, pages 25:1–25:8, New York, NY, USA, 2015. ACM.

[ZH13] Jianlong Zhong and Bingsheng He. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *CoRR*, abs/1303.5164, 2013.

[ZRL15] Ziming Zhong, V. Rychkov, and A. Lastovetsky. Data partitioning on multicore and multi-gpu platforms using functional performance models. *Computers, IEEE Transactions on*, 64(9):2506–2518, Sept 2015.