Revista Iberoamericana de Automática e Informática Industrial

Revista Iberoamericana de Automática e Informática Industrial 21 (2024) 1–16

www.revista-riai.org

Uso de GPUs en aplicaciones de tiempo real: Una revisión de técnicas para el análisis y optimización de parámetros temporales

Iosu Gomez[a,b,*], Unai Díaz de Cerio[a], Jorge Parra[a], Juan M. Rivas[b], J. Javier Gutiérrez[b]

[a]*Área de Sistemas Embebidos Confiables, Ikerlan Centro de Investigación Tecnológico, Basque Research and Technology Alliance (BRTA), Arrasate-Mondragón, España.*

[b]*Grupo de Ingeniería Software y Tiempo Real, Universidad de Cantabria, Santander, España.*

To cite this article: Gomez, I., Díaz de Cerio, U., Parra, J., Rivas, J. M., Gutiérrez, J. J. 2024. Using GPUs in Real-Time Applications - A Review of Techniques for Analyzing and Optimizing the Timing Parameters. Revista Iberoamericana de Automática e Informática Industrial 21, 1-16. https://doi.org/10.4995/riai.2023.20321

Resumen

La conducción autónoma despierta un interés cada vez mayor en la industria, no solo en el sector de la automoción, sino también en el transporte de personas o mercancías por carretera o ferrocarril y en entornos de fabricación más controlados. Los sistemas ciber-físicos que se están proponiendo para este tipo de aplicaciones requieren de una gran capacidad de cómputo (arquitecturas hardware con varios núcleos, GPUs, NPUs…) para poder atender y reaccionar a una múltiple y compleja cantidad de sensores (cámaras, radar, LiDAR, medida de distancia, etc.). Por otro lado, este tipo de sistemas debe atender a requisitos de seguridad funcional y también de tiempo real. Este último aspecto plantea retos en los que se está trabajando intensamente y en los que aún quedan muchas cuestiones por resolver. En este trabajo, se hace una revisión de la literatura más reciente del uso de arquitecturas heterogéneas con GPUs en aplicaciones de tiempo real. Estos trabajos proponen soluciones para la estimación de cotas de tiempos de ejecución y respuesta temporal, proponiendo diferentes estrategias de optimización destacando la mitigación de interferencia en la memoria.

Palabras clave: Sistemas embebidos, Sistemas de tiempo real, Planificación, Procesadores heterogéneos, GPUs.

Using GPUs in Real-Time Applications - A Review of Techniques for Analyzing and Optimizing the Timing Parameters

Abstract

Autonomous driving is attracting an increasing attention in industry, not only in the automotive sector, but also in the transport of people or goods by road or railway and in more controlled manufacturing environments. The cyber-physical systems that are being proposed for this type of applications require a large computing capacity (hardware architectures with several cores, GPUs, NPUs...) to be able to attend and react to a multiple and complex amount of sensors (cameras, radar, LiDAR, measure of distance, etc.). On the other hand, this type of system must meet both safety and real-time requirements. This last aspect poses challenges on which intensive work is being done and on which there are still many open issues. In this work, a review of the most recent literature on the use of heterogeneous architectures with GPUs in real-time applications is made. These works mainly propose some solutions to the estimation of bounds to the execution times and response times, and consider different optimization strategies emphasising memory interference mitigation.

Keywords: Embedded systems, Real-time systems, Scheduling, Heterogeneous processors, GPUs.

*Autor para correspondencia: iosu.gomez@ikerlan.es

1. Introducción

Los sistemas ciber-físicos modernos deben soportar aplicaciones que cada vez requieren una mayor carga computacional, por lo que se apoyan en arquitecturas heterogéneas con varios núcleos procesadores y aceleradores hardware específicos como GPUs (*Graphics Processing Units*) o NPUs (*Neural Processing Units*). Así por ejemplo, en aplicaciones para movilidad inteligente se utilizan algoritmos de computo intensivo (como algoritmos de Inteligencia Artificial) para procesar datos complejos que provienen de los sensores de percepción avanzada (Yurtsever et al., 2020). Por otra parte, este tipo de aplicaciones debe satisfacer ciertos requisitos de seguridad funcional y también de tiempo real, por lo que no solo es fundamental que los resultados del cómputo sean correctos, sino que estos se produzcan dentro de los plazos temporales impuestos. Una muestra del gran interés que suscita la investigación del uso de GPUs en este tipo de aplicaciones de tiempo real, la encontramos en los diferentes retos industriales que se han ido proponiendo en congresos internacionales del área (Hamann et al., 2019; Boniol and Mohan, 2022; Andreozzi et al., 2022).

En respuesta a esta necesidad, los fabricantes han propuesto diferentes tipos de plataformas hardware heterogéneas que incluyen estos aceleradores hardware. De entre estos aceleradores, los que más se han popularizado en el campo de la investigación de la movilidad inteligente son las unidades de procesamiento gráfico (GPUs). Las GPUs se integran en estas plataformas heterogéneas como co-procesadores junto con una o más CPUs, y no pueden funcionar por tanto como elementos independientes. Así, dependiendo de la integración en estas plataformas, se pueden encontrar dos categorías principales: (1) arquitecturas discretas, en las que la GPU se implementa en una tarjeta que se inserta en una ranura de expansión en la placa base del sistema (conectada mediante bus PCIe por ejemplo), contando con su propia memoria dedicada, y (2) arquitecturas integradas, en las que la GPU está implementada junto a las CPUs en un mismo MPSoC (*Multiprocessor System on Chip*), y se comunican con las CPUs mediante la memoria compartida del sistema. Este trabajo está centrado principalmente en las GPUs integradas, ya que son las mayormente utilizadas en sistemas ciber-físicos y aplicaciones como las mencionadas, debido a su menor coste y consumo energético. Las arquitecturas discretas, en cambio, se usan más en sistemas que requieren mayor potencia de cómputo, como videojuegos 3D, minería de criptomonedas, o *machine-learning*.

Las ventajas obtenidas en el uso conjunto de CPUs y GPUs (u otros componentes de procesamiento) para aplicaciones de cómputo intensivo, se pueden ver mermadas por cierto deterioro en el rendimiento producido por el aumento de la interferencia en la memoria. Este hecho es especialmente relevante en sistemas críticos que deben garantizar requisitos temporales, ya que esa interferencia puede generar grandes variaciones en los tiempos de ejecución que pueden comprometer la predictibilidad. Los principales estudios dedicados a identificar, caracterizar y proponer soluciones para mitigar este problema de interferencia en la memoria serán objeto de análisis en este trabajo.

Así pues, este trabajo presenta una revisión de la literatura teniendo en cuenta los estudios más recientes y significativos sobre el uso de arquitecturas heterogéneas (principalmente GPUs integradas) en aplicaciones de tiempo real. La revisión realizada se ha abordado desde dos puntos de vista que se complementan. En primer lugar, se ha realizado una revisión de aquellos trabajos enfocados a analizar detalladamente el funcionamiento interno de las GPUs integradas y el uso que hacen de la memoria global, incluyendo la problemática mencionada de la interferencia de memoria. Con esto se pretende brindar al lector una visión detallada de las arquitecturas con GPU integrada. En segundo lugar, se recogen trabajos que muestran diferentes técnicas de optimización que se basan en mitigar el efecto de la interferencia y mejorar la predictibilidad de las aplicaciones que ejecutan en la GPU. Una revisión exhaustiva de las técnicas recientes que compone el bloque central de este trabajo. Adicionalmente, en este trabajo se revisan las técnicas más recientes de modelado y análisis de planificabilidad, destacando así los trabajos centrados en la problemática del modelado de aplicaciones basadas GPUs como una rama diferenciada dentro de este campo. Como resultado se presenta una clasificación de estos trabajos, atendiendo a los problemas que abordan y cómo los resuelven, que puede ser de utilidad a los investigadores interesados en esta tecnología.

En el trabajo (Perez-Cerrolaza et al., 2022), se realiza una revisión exhaustiva y general de aspectos de seguridad funcional y tiempo real en GPUs. El trabajo que se presenta aquí se puede ver como una ampliación de la parte de esa revisión centrada únicamente en las técnicas de optimización de parámetros temporales con el objetivo de cumplir plazos en aplicaciones de tiempo real.

El presente documento está organizado de la siguiente manera. En el apartado 2 se realiza una descripción de las plataformas heterogéneas con GPU integrada, detallando su arquitectura, modelo de ejecución y aspectos relevantes del acceso a memoria. En el apartado 3, se revisan las principales técnicas de optimización en el uso de GPUs cuyo objetivo principal es la reducción de las interferencias de memoria. El apartado 4 se dedica a la revisión de trabajos sobre el modelado y análisis de planificabilidad de sistemas de tiempo real en plataformas heterogéneas con GPU. Finalmente, en el apartado 5 se muestran las conclusiones y se incluyen dos tablas que resumen todos los trabajos recogidos en este artículo, así como futuras líneas de trabajo.

2. Plataformas heterogéneas con GPU integrada

En este apartado se explica detalladamente la arquitectura y funcionamiento interno de la GPU con la intención de dar contexto al lector. Así, se muestra información acerca del modelo de ejecución y mecanismos de arbitraje interno, además de mostrar los diferentes modelos de trasferencia de información entre CPU y GPU. Se parte de la base del funcionamiento básico de las plataformas heterogéneas en general: descarga de datos por parte del dispositivo acelerador, procesamiento en este dispositivo y carga de datos de vuelta a la CPU.

Es importante tener en cuenta que, en el caso de las GPUs, cierta información tanto de la arquitectura como de los mecanismos internos o de los drivers no suele ser de carácter público. Este hecho ha motivado diversos trabajos de investigación de ingeniería inversa con el objetivo de arrojar más luz sobre el comportamiento de los mecanismos internos de la GPU.

La arquitectura y los fundamentos del funcionamiento de las GPUs mostrados en este apartado y en el resto del trabajo, se han descrito a partir de la terminología de CUDA (Nvidia, 2023), la cual se aplica a GPUs del fabricante NVIDIA. CUDA es el modelo de programación diseñado por NVIDIA para sus arquitecturas con GPUs. Para otros fabricantes de GPU, se utiliza una terminología similar basada en el modelo de programación OpenCL (Khronos, 2023). En la Tabla 1, se puede ver una comparativa de algunos de los términos utilizados por uno u otro modelo.

Tabla 1: Equivalencia entre la terminología CUDA y OpenCL.

CUDA	OpenCL
CUDA Core	Processing Elements
Warp	Wavefront
Streaming Multiprocessor	Compute Unit
(Thread) Block	Work-group
Thread	Work-item
Kernel	Kernel
Stream	Command Queue

2.1. Arquitectura

Conforme a la terminología CUDA, las GPUs se componen de unidades de procesamiento paralelo llamadas *Stream Multiprocessors* (SM). Cada una de estas unidades se componen de múltiples núcleos o *CUDA cores*, además de diferentes elementos y recursos necesarios para el procesamiento paralelo como registros, niveles de caché o *Tensor cores*. Los *CUDA cores* procesan agrupaciones de 32 hilos (*threads*) conocidos como *Warps* de manera *Single Instruction Multiple Data* (SIMD). La ejecución SIMD se basa en que todos los núcleos CUDA ejecutan la misma instrucción de código simultáneamente sobre diferentes datos garantizando el paralelismo entre datos. Las diferentes agrupaciones de núcleos CUDA que pueden estar ejecutándose en un SM son planificadas mediante uno o más *Warp Schedulers*.

Con el objetivo de reducir las latencias provocadas por el acceso a la memoria global, las GPUs utilizan varios niveles de caché (dependiendo del modelo hardware utilizado) diseñados para permitir un acceso más rápido y eficiente a los datos.

2.2. Modelo de ejecución de la GPU

Para poder ejecutar tareas en la GPU, la CPU tiene que cargar parte del código a ejecutar en el acelerador. A este trozo de código se le denomina *Kernel*. Cuando una tarea en la CPU requiere uso de la GPU, en primer lugar, el driver parte el *Kernel* en diferentes bloques de hilos o *Thread Blocks*. Estos bloques son una colección de hilos determinados por el programador que ejecutan las mismas instrucciones, pero que operan sobre diferentes porciones de datos. La transmisión del *Kernel* a la GPU se realiza mediante la memoria compartida. El estudio (Olmedo et al., 2020) muestra una perspectiva general de las fases por las que transcurre un *Kernel* desde que es enviado a la GPU hasta que completa su ejecución. A continuación, en este apartado se describen detalladamente dichas fases.

Cuando se lanza un *Kernel* en la GPU, la operación se puede realizar de manera síncrona o asíncrona. En el modo síncrono, el hilo de CPU se queda esperando a la respuesta de la GPU. En el modo asíncrono, en cambio, el hilo en la CPU sigue realizando operaciones mientras se ejecuta el *Kernel* en la GPU. Después, existen diferentes mecanismos para volver a sincronizar los datos entre la CPU y la GPU.

Una vez en la GPU, los bloques se distribuyen en colas FIFO de operaciones GPU llamadas *Streams*. Las operaciones GPU se clasifican en dos tipos. En primer lugar, se encuentran las operaciones de copia de datos, que son procesadas por un motor de copia llamado *Copy Engine* (CE) de la GPU. En segundo lugar, se encuentran los bloques de *Kernel* a ejecutar en los SM. Al motor encargado de ejecutar los bloques se le denomina *Execution Engine* (EE). El CE y el EE son mecanismos independientes, por lo que pueden funcionar concurrentemente.

Si el programador solo asigna un *Stream*, la ejecución se realiza de manera secuencial: copia de datos de la CPU a la GPU, ejecución del *Kernel*, y copia de la GPU a la CPU. Sin embargo, es posible asignar bloques a diferentes colas de ejecución para poder realizar operaciones GPU concurrentemente. Esta distribución queda en manos del programador, pero si éste no lo especifica, los bloques son asignados a un mismo *Stream*. En el caso de utilizar múltiples *Streams*, se pueden ejecutar concurrentemente tanto los bloques de hilos como las operaciones de copia, ya que el CE y el EE son dos mecanismos independientes.

Cuando se asignan diferentes colas, es posible asignarles una prioridad alta o una prioridad baja. Como se muestra en (Singh et al., 2022), existe la expulsión entre *Kernels* asignados a diferentes *Streams*, pero esta expulsión no se hace de manera directa, solo se puede realizar cuando haya finalizado la ejecución de un bloque de hilos u operación de copia.

Para asignar los bloques de ejecución a los SM y los bloques de copia al CE, existe un mecanismo de arbitraje llamado *Thread Block Scheduler*. En (Amert et al., 2018), se muestra cómo se asignan los bloques de las colas *Stream* a las unidades de procesamiento. En este estudio, se utiliza la ingeniería inversa para modelar el funcionamiento observado de la asignación de bloques del *Kernel* a los SMs. Asumen dos colas FIFO adicionales a los *Stream*: una cola de operaciones del EE y otra del CE. La primera, se encarga de encolar los bloques de *Kernel* para ser asignados a los SM. De la misma manera, la cola CE contiene las operaciones de copia. Asimismo, definen las reglas de arbitraje que cumplen los bloques de copia y *Kernel* para moverse entre colas.

Cuando uno o más bloques llegan al SM se descomponen en los previamente mencionados fragmentos de 32 hilos llamados *Warp*. Es posible ejecutar múltiples *Warps* concurrentemente sobre una misma unidad de procesamiento o core dependiendo de la cantidad de *Warp Scheduler* que contenga la unidad de procesamiento. Esta cantidad puede variar dependiendo de la arquitectura utilizada. Cada planificador se encarga de seleccionar un *Warp* que tenga una instrucción lista para ejecutar y darle el acceso a los núcleos CUDA.

Respecto a las políticas de planificación dentro de los SM, en (Singh et al., 2022) mencionan dos políticas de planificación para los *Warp Scheduler*: *greedy-then-oldest* (GTO) y *Loose Round-Robin* (LRR). Mientras que en (Olmedo et al., 2020), mencionan que para las arquitecturas utilizadas en GPUs integradas la política utilizada puede asemejarse más a LRR. Con esta política, los *Warps* se ejecutan de forma rotatoria hasta que uno alcanza una dependencia insatisfecha (una función que espera datos de la memoria, un punto de sincronización, etc.).

Entonces, la ejecución del *Warp* se detiene, para que pueda programarse el siguiente *Warp* sin dependencias insatisfechas de la cola.

2.3. Comunicación CPU-GPU por memoria compartida

Como se ha mencionado anteriormente, en arquitecturas de GPU integradas, la CPU comparte con el acelerador hardware la memoria global del sistema como medio de comunicación entre ambos. La principal finalidad de esta comunicación es suministrar a la GPU los datos que requieren ser procesados en paralelo y, posteriormente, recopilar la respuesta generada por el acelerador.

Existen diferentes modelos de comunicación CPU-GPU. Siguiendo con la metodología CUDA, los traspasos de datos entre la CPU y la GPU por la memoria compartida se puede realizar mediante las siguientes configuraciones:

- *Standard Copy* (SC): La técnica de copia estándar es un método comúnmente utilizado para copiar datos de la CPU a la GPU. En el espacio físico de la memoria global se crean dos espacios lógicos, uno para la CPU y otro para la GPU. De esta manera el mecanismo de CE de la GPU puede transferir datos de un espacio a otro mediante *Direct Memory Access* (DMA).

- *Zero Copy* (ZC): Esta técnica conocida como cero-copia utiliza un espacio común de la memoria compartida al que tanto la CPU como la GPU pueden acceder directamente. Los datos se pasan mediante punteros a la GPU que apuntan a un espacio de direcciones compartido en la memoria global. De esta forma, la GPU puede acceder a los datos sin la necesidad de realizar ninguna transferencia o copia de memoria adicional. Al acceder directamente a la memoria, la caché de la GPU permanece inactiva.

- *Zero Copy* + Coherencia Hardware: La combinación de la técnica de cero-copia con la coherencia de hardware se utiliza para optimizar el rendimiento de sistemas que requieren el procesamiento intensivo de datos. La coherencia de hardware se logra mediante la inclusión de un hardware de coherencia I/O que accede directamente al último nivel de caché de la CPU mientras la caché de la GPU sigue inactiva. A diferencia de las demás configuraciones, ésta depende de la arquitectura utilizada ya que es necesario que disponga del mecanismo hardware.

- *Unified Memory* (UM): La memoria unificada es un modelo de comunicación que permite a la CPU y la GPU acceder a la misma memoria física compartida, aunque se mantengan los espacios lógicos separados en la programación. Facilita la transferencia de datos entre la CPU y la GPU, ya que no es necesario realizar copias de memoria adicionales para compartir datos entre los dos espacios lógicos. El programador no tiene control ni conocimiento de lo que ocurre por debajo y queda completamente bajo el control del driver utilizado.

De las técnicas mencionadas, SC destaca por su simplicidad y facilidad de uso, aunque puede presentar problemas de rendimiento debido a la latencia involucrada en la transferencia de datos entre la CPU y la GPU. Según el estudio (Lumpp et al., 2021), SC es más eficaz (en términos de tiempo) que ZC e incluso que la copia mediante memoria unificada (Otterness et al., 2017). En modo de cero-copia, tanto la CPU como la GPU tienen que acceder a los datos mediante punteros a la memoria, anulando el efecto de las cachés. Esto significa que aunque se hayan reducido las transferencias de memoria, las tareas que requieren datos se encuentran obligadas a acceder a la memoria global en lugar de a la caché, lo que puede dar lugar a tiempos de acceso más altos. Sin embargo, esta afirmación es solo válida para aplicaciones que permitan cargar todos los datos mediante SC en la caché de la GPU y no se vean obligadas a acceder a la memoria global.

En arquitecturas que implementan la coherencia hardware, el método ZC es optimizado (Calderón et al., 2022) debido a que la GPU puede acceder al último nivel de caché de la CPU en vez de tener que acceder a la memoria global. El estudio comparativo realizado en (Marchi et al., 2021) muestra que la implementación de la coherencia en la técnica ZC, además de mejorar significativamente el acceso a memoria por ZC, también puede ser beneficiosa en comparación con el método de copia estándar. Los tiempos de ejecución mostrados en el caso de estudio expuesto en al artículo muestran una comparativa entre tareas GPU con los modelos de copia SC y ZC en dos plataformas diferentes, una con coherencia hardware y otra sin. En el primer experimento, se reduce el uso de la caché haciendo que la tarea de GPU tenga que acceder más a la memoria global pese a utilizar SC. En la plataforma sin coherencia hardware se puede observar un incremento en la ejecución de la tarea de 1053 ms utilizando el método SC, que llega hasta 1316.1 ms con ZC. En cambio, en la plataforma con coherencia hardware y gracias al mecanismo de coherencia, el método ZC realiza menos accesos a la memoria global reduciendo los tiempos de ejecución. En los resultados se puede observar que de esta manera, el método ZC es más rápido que el SC: 256.9 ms frente a 381.2 ms respectivamente. Para el experimento en el que se realiza un mayor uso de la caché, el ZC con coherencia hardware pierde la ventaja, siendo incluso más lento que SC, 244.7 ms en ZC frente a 207.7 ms en SC.

Por último, la memoria unificada se presenta cómo una opción interesante en muchos escenarios, ya que permite reducir significativamente la necesidad de transferencias de memoria, y mejora el rendimiento del sistema al minimizar el tiempo de latencia en comparación con SC. Según (Fickenscher et al., 2017), en su comparativa entre ZC y UM, se observan accesos en general más rápidos en la memoria unificada, aunque pierde predictibilidad en los tiempos de ejecución. Además, destaca que esto solo es válido para un tamaño limitado de memoria, a partir de una cierta cantidad empieza a ser más lenta.

Independientemente del modelo de comunicación utilizado entre la CPU y la GPU, existen ciertos aspectos del driver que pueden ser desconocidos para el programador. De acuerdo con (Calderón et al., 2019), el método de asignación de memoria de CUDA, denominado *CUDA Alloc*, asigna ciertos recursos extra sobre los requeridos. Estos recursos adicionales son independientes del método de copia y plataforma utilizada, siendo éstos similares en los diferentes casos. El estudio obtiene la información de la asignación de memoria CUDA mediante algoritmos de ingeniería inversa, y muestra cómo aplicar estos algoritmos

en un caso de uso dentro de la movilidad inteligente. El objetivo es proporcionar al desarrollador las pautas a seguir para implementar los algoritmos de ingeniería inversa en su sistema y así poder obtener la cantidad de memoria real asignada.

2.4. *Interferencia CPU-GPU*

Principalmente, la interferencia entre la CPU y la GPU en la computación heterogénea se debe a la contención en los diferentes niveles de memoria. Los estudios mostrados en (Cavicchioli et al., 2017, 2020; Capodieci et al., 2022) identifican los puntos de contención en los diferentes niveles de memoria para múltiples plataformas de GPU integradas y caracterizan dicha interferencia. Otros trabajos como (Houdek et al., 2017; Bechtel and Yun, 2023), también muestran cómo la interferencia mediante el acceso intensivo a memoria por parte de la CPU y la GPU puede degradar la respuesta temporal de tareas ejecutadas en ambas unidades de procesamiento individualmente. Además, existen otros factores que pueden causar bloqueos a nivel de software (Yang et al., 2018). En este apartado, se muestran posibles causas de interferencia y bloqueos en la computación de MPSoC basados en CPU-GPU integrada.

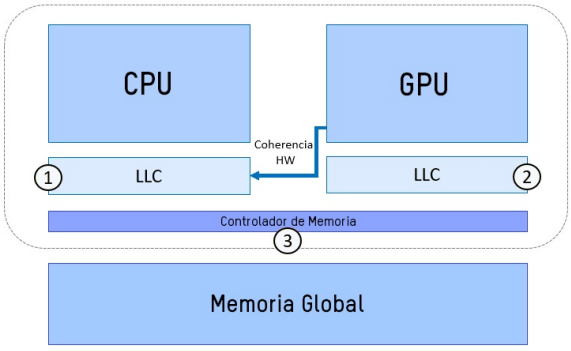

Figura 1: Principales puntos de contención en la memoria de plataformas integradas CPU-GPU.

El primer punto de contención se encuentra en el último nivel de caché (LLC, *Last Level Cache*) de la CPU, tal y como se muestra en la Figura 1. Esta contención se da, en primer lugar, entre los diferentes núcleos de la CPU. La memoria caché está ubicada entre la memoria global y las unidades de procesamiento y se utiliza para agilizar el rendimiento del sistema. Al ejecutar múltiples núcleos de la CPU concurrentemente es más probable que alguno sobrescriba datos en líneas de caché utilizadas por otros núcleos. Cuando esto ocurre, la CPU no encuentra el dato en la caché y tiene que acceder a la memoria principal para buscarlo. Este efecto aumenta el tiempo de ejecución y disminuye la predictibilidad del sistema (Dasari et al., 2013). Otro motivo de contención en el LLC de la CPU es debido al mecanismo de coherencia hardware entre la CPU y la GPU que implementan algunas arquitecturas (Cavicchioli et al., 2017). Como gracias a este mecanismo la GPU también es capaz de acceder al LLC de la CPU, aumenta la probabilidad de fallo de caché al intentar acceder a una línea de datos.

El segundo punto de contención mostrado en la Figura 1 se encuentra en el LLC de la GPU. Al igual que ocurre en el lado de la CPU, múltiples SM y *CUDA cores* pueden estar accediendo a la caché compartida provocando fallos de caché en el acceso a los datos (Olmedo et al., 2018; Yandrofski et al., 2022). Otros trabajos (Otterness et al., 2016, 2017) han evaluado la interferencia entre *Kernels* concurrentes y su impacto en el tiempo de ejecución de peor caso (WCET) de los propios *Kernels*. Estos trabajos indican que paradójicamente, a pesar de la interferencia mutua de diferentes bloques de ejecución, hay casos en los que se obtiene una mejor respuesta temporal al aumentar la utilización de la GPU. En cualquier situación, es imprescindible analizar exhaustivamente cada caso y obtener un análisis detallado de los tiempos de ejecución e impacto de las interferencias provocadas por el acceso concurrente a memoria. En contraposición, en términos de predictibilidad es altamente recomendable proporcionar un aislamiento temporal entre *Kernels* (Olmedo et al., 2018) evitando la interferencia de memoria.

El tercer punto de contención de la Figura 1 se encuentra en el acceso a la memoria global (Cavicchioli et al., 2017). Cuando una tarea (sea de CPU o de GPU) necesita acceder a la memoria global, realiza una petición al controlador de memoria (MC, *Memory Controller*). El MC gestiona el acceso a los bancos de memoria. Si éste recibe múltiples peticiones al mismo tiempo, es posible saturar el ancho de banda en el acceso a la memoria produciendo latencias en las diferentes tareas que intentan acceder a ella. De esta manera, la ejecución de tareas en la CPU puede interferir en la ejecución de la GPU incrementando tanto su WCET como la variabilidad de éste y viceversa.

Como se ha mencionado anteriormente, la contención a diferentes niveles de memoria no es la única causa de interferencia o bloqueo en arquitecturas CPU-GPU. El estudio (Yang et al., 2018) muestra cómo la sincronización entre la CPU y la GPU puede dar lugar a bloqueos a nivel del driver en la ejecución de las tareas. Dependiendo del origen, los bloqueos por causa de la sincronización se dividen en dos grupos. Por un lado, los bloqueos en la sincronización explícita son aquellos debidos a puntos de sincronización puestos por el programador. Estos puntos de sincronización pueden estar tanto en la CPU como en la GPU. Por otro lado, los bloqueos por sincronización implícita ocurren como efectos colaterales de la API de CUDA. Cuando un hilo de la CPU realiza una llamada a la API de CUDA, puede suceder que la ejecución del *Kernel* en la GPU se bloquee hasta que se complete la tarea solicitada por la llamada. Por lo tanto, es importante tener en cuenta la sincronización entre la CPU y la GPU al diseñar aplicaciones para evitar estos bloqueos y mejorar el rendimiento.

3. Técnicas de optimización en el uso de GPUs

En este apartado se abordan una serie de técnicas diseñadas para mitigar el efecto de las interferencias de memoria, optimizar la respuesta temporal y aumentar la predictibilidad en arquitecturas de cómputo heterogéneas. Algunas de estás técnicas se basan en propuestas existentes para plataformas multiprocesador (Lugo et al., 2022) que han sido adaptadas para abarcar plataformas heterogéneas. Sin embargo, otras técnicas presentadas son exclusivas para optimizar el rendimiento de la GPU.

3.1. *Coloración de página*

La coloración de página o *page coloring*, es una técnica utilizada para mitigar efectos negativos como la interferencia en la memoria en entornos multiprocesador. Se basa en particionar los bancos de memoria y líneas de caché con el propósito de

distribuir las páginas definidas en diferentes procesadores y así aislar regiones separadas de la caché o de la memoria compartida. De esta manera, pese a que se mantenga la interferencia en el bus de acceso, se permite minimizar los conflictos en los bancos de memoria y caché mejorando la respuesta temporal del sistema, además de aprovecharse de cierto paralelismo. La coloración de página es utilizada en sistemas multiprocesador por múltiples trabajos, (Zhang et al., 2009; Liu et al., 2012; Suzuki et al., 2013; Ye et al., 2014; Yun et al., 2014; Mancuso et al., 2015; Kim and Rajkumar, 2016; Kloda et al., 2019; Lim and Kim, 2019; Park et al., 2020) entre otros, como una solución efectiva para mitigar la interferencia de memoria.

Centrándonos en el contexto de la mitigación en plataformas con GPU, se ha desarrollado una técnica denominada FractionalGPU (Jain et al., 2019) donde se llevan las ideas de la coloración de página para multiprocesador a la GPU. Uno de los objetivos de FractionalGPU es aumentar la predictibilidad de tareas que se ejecutan concurrentemente en la GPU. La idea principal consiste en particionar la GPU (ver apartado 3.6), donde cada partición contiene un conjunto de SMs específicos. A cada partición en la GPU se le asigna una página física específica. De esta manera, se limita la interferencia en la memoria de diferentes particiones al no poder acceder a las mismas líneas de memoria. La mayor limitación de éste método es que para permitir el acceso concurrente de cada partición de GPU a su página de memoria asignada, es necesario que exista un sistema de interconexión que contenga diversos controladores de memoria. Hasta el momento, las únicas arquitecturas que satisfacen este requisito son las arquitecturas de GPU discretas. Por lo tanto, la técnica se centra en reducir la interferencia en los niveles de memoria de la GPU. De transferir esta técnica a arquitecturas integradas, podría servir también para reducir las interferencias en la memoria global del sistema.

3.2. Modelo de ejecución predecible

El modelo de ejecución predecible (Pellizzoni et al., 2011) PREM (*Predictable Execution Model*) es una técnica utilizada en plataformas multiprocesador para mitigar los efectos de la interferencia debido al acceso simultáneo de tareas a la memoria global. El modelo de ejecución PREM se basa en la idea de que la memoria del sistema es un recurso crítico que se concede y se libera en intervalos de tiempo regulares conocidos como fases. Cada tarea se compone de una fase de memoria al inicio para cargar los datos necesarios y una fase de ejecución donde no se realiza ningún acceso a memoria. Las fases de memoria de diferentes tareas no pueden coincidir en el tiempo para evitar crear interferencias. Originalmente (Pellizzoni et al., 2011), las tareas no pueden ser expulsadas por otras de mayor prioridad para garantizar que el contenido de la caché no pueda ser alterado. Además, para sistemas multiprocesador, hay que sincronizar las diferentes tareas para que no accedan a la memoria al mismo tiempo. En otras implementaciones (Schuh et al., 2020), se aíslan las fases de memoria en una misma CPU para evitar cualquier concurrencia entre las fases de memoria.

Con el uso de plataformas de cómputo heterogéneas, el problema de interferencia ya no reside solo en la interferencia entre diferentes núcleos de una CPU. Como se ha mostrado en apartados anteriores, el uso de aceleradores hardware como la GPU ha aumentado esta problemática. Al añadir más unidades de procesamiento, aumentan los dispositivos que acceden a la memoria simultáneamente, aumentando las fuentes de interferencias. Para solucionar este problema algunos estudios (Houdek et al., 2017; Forsberg et al., 2017b, 2019) han expandido el modelo de ejecución predecible para que abarque la integración de GPUs. Además de dividir las fases de memoria y ejecución de las CPUs, se asignan también fases de memoria y de ejecución para aquellos hilos que requieren el uso de la GPU. Todas las fases de memoria, tanto de CPU como de GPU, se sincronizan de manera que no se superpongan en el tiempo. De esta manera, se mitiga completamente la interferencia que la GPU pueda crear a otras tareas críticas de la CPU y viceversa. En los estudios mencionados se puede observar cómo al ejecutarse de manera PREM y evitar la contención de memoria, la medición del WCET de las tareas bajo estudio mejora significativamente. Adicionalmente, en (Forsberg et al., 2021) se formaliza la extensión del modelo PREM para plataformas heterogéneas, denominado HePREM.

Una solución basada en el modelo expuesto es GPUGuard (Forsberg et al., 2017a). Se ejecuta a nivel del *Kernel* de Linux y controla el acceso a la memoria global garantizando que la GPU no realiza accesos fuera de su fase de memoria. De esta manera, la CPU puede acceder a la memoria sin interferencia de la GPU en sus respectivas fases. Las fases se asignan mediante temporizadores y la sincronización CPU-GPU se lleva a cabo mediante *flags* en la memoria compartida. Para poder asignar el tiempo en estas fases es necesario conocer los WCETs de la GPU, que deben incluir tanto la propia ejecución como el tiempo de transferencia de memoria.

Otra solución para mitigar la interferencia en sistemas basados en arquitecturas CPU-GPU es SiGAMMA (Capodieci et al., 2017). Presenta una solución que, a nivel del software, arbitra los accesos a memoria tanto por parte de la CPU como de la GPU. A diferencia de la solución anterior, considera que la ejecución de un hilo en la GPU no es compatible con el modelo de ejecución en PREM debido a que además del CE, el EE también puede acceder a la memoria si un núcleo CUDA no encuentra un dato en la caché y tiene que acceder a la memoria global, creando interferencias no previstas. Por lo tanto, se considera a la tarea completa en la GPU como si fuese una sola fase de memoria. De esta manera, dentro del modelo PREM que se esta ejecutando en la CPU, la actividad de la GPU se ejecuta dentro de la fase de computación de la CPU. En caso de tener una plataforma multiprocesador, la fase de actividad en la GPU se reduce al espacio de tiempo donde todos los núcleos coinciden en su fase de computación. Otro punto interesante de esta solución es la técnica de expulsión del *Kernel*. Al priorizar los tiempos de cumplimiento de las tareas críticas de la CPU, se asignan las ventanas temporales acorde a estas tareas y no basándose en los WCET de la GPU. Debido a esto, puede ocurrir el caso en el que finalice el tiempo de la fase de actividad de la GPU pero el *Kernel* no haya finalizado y siga ejecutándose en el dispositivo. Como no existen mecanismos internos para la expulsión de *Kernels* en la GPU, en SiGAMMA se incorpora una técnica de expulsión del *Kernel* basándose en el uso de los *Streams* de prioridad alta y baja. Se trata de enviar un *Kernel* vacío y de una duración determinada, que no realice accesos a memoria, al *Stream* de prioridad alta de la GPU cuando finaliza su fase. Así se evita que la tarea de GPU, de baja prioridad,

interfiera en la memoria fuera de su fase.

En (Spliet and Mullins, 2022), se muestra otro enfoque del modelo PREM para arquitecturas GPU integradas en un entorno llamado Sim-D. A diferencia de los trabajos anteriores, en los que se utiliza el modelo de ejecución para sincronizar las fases de la CPU con las fases de la GPU, este trabajo se centra en aplicar el modelo PREM entre *Kernels* ejecutándose en la GPU. De esta manera, se mitiga la interferencia entre bloques de hilos ejecutándose en la GPU. Cada bloque en la GPU tiene su fase de computación y su fase de memoria. El entorno presentado planifica pares de bloques de hilos para sincronizar la fase de computación de un bloque con la fase de memoria del otro. El trabajo incluye un análisis de tiempos de respuesta para la solución propuesta. Aplicando la solución propuesta, pese a perder un 9.2 % de rendimiento en la GPU, mejora la predictibilidad al conseguir ajustar los tiempos de respuesta estimados dentro de un rango del 12.7 % respecto del real.

3.3. Reserva de ancho de banda de memoria

Otra técnica de mitigación de interferencia por el acceso a memoria que se ha importado de plataformas multiprocesador a plataformas heterogéneas es la reserva del ancho de banda en el acceso a la memoria. Este mecanismo consiste en asignar una porción del ancho del bus de memoria a una determinada tarea, aplicación o CPU. Con esto se garantiza que una aplicación no sobrecargue el bus de memoria evitando que realice una carga superior a la porción asignada. Pese a que sigue habiendo acceso compartido al bus, se mitiga el efecto de la interferencia causada por otras tareas en ejecución.

Se pueden encontrar diferentes soluciones de la técnica mencionada para sistemas multiprocesador. En (Yun et al., 2013), se presenta MemGuard, una solución que reserva una porción específica del ancho de banda disponible para una tarea o aplicación particular regulando las peticiones realizadas por cada CPU. De esta manera, se garantiza el rendimiento de una tarea sin que ésta se vea tan afectada por la contención de otras que accedan a memoria simultáneamente. La solución GPUGuard mostrada en el apartado anterior, se basa en MemGuard para garantizar el aislamiento entre las fases de memoria. Otra solución propuesta para la reducción de la contención en la memoria mediante la reserva del acceso al ancho de banda en plataformas multiprocesador es BWLOCK (Yun et al., 2017). Dicha solución se basa en técnicas de restricción de recursos (*Throttling Based Techniques*) para aumentar el control sobre la contención. Si hay al menos una tarea de tiempo real y una tarea que no es de tiempo real que excede el umbral del ancho de banda asignado, se realiza una restricción en la CPU donde se está ejecutando la última para reducir sus accesos a memoria. La restricción se basa en reducir la frecuencia de la CPU que ejecuta una tarea que no es de tiempo real para minimizar sus accesos a memoria reduciendo así los bloqueos por el consumo de ancho de banda.

Basándose en los anteriores trabajos para sistemas multiprocesador, (Ali and Yun, 2018) propone una solución para plataformas heterogéneas con GPU integrada llamada BWLOCK++. Esta técnica se limita a proteger las tareas críticas que requieren el uso de la GPU de la interferencia por contención de memoria provocadas por otras tareas de CPU no críticas. Para proteger la ejecución de un *Kernel* ejecutándose en la GPU, cuando una tarea de CPU no crítica que está ejecutándose simultáneamente supera su cuota de ocupación del ancho de banda, el entorno BWLOCK++ aplica una restricción a la CPU donde se ejecuta dicha tarea.

Uno de los problemas que presenta la solución anterior, aparte de estar pensada para sistemas de tiempo real laxo, es que afecta muy negativamente a las tareas que no son de tiempo real. Para mitigar este efecto y mejorar el rendimiento del sistema, en (Aghilinasab et al., 2020) se muestra un esquema de restricción que optimiza los resultados obtenidos en BWLOCK++. Se basa en ajustar dinámicamente el ancho de banda asignado a las tareas de CPU que no son de tiempo real, aprovechando la holgura en el tiempo de respuesta de las tareas críticas que requieren el uso de la GPU. Es decir, si el sistema observa que la tarea crítica va a cumplir con cierta holgura su plazo, se reduce la restricción en las CPU. Para llevar esto a cabo, la solución presenta un método para estimar dinámicamente el tiempo de ejecución de un *Kernel* en la GPU. Este método se basa en medir el progreso de ejecución del *Kernel* observando cuantos bloques de hilos GPU se han completado. Además, la estimación de tiempo de ejecución tiene en cuenta la posible interferencia que pueda haber.

3.4. Kernel Persistente

La técnica del *Kernel* Persistente es una estrategia de optimización utilizada en las plataformas con GPU para mejorar el rendimiento en aplicaciones con cargas de trabajo intensivas y repetitivas. En el modelo estándar de ejecución en una GPU, las copias de datos y los lanzamientos de los *Kernels* se realizan cada vez que se requiera procesar datos en la GPU. Además de aumentar las sobrecargas en los tiempos de respuesta de las aplicaciones, aumenta la ocupación del bus de acceso a la memoria. La técnica del *Kernel* Persistente aborda este problema minimizando la sobrecarga asociada al lanzamiento de los *Kernels*. En lugar de lanzar y finalizar la ejecución de un *Kernel* cada vez que se necesita ejecutar una tarea, el *Kernel* es lanzado una única vez y éste se ejecuta en un bucle. Cada vez que se requiere de la computación del *Kernel*, la CPU le facilita los datos a la GPU y le indica, mediante un mecanismo de sincronización, que tiene los datos disponibles para la ejecución de la tarea. De esta manera, es posible ejecutar repetidamente un *Kernel* mitigando el efecto del lanzamiento y traspaso del mismo.

Diferentes trabajos han abordado la técnica del *Kernel* Persistente. En (Gupta et al., 2012), se muestra una definición formal inicial para esta técnica, exponiendo los casos de uso en los que su aplicación puede ser beneficiosa para el rendimiento y respuesta temporal del sistema. Esta teoría es aplicada por (Capodieci and Burgio, 2016), donde utilizan el modelo de ejecución del *Kernel* Persistente para ejecutar algoritmos genéticos. Estos algoritmos necesitan iterar una gran cantidad de veces antes de obtener un resultado válido, por lo que encaja perfectamente con el enfoque del *Kernel* Persistente.

En (Kim et al., 2022), se describe otra manera diferente de aplicar la técnica del *Kernel* Persistente. A diferencia del ejemplo mencionado previamente, en este enfoque cada iteración del *Kernel* se ejecuta con datos independientes para cada iteración, y no se basa en los resultados generados en iteraciones anteriores. Para ello es necesaria una sincronización más compleja

entre la GPU y la CPU. En el estudio mencionado, los datos se transmiten de un dispositivo a otro mediante copia estándar. Para indicar que se ha completado cada operación de copia se utilizan *flags* ubicados en un espacio de memoria común a CPU y GPU.

En los estudios mencionados anteriormente se puede observar cómo se eliminan las sobrecargas asociadas al lanzamiento repetitivo de cada *Kernel* al utilizar las técnicas del Kernel Persistente. Pese a esto, las aplicaciones que requieren datos actualizados en cada iteración del *Kernel*, presentan latencias asociadas a la copia de memoria desde un espacio de memoria a otro. Para optimizar el traspaso de datos de la CPU a la GPU una opción es utilizar la política de cero copia. De esta manera, se mitigan las latencias por la copia de datos creando un espacio de memoria común visible tanto para la CPU como para la GPU. Los estudios (Milluzzi and George, 2017; Calderón et al., 2021, 2022) tratan este tema. La sincronización entre la CPU y la GPU sigue teniendo un papel muy importante. Cuando una de las unidades de procesamiento escribe los datos en la memoria, éstos deben sincronizarse mediante *flags* en la memoria global que indiquen a la GPU cuándo los tiene disponibles para ser procesados y, del mismo modo, indicarán cuándo la CPU tiene disponibles los resultados del procesamiento en la GPU.

3.5. Kernel expulsable

En (Chen et al., 2017), se muestra una solución software que habilita una planificación con expulsión para la GPU llamada EffiSha, que permite expulsar el *Kernel* en ejecución dentro de la GPU. Esta técnica permite ejecutar tareas en la GPU con una política de planificación expulsable mitigando así latencias y bloqueos no deseados. El entorno de ejecución se divide en dos partes: una herramienta de transformación de código y el servicio o "demonio" EffiSha. En primer lugar, el entorno se encarga de modificar el código del *Kernel* para que se ejecute como un *Kernel* Persistente que comprueba en cada iteración si es necesario que sea expulsado. El servicio EffiSha se encarga de indicar a los *Kernel* en ejecución cuándo tienen que ser expulsados, y de relanzarlos cuando sea debido. Para ello, EffiSha utiliza dos políticas de planificación expulsables: una basada en prioridades fijas expulsables (P-FP) y la otra en prioridades dinámicas. La idea está tomada de (Basaran and Kang, 2012) donde dividen el *Kernel* en fragmentos pequeños para poder expulsar la tarea de la GPU cuando termina cada fragmento. Al utilizar *Kernel* Persistente y dado que los puntos de expulsión se encuentran al final de cada iteración del *Kernel*, no se necesita trabajo adicional para guardar o restaurar los estados de los hilos, ya que todas las tareas han finalizado. El trabajo original presenta una política de planificación basada en prioridades fijas (*High*, *Middle*, *Low*) denominada *Preemptive Kernel Mode* (PKM). La planificación PKM no solo tiene en cuenta los bloques que componen los *Kernels* sino que también planifica las operaciones de copia.

En (Hartmann and Margull, 2019), se presenta GPUart, un planificador para tareas de GPU que habilita la planificación con expulsión del *Kernel*. Se basa en permitir la expulsión del *Kernel* dentro de cada bloque de hilos añadiendo puntos fijos de expulsión, técnica que se denomina *Limited Preemptive*. Las tareas en la GPU se planifican de manera EDF expulsable (P-EDF) o FP expulsable (P-FP). GPUart esta diseñado para funcionar en tres capas diferentes: (1) capa de abstracción, que proporciona una interfaz de llamadas para activar un *Kernel* en el planificador y recibir una notificación cuando finaliza su ejecución; (2) capa de implementación, que contiene la implementación de todos los *Kernels* del sistema a ejecutar en la GPU; (3) capa de planificación, que planifica los *Kernels* y se encarga de realizar las peticiones de expulsión en la GPU. El planificador indica al *Kernel* mediante *flags* en la memoria compartida (con ZC) en qué punto de expulsión debe parar su ejecución en la GPU. Una vez llegado a ese punto, se guardan los datos en un *buffer* y cuando se vuelve a lanzar el *Kernel*, se le indica desde qué punto comenzar y se le entregan los datos. En los resultados obtenidos, se muestra cómo en el caso de prueba descrito no solo se reducen los tiempos de respuesta utilizando GPUart (en un factor de 221), sino que todas las tareas cumplen sus plazos temporales. Sin embargo, el uso de esta técnica produce un incremento en los WCETs entre el 11 % y 17.26 % debido a la inserción de los puntos de expulsión en el código.

En (Ayala-Barbosa and Mendez-Monroy, 2022), se presenta un entorno para la planificación con expulsión para plataformas CPU-GPU integradas. El objetivo es planificar las tareas de GPU de una manera expulsable para mejorar el cumplimiento de plazos temporales. Al igual que en el caso anterior, para habilitar la expulsión de *Kernels* en la GPU es necesario editar su código añadiendo puntos de expulsión. El *Kernel* en ejecución solo se puede expulsar en uno de estos puntos. Cuando se expulsa el *Kernel*, guarda todas las variables en un espacio de memoria común entre la CPU y la GPU. Cuando la CPU requiere reanudar la tarea en la GPU, se vuelve a lanzar el *Kernel* indicándole desde qué punto de control arrancar y en qué dirección de la memoria compartida tiene los datos que necesita. El trabajo destaca que cualquier política de expulsión es válida, pero proponen utilizar como referencia la política EDF expulsable.

En (Lee et al., 2021), se presenta una solución para habilitar una expulsión de manera abrupta del *Kernel* de la GPU. En lugar de insertar puntos de expulsión mediante inserción de código en el *Kernel* como en trabajos anteriores, este nuevo enfoque afronta la problemática de que si un *Kernel* es expulsado abruptamente, la GPU descarta el contexto y no es posible reanudarlo más adelante. Como solución los autores proponen modificar el *driver* OpenCL. El planificador clasifica los *Kernels* en idempotentes y no idempotentes. Si es idempotente, gracias a la modificación del *driver*, es posible reanudar la ejecución de un *Kernel* expulsado simplemente relanzándolo. En el caso contrario, el Sistema Operativo toma una captura del espacio de memoria (estado del *Kernel*, variables, configuración, etc.) cuando se lanza un *Kernel* de mayor prioridad expulsando al que estaba en ejecución. Cuando se relanza el *Kernel* expulsado, se vuelve a copiar la captura obtenida anteriormente. Esta técnica permite una expulsión más abrupta del *Kernel*, pero al realizar muchas copias de datos, puede incurrir en grandes sobrecargas cuando el uso de la GPU es alto.

3.6. Particionado en la GPU

Como se ha mostrado anteriormente en (Otterness et al., 2016, 2017), dependiendo de las aplicaciones lanzadas a la GPU, puede ser recomendable lanzar varios *Kernels* concurrentemente aumentando la utilización de la GPU pese a la interfe-

rencia entre *Kernels* que pueda provocarse. El problema de optimizar la utilización de la GPU es que, por defecto, no existen mecanismos para el programador que permitan la asignación de *Kernels* a SMs. De tal manera que no hay ningún control sobre el uso de los recursos por parte de la GPU. El objetivo de las técnicas basadas en el particionado de la GPU es que el programador tenga el control sobre la asignación de los bloques del Kernel a los SM, y que esto permita optimizar el rendimiento de la ejecución en la GPU, optimizar la utilización y reducir los tiempos de ejecución.

En (Wu et al., 2015) consiguen particionar espacialmente la GPU mediante la asignación de bloques a SM. Este mecanismo se basa en el uso de *Kernels* Persistentes para la asignación de bloques a SM. En cada SM se ejecuta un hilo persistente al que se le indica cuál es el trabajo que debe realizar. El programador puede seleccionar en qué SM se debe ejecutar cada trabajo mediante un identificador. De esta manera, cuando se le indica a cada *Kernel* que tiene un trabajo que realizar, éste selecciona el trabajo con el identificador que coincide con el del SM en el que se está ejecutando. Gracias a esta técnica el programador tiene el control sobre la asignación de *Kernels* a SM en vez de depender del planificador interno de la GPU. Como resultado, se mejoran los tiempos de respuesta de los *Kernels* en la GPU. Pese a que el código añadido al *Kernel* original puede añadir sobrecargas en los tiempos de ejecución (2.8 % de media), éstas son despreciables en comparación con los beneficios obtenidos al acelerar los tiempos de respuesta en un 21 % de media. Además, se ha visto que mediante esta asignación de bloques también se reducen los fallos de caché al compararlo con el planificador por defecto. Gracias a esta asignación, son posibles técnicas como FractionalGPU (Jain et al., 2019) mostrada en el apartado 3.1 sobre técnicas basadas en coloración de página.

En (Janzén et al., 2016), se muestra otra técnica con la que se puede mejorar el rendimiento del sistema al particionar la GPU. Se basa en una técnica software que proporciona aislamiento entre *Kernels* redistribuyendo los bloques de ejecución en los SM deseados. Para conseguir esta asignación de *Kernel* a SM, primero se calcula cuántos bloques de ejecución por *Kernel* hacen falta para llenar un SM, y cuántos bloques son necesarios para llenar la GPU. Entonces, se lanza el *Kernel* con la cantidad necesaria para ocupar al completo la GPU. Un código de control dentro del *Kernel* verifica, mediante el ID único que tienen estas unidades de procesamiento, si la ejecución se está realizando en el SM deseado. Si no fuera así, el bloque termina su ejecución. De esta manera, el *Kernel* se ejecuta en los SM asignados, dejando el resto completamente libres para otros *Kernels*. Los resultados muestran que se mejora el rendimiento de la GPU en un 9 % al disminuir los tiempos de respuesta.

Similar al caso anterior, (Saha et al., 2019) presenta un entorno de ejecución llamado STGM. El objetivo principal es particionar la GPU para aumentar su utilización, y al igual que en la solución mostrada anteriormente, utilizan el ID propio de los SM para indicar a las tareas lanzadas en qué SM tienen que ejecutarse. Si el ID asignado a una tarea no coincide con el asignado al *Kernel*, éste detiene su ejecución. Esta técnica reduce los tiempos de respuesta mejorando la planificabilidad del conjunto de tareas. Además, presentan un análisis para estimar los bloqueos y la respuesta en el peor de los casos.

En (Bakita and Anderson, 2023), se logra particionar la ejecución en la GPU activando campos existentes, pero poco conocidos, en una estructura de datos de NVIDIA llamada TMD (*Task MetaData*), utilizada a la hora de transferir cómputo a la GPU y asignada al *Kernel*. Mediante ingeniería inversa en la GPU, descubren que dentro de la estructura TMD existe una máscara de bits que activan o desactivan los SMs para cada *Kernel*. Logrando modificar los valores de esta máscara, se proporciona un aislamiento espacial de los SMs al indicar a cada *Kernel* qué SM tiene activos y cuáles desactivados. Al tener el control de la asignación de bloques a SM se logra aumentar la utilización de la GPU evitando bloqueos entre *Kernels* y reduciendo los tiempos de respuesta.

3.7. Planificación del acceso a la GPU

En este apartado se muestran diferentes técnicas que proponen controlar el acceso a la GPU mediante implementaciones que funcionan como planificadores. De esta manera se busca no depender únicamente del planificador interno de la GPU, cuya política y estructura interna es habitualmente desconocida. Mediante políticas de planificación definidas por el desarrollador, estas implementaciones pueden garantizar que las tareas de GPU se ejecuten de una manera más predecible y garantizando los plazos de tiempo impuestos.

En (Kim et al., 2018), se propone una solución basada en un hilo servidor que recibe y gestiona peticiones de tareas que requieren el uso de la GPU. Solo se permite acceder a la GPU a una tarea a la vez, suspendiendo las demás tareas que también han realizado una petición al servidor. Este servidor planifica las tareas mediante prioridades fijas, por lo que si tiene más de una petición, ejecuta las tareas en orden de mayor a menor prioridad. Las tareas no son expulsables, por lo que, si hay una tarea de menor prioridad ejecutándose, una de mayor prioridad que realice la petición en ese momento puede ser bloqueada. Estos bloqueos producen aumentos en los tiempos de respuesta, que deben ser tenidos en cuenta a la hora de realizar el análisis de planificabilidad. La ventaja que aporta esta técnica es que el sistema no depende del planificador interno de la GPU, ya que es el servidor el que gestiona la ejecución en el acelerador aumentando la predictibilidad del sistema.

Una solución similar basada en un middleware que funciona como un servidor para gestionar tareas de GPU es CARSS (Baek et al., 2020). Como en el caso anterior, las diferentes tareas realizan peticiones al servidor notificando su intención de acceder al acelerador hardware. Este servidor es capaz de arbitrar el acceso a la GPU de los trabajos en base a tres políticas de planificación de manera no expulsable: (1) *Least Slack First* (LSF) que planifica dinámicamente las tareas de menor tiempo de holgura primero, (2) *Rate Monotonic* (RM) que planifica las tareas de menor periodo primero y, por último, (3) *Earliest Deadline First* (EDF) que planifica las tareas con plazos más cercanos primero. Proporcionando diferentes casos de uso práctico, se demuestra cómo mejora el rendimiento en diferentes plataformas heterogéneas. Entre un 1 % y un 7.5 % en plataformas discretas, y entre un 1.5 % y un 9.1 % en integradas.

En los estudios (Marchi et al., 2021) y (Li et al., 2023), se muestran diferentes soluciones basadas en el middleware ROS2 (*Robot Operating System* 2). En ambos casos una aplicación de ROS2 ejerce de servidor arbitrando el acceso a la GPU de otras

aplicaciones ROS2. El primero de los métodos aplica diferentes políticas para el traspaso de datos mediante la memoria compartida (expuestas en el apartado 2.3), para ver cúal es la óptima en cada caso. El segundo trabajo, en cambio, muestra los resultados de rendimiento aplicando diferentes políticas de planificación para las tareas de GPU, llamadas *Exclusive Policy* (EP), *Sharing Policy* (SP) y *Preemption Policy* (PP).

En (Elliott et al., 2013), se presenta GPUSync, un entorno para la sincronización del acceso de tareas a una o múltiples GPUs. Gestiona los accesos de las tareas a las GPUs y asegura que ninguna tarea acceda a una GPU mientras otra esté ejecutando, evitando que interfiera en su ejecución. De esta manera se puede ejecutar las tareas con mayor predictibilidad. Además, es compatible con diferentes políticas de planificación como RM o EDF.

En (Xu et al., 2016), proponen algoritmos para la planificación basándose en técnicas de servidores y particionado. Se abordan desde tareas ejecutadas exclusivamente en la CPU (tareas de tiempo real y tareas de no tiempo real), hasta tareas tanto de tiempo real como *Best-Effort* (BE) que hacen uso de la GPU. El sistema se compone de diferentes particiones, cada una con un servidor, para cada tipo de tarea mencionada anteriormente. Tras descomponer las tareas en servidores, un planificador global planifica el acceso de cada partición a los recursos (CPU y GPU). A cada CPU se le asigna una partición con un servidor de tareas de tiempo real, que son planificadas por el servidor. Las tareas BE para la CPU son planificadas globalmente. Para la GPU, en cambio, existen dos servidores, el servidor de tiempo real y el servidor BE. El servidor de tiempo real para la GPU planifica las tareas localmente mediante EDF y el servidor BE mediante prioridades fijas. Utilizan tres diferentes esquemas para compartir el acceso la GPU entre los dos servidores: (1) el servidor BE solo tiene acceso a la GPU si no hay ninguna petición de un servidor de tiempo real, (2) mediante particionado en la GPU (similar al mostrado en el apartado 3.6) se permite que las tareas RT y BE puedan acceder simultáneamente a la GPU en SMs aislados, (3) se calcula una estimación de holgura para las tareas de tiempo real y así reducir el tiempo de respuesta de las tareas BE.

En (Suzuki et al., 2016), se presenta GPUvm, una arquitectura para la virtualización del acceso a la GPU en el nivel de hipervisor. Una máquina virtual (VM, *Virtual Machine*) que requiere el uso de la GPU realiza una llamada al hipervisor. Mediante esta llamada el hipervisor planifica el acceso de cada VM a la GPU. Utiliza un algoritmo de planificación denominado *Bandwidth-Aware Non-preemptive Device* (BAND) (Kato et al., 2012). Cuando una VM intenta ejecutar una tarea en la GPU, el algoritmo mira si hay alguna tarea en curso antes de permitir el acceso. Si hay una VM ocupando la GPU, se encola la petición y la tarea correspondiente se queda suspendida hasta que se libera el recurso. Cuando la GPU se libera, el planificador concede el acceso a la siguiente VM.

El trabajo (Zhou et al., 2018) presenta S^3DNN, un entorno diseñado para planificar y optimizar la ejecución de tareas basadas en redes neuronales profundas (DNNs, *Deep Neural Networks*) orientadas a la detección de objetos. Por un lado, el entorno toma como entrada las múltiples fuentes de imágenes a las que se les quiere aplicar la detección de objetos. Por otro lado, planifica la entrada de las diferentes instancias del algoritmo de DNN mediante prioridades dinámicas. La política de planificación utilizada para el acceso a la GPU es la conocida como *Least Slack First* (LSF). Con esto consiguen mejorar los tiempos de respuesta para el cumplimiento de plazos.

En (Capodieci et al., 2018) se presenta un prototipo de planificador EDF con expulsión para plataformas GPU integradas. El sistema se basa en un hipervisor disponible solo en algunas plataformas específicas de NVIDIA como la familia NVIDIA Drive. Mediante aislamiento espacial en la GPU, se logra que diferentes particiones accedan concurrentemente a la GPU como si fuese cualquier otro recurso compartido del sistema particionado. El planificador funciona por encima del hipervisor. Se añade el mecanismo *Constant Bandwith Server* (CBS) (Abeni et al., 1998) al planificador EDF que proporciona aislamiento entre tareas ante un mal comportamiento temporal de una aplicación.

En (Bateni et al., 2020) se presenta un entorno de desarrollo para arquitecturas CPU-GPU integradas que estima cuál es el método de acceso a la memoria óptimo para cada tarea GPU, y planifica las tareas según la estimación realizada. El entorno de desarrollo considera SC (*Standard Copy*), UM (*Unified Memory*) y ZC (*Zero Copy*), como métodos de traspaso de datos por memoria compartida. El planificador utiliza la política EDF para mantener la cola de tareas lista para ejecutar, mientras decide qué *Kernels* son óptimos para co-ejecutarse teniendo en cuenta su configuración de memoria. Por ejemplo, aprovechando los tiempos de copia de datos de una tarea GPU para lanzar un *Kernel* que no requiere copia de datos de un espacio a otro. Llama a esta política *Memory Management Policy* (MM Policy). Mediante esta técnica reducen los tiempos de respuesta en un 10 % de media para una aplicación de movilidad inteligente y entre el 58.9 y el 11.2 % para una aplicación de detección de obstáculos para drones.

En (Kang et al., 2021) se propone LaLaRAND, un entorno para la planificación de DNNs que permite un esquema de asignación CPU-GPU para mejorar y garantizar las tareas dentro de una DNN de tiempo real. La manera tradicional de planificar este tipo de algoritmos se basa en la asignación estática de capas DNN en cada recurso (CPU o GPU). En la planificación propuesta, las capas de la DNN se asignan dinámicamente a la CPU o a la GPU siguiendo con una política definida como *System-Wide Scheduler* (SWS). Bajo esa política, se calculan los costes de ejecución en cada uno de los recursos (en base al uso de memoria, sincronización, etc.) y se planea la asignación óptima para garantizar el cumplimiento de plazos temporales. Además, este entorno integra una técnica de cuantificación que optimiza los tiempos de ejecución del algoritmo en la CPU. Los resultados obtenidos muestran que en el caso de estudio evaluado en una plataforma GPU integrada, el rendimiento general de la aplicación empeora hasta un 0.4 % al implementar el entorno, pero se consigue una mejora de la planificabilidad entre un 56 % y un 80 %.

En (Yao et al., 2022), se presenta $WAMP^2S$, una plataforma software que implementa un prototipo de planificador para la GPU. Consiste en un planificador pseudoexpulsable que planifica el acceso a la GPU de diferentes máquinas virtuales o contenedores. Se denomina planificación pseudoexpulsable, pues no expulsa directamente los *Kernels* de la GPU, sino que ajusta dinámicamente el acceso a la GPU de las diferentes máquinas

virtuales. Utiliza un algoritmo llamado *Priority-Based Greedy Scheduling Policy* (PBGSP) para la planificación del acceso a la GPU en base a prioridades y de manera pseudoexpulsable. WAMP^{2}S se evalúa en un hardware con una GPU discreta concluyendo que, en el caso de estudio probado, la planificabilidad aumenta en un 68.2 %. Uno de los inconvenientes de la plataforma es la sobrecarga que introduce debida a la planificación. Esta sobrecarga oscila entre 20 y 180 μs dependiendo del número de aplicaciones ejecutándose al mismo tiempo. Mediante técnicas de optimización consiguen reducir estas sobrecargas hasta un 60.1 % de media.

El trabajo (Roeder et al., 2021) muestra otro enfoque para la planificación de tareas en arquitecturas CPU-GPU integradas basado en *Fordward List Scheduling* (FLS). En la planificación FLS, primero se ordenan las tareas de una manera determinada para ser posteriormente planificadas en ese orden, existiendo múltiples políticas para ordenar las tareas. En el trabajo mencionado, el algoritmo propuesto toma un DAG (*Directed Acyclic Graph*) como entrada, proporciona un tiempo de activación para cada tarea y evalúa si una tarea puede dar lugar a interferencia de memoria. Si prevé que una tarea puede sufrir de interferencia, propone un nuevo tiempo de activación y estima su nuevo WCET. Una vez evaluados los tiempos de activación y los nuevos WCET, se planifican las tareas en base a las nuevas estimaciones. En los resultados obtenidos, la planificabilidad del sistema aumenta entre un 11 % y un 24 %.

En (Zou et al., 2023), se presenta RTGPU, un entorno que planifica el acceso a la GPU de manera paralela con el objetivo de aumentar la utilización de la GPU, haciendo así que se pueda aumentar también la planificabilidad del sistema. Además, el trabajo presenta un análisis de tiempos de respuesta con técnicas para reducir el pesimismo. La solución presentada se centra en arquitecturas GPU discretas. En primer lugar, para poder planificar un acceso paralelo en la GPU, es necesario particionar los recursos (SMs) de la GPU. Para ello, se basan en el uso de *Kernels* Persistentes asociados a un SM donde el planificador le comunica a cada *Kernel* Persistente qué trabajo debe realizar, consiguiendo así la asignación de *Kernel* a SM. La técnica es similar a la mostrada por (Wu et al., 2015) en el apartado 3.6. El entorno propuesto planifica las tareas del lado de la CPU en base a prioridades fijas mientras que utiliza una política de *Federative Scheduling* para la GPU. Esta política tiene en cuenta los recursos que necesita un *Kernel* para cumplir sus plazos temporales, y en función de ello, realiza la asignación de *Kernel* a SM y decide cuántos SMs dar a cada uno. Los resultados del trabajo muestran una mejora en la planificabilidad de un 57 % respecto a trabajos anteriores.

3.8. Planificación del acceso a la memoria

En (Li and Aamodt, 2016) se propone una política de planificación que gestiona el acceso al controlador de memoria con el objetivo de reducir la interferencia entre núcleos GPU. Se encarga de gestionar mediante hardware las peticiones de memoria de diferentes núcleos GPU para reducir los fallos de caché provocados por el acceso compartido. Coordina la LLC con el controlador de la memoria global para priorizar las peticiones con mayor *Inter-Core Locality* (ICL), es decir, el punto en el que un mayor número de núcleos acceden a un mismo dato de lectura en una misma linea de caché, el dato no se encuentra disponible y las peticiones se fusionan para realizar una sola lectura de la memoria global. En los resultados obtenidos, las latencias producidas por los fallos de caché se reducen en un 28 % de media.

El artículo (Fang et al., 2020) propone una estrategia de planificación de memoria paso-a-paso, ubicada en el MC, con el objetivo de mitigar la interferencia en la memoria. Primero, el MC identifica el origen de las peticiones de memoria. Después, aísla las peticiones de la CPU de las peticiones de la GPU para que estas últimas no interfieran con las primeras. Para las peticiones de la CPU, implementa una estrategia de partición dinámica de bancos de memoria. En cambio, para las peticiones de la GPU, se asignan prioridades a las peticiones en base a su criticidad. Un núcleo GPU se considera crítico cuando tiene un alto ratio de peticiones de memoria pendientes en cola. La estrategia de planificación mostrada se llama *Core Criticality-Aware Memory Scheduling* y tiene como objetivo priorizar las peticiones de memoria en función de las características de la aplicación y de los accesos a la memoria para mejorar el rendimiento del sistema. De esta manera se consigue mitigar la interferencia en el acceso a la memoria, logrando una mejora en el rendimiento del sistema de un 17 % de media.

4. Modelado y análisis de planificabilidad

En este apartado se ha realizado una revisión de trabajos recientes acerca del modelado y análisis de planificabilidad de sistemas de tiempo real con arquitecturas GPU integradas.

En (Höttger et al., 2019), los autores presentan un análisis de tiempos de respuesta para tareas en la CPU totalmente expulsables y planificadas por *Rate Monotonic* y utilizando una técnica de ventanas temporales en la GPU. Esto se combina con un análisis para tareas en la GPU basado en una planificación *Weighted Round Robin* (WRR). Además, definen un modelo de contención de memoria para los CE de la GPU en el que distinguen entre los mecanismos de traspaso de datos asíncronos y síncronos de la GPU. El análisis de tiempo de respuesta de la CPU considera la latencia debido al acceso a datos de las tareas, la latencia por contención en la memoria y los traspasos de datos síncronos o asíncronos. El análisis de la GPU, en cambio, tiene en cuenta el tiempo requerido por los traspasos de datos del CE y también la interferencia en la memoria provocada por el acceso a la misma de otras tareas en la CPU.

En (Diewald et al., 2019), se presenta un análisis de tiempos de respuesta para tareas ejecutadas en plataformas heterogéneas con aceleradores hardware y transferencia de datos por DMA. Se centran en dos efectos principales dentro de las plataformas heterogéneas: (1) la sincronización entre la computación y la transferencia de datos, y (2) el uso de técnicas *prefetching* para aumentar el rendimiento de la plataforma. Las técnicas *prefetching* se basan en precargar los datos en la memoria o caché antes de que la tarea los necesite. De esta manera, se consigue evitar las latencias provocadas por la carga de datos, y se mejora el rendimiento del sistema al aumentar el paralelismo entre la CPU y el DMA. Una de las limitaciones de este estudio es que no trata las interferencias en la memoria producidas por la transferencia de datos del DMA. Este análisis podría ser compatible con el modelo de ejecución PREM, pues ambos consideran consideran una precarga y una descarga de datos antes y

después de la computación de la tarea.

En (Krawczyk et al., 2019), se muestra un análisis de tiempos de respuesta de principio a fin. Para este análisis, se asume que todas las tareas (tanto las de la CPU como los CE y EE de la GPU) son expulsables. También se asume que todas las tareas acceden a la memoria global al inicio y al final de una operación de copia CPU-GPU. El análisis tiene en cuenta estas latencias producidas por el acceso a la memoria tanto por parte de la CPU como de la GPU. También considera las latencias en el peor de los casos debido a la contención en la memoria. Las tareas de la CPU se planifican en base a prioridades fijas (P-FP), mientras que para el caso de la GPU, se asume que los *Kernels* se planifican de manera WRR con ventanas de tiempo fijas. El trabajo incluye técnicas de optimización para asignación de ta-

Tabla 2: Clasificación de las técnicas revisadas para acotar y optimizar parámetros temporales.

Trabajo	Técnica	Problema abordado			Arquitectura GPU
		Operaciones GPU	Interferencia de memoria	Planificación	
(Jain et al., 2019)	Coloración de página		GPU		Discreta
(Houdek et al., 2017)	Modelo de ejecución predecible		CPU-GPU		Integrada
(Forsberg et al., 2017b)			CPU-GPU		Integrada
(Forsberg et al., 2019)			CPU-GPU		Integrada
(Forsberg et al., 2021)			CPU-GPU		Integrada
(Forsberg et al., 2017a)			CPU-GPU		Integrada
(Capodieci et al., 2017)			CPU-GPU		Integrada
(Spliet and Mullins, 2022)			GPU		Integrada
(Ali and Yun, 2018)	Reserva de ancho de banda		CPU-GPU		Integrada
(Aghilinasab et al., 2020)			CPU-GPU		Integrada
(Gupta et al., 2012)	*Kernel* Persistente	Traspaso *Kernel*			Discreta
(Capodieci and Burgio, 2016)		Traspaso *Kernel*			Discreta
(Kim et al., 2022)		Traspaso *Kernel*			Discreta
(Milluzzi and George, 2017)		Traspaso *Kernel* y Datos			Integrada
(Calderón et al., 2021)		Traspaso *Kernel* y Datos			Integrada
(Calderón et al., 2022)		Traspaso *Kernel* y Datos			Integrada
(Chen et al., 2017)	*Kernel* expulsable			P-FP y Dinámicas	Discreta
(Basaran and Kang, 2012)				PKM	Discreta
(Hartmann and Margull, 2019)				P-FP, P-EDF	Integrada
(Ayala-Barbosa and Mendez-Monroy, 2022)				P-EDF	Integrada
(Lee et al., 2021)					Integrada
(Wu et al., 2015)	Particionado en la GPU			*Kernel* a SM	Discreta
(Janzén et al., 2016)				*Kernel* a SM	Discreta
(Saha et al., 2019)				*Kernel* a SM	Ambas
(Bakita and Anderson, 2023)				*Kernel* a SM	Ambas
(Kim et al., 2018)	Planificación en el acceso a la GPU			FP	Integrada
(Baek et al., 2020)				LSF, RM, EDF	Ambas
(Marchi et al., 2021)		Traspaso de Datos		Planificador ROS2	Integrada
(Li et al., 2023)				EP, SP, PP	Discreta
(Elliott et al., 2013)				RM, EDF	Discreta
(Xu et al., 2016)				EDF, FP + *Kernel* a SM	Discreta
(Suzuki et al., 2016)				BAND	Discreta
(Zhou et al., 2018)				*Least Slack First*	Discreta
(Capodieci et al., 2018)				EDF+CBS	Integrada
(Bateni et al., 2020)				EDF + MM Policy	Integrada
(Kang et al., 2021)				*System-Wide Scheduler*	Integrada
(Roeder et al., 2021)			CPU-GPU	FLS	Integrada
(Yao et al., 2022)				PBGSP	Discreta
(Zou et al., 2023)				*Federated Scheduling*	Discreta
(Li and Aamodt, 2016)	Planificación en el acceso a memoria		GPU	ICL *Aware*	Discreta
(Fang et al., 2020)			CPU-GPU	*Core Criticality-Aware*	Integrada

reas a CPU y asignación de ventanas para la planificación WRR de la GPU.

El trabajo (Casini et al., 2022) presenta un modelo y análisis de tiempos de respuesta para aplicaciones de tiempo real en sistemas particionados con plataformas heterogéneas basadas en aceleradores hardware. Considerando el acelerador hardware como un recurso compartido, realiza un análisis de tiempos de respuesta de principio a fin considerando las tareas de la CPU que requieren el uso del acelerador hardware como tareas auto-suspendidas. De esta manera, una tarea auto-suspendida se compone de tres partes: (1) una fase de ejecución que corresponde con la fase de descarga, (2) la fase de ejecución en el acelerador que se corresponde con la fase auto-suspendida desde el punto de vista de la CPU, y (3) otra fase de ejecución para la finalización de la tarea. Cada una de las fases tiene su WCET. Las tareas en la CPU son planificadas basándose en prioridades fijas, mientras que en la GPU se consideran dos políticas de planificación que pueden ser aplicadas directamente por el acelerador o, por un planificador externo del nivel de aplicación: *Round Robin* (RR) y *Non-Preemptive Frixed Priority* (NP-FP). Debido al modelo de auto-suspensión, se considera que las tareas de la GPU son síncronas. También incluye una optimización de asignación de tareas a CPU y asignación de prioridades. Los autores muestran en (Casini and Biondi, 2022) otro análisis de tiempos de respuesta de aplicaciones en sistemas con particiones temporales sobre hardware heterogéneo, considerando algoritmos de planificación EDF particionado para las tareas en la CPU.

Otro enfoque para el modelado de aplicaciones de tiempo real es el basado en DAGs (*Directed Acyclic Graphs)*. En (Serrano and Quiñones, 2018), se presenta un análisis de tiempos de respuesta para tareas DAG que incluye computación en aceleradores hardware como GPUs o FPGAs. Proponen un algoritmo que transforma el DAG de una tarea para reducir los bloqueos generados en la CPU por las cargas de trabajo ejecutadas en el acelerador. El algoritmo de transformación identifica qué tareas pueden ejecutarse en paralelo a la carga de trabajo en el acelerador y reorganiza el DAG de manera que las tareas en la CPU se ejecuten en paralelo sin que la carga en el acelerador produzca tiempos de bloqueo.

En otro estudio acerca del modelado basado en DAGs (Houssam-Eddine et al., 2021), se desarrolla el modelo HPC-DAG y su análisis de planificabilidad para plataformas heterogéneas. El modelo se centra en plataformas NVIDIA, pero la idea es que pueda ser fácilmente aplicable a otras plataformas. Las tareas se dividen en una cantidad definida de subtareas, que representan las unidades básicas de computación. Cada una de ellas tiene una etiqueta que la vincula con una unidad de procesamiento (CPU, GPU u otros aceleradores) o motor de copia. Pese a que mencionan que su modelo puede admitir diferentes políticas de planificación, el artículo se centra primordialmente en el análisis de planificabilidad para políticas EDF. El trabajo destaca la importancia de tener en cuenta la interferencia de memoria para un análisis más preciso pero lo plantea para trabajo futuro.

En (Rehm et al., 2021) se describe la información necesaria a considerar para realizar un análisis de tiempos de respuesta en plataformas heterogéneas. Detalla el modelo de contención de memoria para la CPU y la GPU. Para calcular las latencias producidas por la contención de memoria en la CPU tiene en cuenta el tiempo de acceso a la memoria, las latencias producidas por la interferencia de otros núcleos de la CPU y la interferencia provocada por la GPU. Para el lado de la GPU en cambio, tiene en cuenta el tiempo de acceso a la memoria y la interferencia producida por los núcleos de la CPU.

En (Cucinotta et al., 2023), se aborda un método para modelar y optimizar la configuración de plataformas y la localización y aceleración de tareas organizadas como DAGs para plataformas heterogéneas. El algoritmo de optimización pretende minimizar el consumo de energía cumpliendo con los plazos temporales definidos. En el modelo definido, las tareas se planifican de manera EDF expulsable con CBS. Además, algunas tareas pueden implementar parte de su ejecución en aceleradores hardware, asumiendo que los aceleradores atienden las solicitudes de manera FIFO. El estudio no propone análisis de tiempos de respuesta, éstos son obtenidos mediante simulación.

5. Resultados y trabajo futuro

En este trabajo se ha realizado una revisión de la literatura más reciente y significativa sobre técnicas de optimización, modelado y análisis de planificabilidad para aplicaciones de tiempo real que utilizan GPUs. La Tabla 2 resume y clasifica los trabajos revisados considerando las técnicas de optimización para arquitecturas con GPU mostradas en el apartado 3. En esta tabla aparecen los datos más característicos para cada uno de los trabajos: técnica de optimización utilizada, problemas abordados y para qué tipo de arquitecturas se han validado (discretas, integradas o ambas). Dentro de los problemas abordados se muestran tres categorías. En primer lugar, se muestra si el trabajo optimiza algún tipo de operación de GPU, es decir, la copia de datos y/o el traspaso del *Kernel*. En segundo lugar, si el trabajo tiene como objetivo mitigar la interferencia en la memoria, especifica si es interferencia producida entre la CPU y la GPU o entre diferentes hilos dentro de la GPU. Por último, si la técnica utiliza alguna política de planificación, indica cuál o cuáles utiliza. Igualmente, en la Tabla 3 se clasifican las técnicas más recientes de modelado y análisis de tiempos de respuesta mostradas en el apartado 4. La tabla detalla si se considera un análisis de tiempos de respuesta (RTA), si el modelado se hace a partir de DAGs, las políticas de planificación usadas (para CPU y GPU), si tienen en cuenta la interferencia en la memoria y, en el caso de considerar optimización, qué se optimiza.

Algunas técnicas de optimización no buscan directamente garantizar el cumplimiento de plazos para aplicaciones de tiempo real, sino simplemente mejorar el rendimiento del sistema. Aunque existen otras técnicas de este tipo, en este trabajo recopilatorio se han seleccionado en este contexto las que se han considerado más relevantes para aplicaciones de tiempo real. Además, cabe destacar que dentro de las soluciones de tiempo real, existen enfoques tanto para tiempo real laxo como para tiempo real estricto, abordando diferentes niveles de exigencia en cuanto al cumplimiento de los plazos en estas aplicaciones. El objetivo de esta revisión ha sido mostrar el amplio abanico de técnicas y entornos desarrollados en los últimos años para que puedan servir de base a los investigadores que se quieran acercar a este tema.

Tabla 3: Clasificación de los trabajos revisados sobre el modelado y análisis de tiempos de respuesta.

Trabajo	RTA	DAGs	Planificación		Interferencia de memoria	Optimización
			CPU	GPU		
(Serrano and Quiñones, 2018)	✓	✓				Bloqueos en la CPU
(Diewald et al., 2019)	✓					Asig. tareas
(Höttger et al., 2019)	✓		RM	WRR	✓	
(Krawczyk et al., 2019)	✓		FP	WRR	✓	Asig. tareas y ventanas GPU
(Houssam-Eddine et al., 2021)	✓	✓	EDF			Consumo de energía
(Rehm et al., 2021)					✓	
(Casini et al., 2022)	✓		FP	RR y NP-FP		Asig. tareas y prioridades
(Casini and Biondi, 2022)	✓		EDF			Consumo de energía
(Cucinotta et al., 2023)	✓	✓	EDF + CBS			

Un punto a debatir respecto a las técnicas de optimización revisadas en este trabajo es que se basan en planificación o en entornos de ejecución software que pueden presentar inconvenientes frente a otras técnicas basadas en hardware. Las técnicas basadas puramente en software proporcionan una capa de abstracción que permite al desarrollador un control sobre el uso de la GPU que antes no tenía. Pese a que estas técnicas puedan aumentar tanto la predictibilidad como la planificabilidad del sistema, también pueden degradar el rendimiento de las aplicaciones e incurrir en sobrecargas, al contar con algoritmos de planificación más complejos, añadir más código al incorporar puntos de expulsión a los *Kernels*, etc. Estos costes constituyen un aspecto importante a considerar a la hora de implementar un sistema de estas características, pues pueden llegar a degradar su comportamiento temporal consiguiendo un efecto contrario al perseguido.

En relación al modelado de las GPUs, es un campo reciente que está en desarrollo y queda patente la necesidad de seguir investigando en este ámbito. Como se ha visto en el apartado 2, actualmente no existe un modelo claro de la GPU pues no se conoce con certeza la estructura interna y su funcionamiento, debido a que los distintos fabricantes no revelan la arquitectura y funcionamiento interno de las plataformas con GPUs. Esto hace que obtener un modelo más preciso del funcionamiento interno requiera de más investigaciones de ingeniería inversa como las mencionadas en este trabajo.

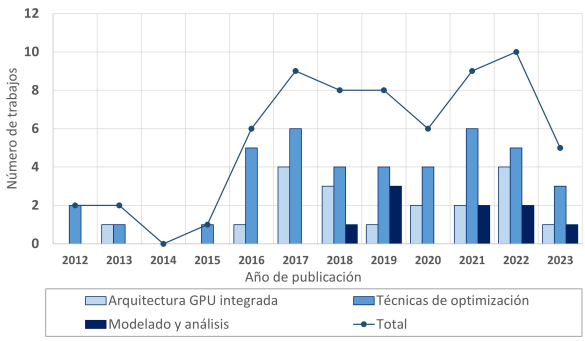

Figura 2: Distribución anual de trabajos revisados acerca del uso de GPUs en aplicaciones de tiempo real.

Por último, se puede decir que el uso de GPUs en sistemas críticos de tiempo real está ganando relevancia en el ámbito de la investigación en los últimos años. La Figura 2 muestra una gráfica con la distribución anual de los artículos revisados en este trabajo sobre el uso de GPUs en aplicaciones de tiempo real en la que se refleja esta situación. A medida que esta área siga siendo explorada, se espera que haya una amplia gama de estudios adicionales que se sumen a la revisión presentada en este trabajo. Esto motiva a seguir investigando y continuar afrontando los retos abiertos que están por resolver dentro de este campo.

Agradecimientos

Este trabajo ha sido parcialmente financiado por MCIN/ AEI /10.13039/501100011033/ FEDER “Una manera de hacer Europa” a través de las subvenciones PID2021-124502OB-C42 y PID2021-124502OB-C44 (PRESECREL).

Referencias

Abeni, L., Buttazzo, G., Superiore, S., Anna, S., 1998. Integrating multimedia applications in hard real-time systems. Real-Time Systems Symposium doi:10.1109/REAL.1998.739726.

Aghilinasab, H., Ali, W., Yun, H., Pellizzoni, R., 2020. Dynamic memory bandwidth allocation for real-time gpu-based soc platforms. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39, 3348 – 3360. doi:10.1109/TCAD.2020.3012210.

Ali, W., Yun, H., 2018. Protecting real-time gpu kernels on integrated cpu-gpu soc platforms. Leibniz International Proceedings in Informatics, LIPIcs 106. doi:10.4230/LIPIcs.ECRTS.2018.19.

Amert, T., Otterness, N., Yang, M., Anderson, J.H., Smith, F.D., 2018. Gpu scheduling on the nvidia tx2: Hidden details revealed. Real-Time Systems Symposium January, 104–115. doi:10.1109/RTSS.2017.00017.

Andreozzi, M., Gabrielli, G., Venu, B., Travaglini, G., 2022. Industrial challenge 2022: A high-performance real-time case study on arm. Leibniz International Proceedings in Informatics, LIPIcs 231. doi:10.4230/LIPIcs.ECRTS.2022.1.

Ayala-Barbosa, J.A., Mendez-Monroy, P.E., 2022. A new preemptive task scheduling framework for heterogeneous embedded systems. ACM International Conference Proceeding Series , 77–84doi:10.1145/3543712.3543756.

Baek, I., Harding, M., Kanda, A., Choi, K.R., Samii, S., Rajkumar, R.R., 2020. Carss: Client-aware resource sharing and scheduling for heterogeneous applications. IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2020-April, 324–335. doi:10.1109/RTAS48715.2020.00008.

Bakita, J., Anderson, J.H., 2023. Hardware compute partitioning on nvidia gpus*. IEEE Real Time Technology and Applications Symposium (RTAS) , 54–66doi:10.1109/RTAS58335.2023.00012.

Basaran, C., Kang, K.D., 2012. Supporting preemptive task executions and memory copies in gpgpus. 2012 24th Euromicro Conference on Real-Time Systems , 287–296doi:10.1109/ECRTS.2012.15.

Bateni, S., Wang, Z., Zhu, Y., Hu, Y., Liu, C., 2020. Co-optimizing performance and memory footprint via integrated cpu/gpu memory management, an implementation on autonomous driving platform. IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2020-April, 310–323. doi:10.1109/RTAS48715.2020.00007.

Bechtel, M., Yun, H., 2023. Analysis and mitigation of shared resource contention on heterogeneous multicore: An industrial case study. doi:arXiv:2304.13110.

Boniol, F., Mohan, S., 2022. IEEE RTSS 2022 industry challenge. URL: `http://2022.rtss.org/industry-session`.

Calderón, A.J., Kosmidis, L., Nicolas, C.F., Cazorla, F.J., Onaindia, P., 2019. Understanding and exploiting the internals of gpu resource allocation for critical systems. IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD 2019-November. doi:10.1109/ICCAD45719.2019.8942170.

Calderón, A.J., Kosmidis, L., Nicolás, C.F., de Lasala, J., Larrañaga, I., 2021. Assessing and improving the suitability of model-based design for gpu-accelerated railway control systems. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 12800 LNCS, 68–83. doi:10.1007/978-3-030-81682-7_5.

Calderón, A.J., Torres, C., Kosmidis, L., Fernando, C., Ramírez, N., Javier, F., Almeida, C., 2022. Real-Time High-Performance Computing for Embedded Control Systems. doi:10.5821/dissertation-2117-371621.

Capodieci, N., Burgio, P., 2016. Efficient implementation of genetic algorithms on gp-gpu with scheduled persistent cuda threads. International Symposium on Parallel Architectures, Algorithms and Programming, PAAP 2016-January, 6–12. doi:10.1109/PAAP.2015.13.

Capodieci, N., Burgio, P., Cavicchioli, R., Olmedo, I.S., Solieri, M., Bertogna, M., 2022. Real-time requirements for adas platforms featuring shared memory hierarchies. IEEE Design and Test 39, 35–41. doi:10.1109/MDAT.2020.3013828.

Capodieci, N., Cavicchioli, R., Bertogna, M., Paramakuru, A., 2018. Deadline-based scheduling for gpu with preemption support. Real-Time Systems Symposium 2018-December, 119–130. doi:10.1109/RTSS.2018.00021.

Capodieci, N., Cavicchioli, R., Valente, P., Bertogna, M., 2017. Sigamma: Server based integrated gpu arbitration mechanism for memory accesses. ACM International Conference Proceeding Series Part F131837, 48–57. doi:10.1145/3139258.3139270.

Casini, D., Biondi, A., 2022. Placement of chains of real-time tasks on heterogeneous platforms under edf scheduling. Proceedings - 2022 25th Euromicro Conference on Digital System Design, DSD 2022 , 149–156doi:10.1109/DSD57027.2022.00029.

Casini, D., Pazzaglia, P., Biondi, A., Natale, M.D., 2022. Optimized partitioning and priority assignment of real-time applications on heterogeneous platforms with hardware acceleration. Journal of Systems Architecture 124. doi:https://doi.org/10.1016/j.sysarc.2022.102416.

Cavicchioli, R., Capodieci, N., Bertogna, M., 2017. Memory interference characterization between cpucores and integrated gpus in mixed-criticality platforms. IEEE Conference Emerging Technologies and Factory Automation doi:10.1109/ETFA.2017.8247615.

Cavicchioli, R., Capodieci, N., Bertogna, M., 2020. Contending memory in heterogeneous socs:evolution in nvidia tegra embedded platforms. 2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) doi:10.1109/RTCSA50079.2020.9203722.

Chen, G., Zhao, Y., Shen, X., Zhou, H., 2017. Effisha: A software framework for enabling efficient preemptive scheduling of gpu. 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 52, 3–16. doi:10.1145/3018743.3018748.

Cucinotta, T., Amory, A., Ara, G., Paladino, F., Natale, M.D., 2023. Multi-criteria optimization of real-time dags on heterogeneous platforms under p-edf. ACM Transactions on Embedded Computing Systems doi:10.1145/3592609.

Dasari, D., Akesson, B., Nélis, V., Awan, M.A., Petters, S.M., 2013. Identifying the sources of unpredictability in cots-based multicore systems. 8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013) , 39–48doi:10.1109/SIES.2013.6601469.

Diewald, A., Barner, S., Saidi, S., 2019. Combined data transfer response time and mapping exploration in mpsocs. Euromicro Conference on Real-Time System (ECRTS) .

Elliott, G.A., Ward, B.C., Anderson, J.H., 2013. Gpusync: A framework for real-time gpu management. Real-Time Systems Symposium , 33–44doi:10.1109/RTSS.2013.12.

Fang, J., Wang, M., Wei, Z., 2020. A memory scheduling strategy for eliminating memory access interference in heterogeneous system. Journal of Supercomputing 76, 3129–3154. doi:10.1007/s11227-019-03135-7.

Fickenscher, J., Reinhart, S., Hannig, F., Teich, J., Bouzouraa, M.E., 2017. Convoy tracking for adas on embedded gpus. 2017 IEEE Intelligent Vehicles Symposium (IV) , 959–965doi:10.1109/IVS.2017.7995839.

Forsberg, B., Benini, L., Marongiu, A., 2019. Taming data caches for predictable execution on gpu-based socs. 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE) , 650–653doi:10.23919/DATE.2019.8715255.

Forsberg, B., Benini, L., Marongiu, A., 2021. Heprem: A predictable execution model for gpu-based heterogeneous socs. IEEE Transactions on Computers 70, 17–29. doi:10.1109/TC.2020.2980520.

Forsberg, B., Marongiu, A., Benini, L., 2017a. Gpuguard: Towards supporting a predictable execution model for heterogeneous soc. Design, Automation and Test in Europe, DATE 2017 , 318–321doi:10.23919/DATE.2017.7927008.

Forsberg, B., Palossi, D., Marongiu, A., Benini, L., 2017b. Gpu-accelerated real-time path planning and the predictable execution model. Procedia Computer Science 108, 2428–2432. doi:10.1016/j.procs.2017.05.219.

Gupta, K., Stuart, J.A., Owens, J.D., 2012. A study of persistent threads style gpu programming for gpgpu workloads. 2012 Innovative Parallel Computing (InPar) doi:10.1109/InPar.2012.6339596.

Hamann, A., Dasari, D., Wurst, F., Sañudo, I., Capodieci, N., Burgio, P., 2019. Waters industrial challenge 2019 final.

Hartmann, C., Margull, U., 2019. Gpuart - an application-based limited preemptive gpu real-time scheduler for embedded systems. Journal of Systems Architecture 97, 304–319. doi:10.1016/j.sysarc.2018.10.005.

Houdek, P., Sojka, M., Hanzalek, Z., 2017. Towards predictable execution model on arm-based heterogeneous platforms. 2017 IEEE 26th International Symposium on Industrial Electronics (ISIE) , 1297–1302doi:10.1109/ISIE.2017.8001432.

Houssam-Eddine, Z., Capodieci, N., Cavicchioli, R., Lipari, G., Bertogna, M., 2021. The hpc-dag task model for heterogeneous real-time systems. IEEE Transactions on Computers 70, 1747–1761. doi:10.1109/TC.2020.3023169.

Höttger, R., Ki, J., Bui, T.B., Igel, B., Spinczyk, O., 2019. Cpu-gpu response time and mapping analysis for high-performance automotive systems. Euromicro Conference on Real-Time System (ECRTS) .

Jain, S., Baek, I., Wang, S., Rajkumar, R., 2019. Fractional gpus: Software-based compute and memory bandwidth reservation for gpus. IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019-April, 29–41. doi:10.1109/RTAS.2019.00011.

Janzén, J., Black-Schaffer, D., Hugo, A., 2016. Partitioning gpus for improved scalability. 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) , 42–49doi:10.1109/SBAC-PAD.2016.14.

Kang, W., Lee, K., Lee, J., Shin, I., Chwa, H.S., 2021. Lalarand: Flexible layer-by-layer cpu/gpu scheduling for real-time dnn tasks. Proceedings - Real-Time Systems Symposium 2021-December, 329–341. doi:10.1109/RTSS52674.2021.00038.

Kato, S., McThrow, M., Maltzahn, C., Brandt, S., 2012. Gdev: First-class gpu resource management in the operating system. 2012 USENIX Annual Technical Conference (USENIX ATC 12) , 401–412.

Khronos, 2023. Opencl. URL: `https://www.khronos.org/opencl`. (Last accessed 2023).

Kim, H., Patel, P., Wang, S., Rajkumar, R.R., 2018. A server-based approach for predictable gpu access with improved analysis. Journal of Systems Architecture 88, 97–109. doi:10.1016/j.sysarc.2018.05.003.

Kim, H., Rajkumar, R., 2016. Real-time cache management for multi-core virtualization. 13th International Conference on Embedded Software, EMSOFT 2016 doi:10.1145/2968478.2968480.

Kim, S., Jung, C., Kim, Y., 2022. Comparative analysis of gpu stream processing between persistent and non-persistent kernels. 13th International Conference on Information and Communication Technology Convergence (ICTC) 2022-October, 2330–2332. doi:10.1109/ICTC55196.2022.9952789.

Kloda, T., Solieri, M., Mancuso, R., Capodieci, N., Valente, P., Bertogna, M., 2019. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) 2019-April, 1–14. doi:10.1109/RTAS.2019.00009.

Krawczyk, L., Wolff, C., Bazzal, M., Govindarajan, R.P., 2019. An analytical approach for calculating end-to-end response times in autonomous driving applications. Euromicro Conference on Real-Time System (ECRTS) .

Lee, H., Kim, H., Kim, C., Han, H., Seo, E., 2021. Idempotence-based preemptive gpu kernel scheduling for embedded systems. IEEE Transactions on Computers 70, 332–346. doi:10.1109/TC.2020.2988251.

Li, D., Aamodt, T.M., 2016. Inter-core locality aware memory scheduling. IEEE Computer Architecture Letters 15, 25–28. doi:10.1109/LCA.2015.2435709.

Li, R., Hu, T., Jiang, X., Li, L., Xing, W., Deng, Q., Guan, N., 2023. Rosgm: A real-time gpu management framework with plug-in policies for ros 2. IEEE Real Time Technology and Applications Symposium (RTAS) , 93–105doi:10.1109/RTAS58335.2023.00015.

Lim, Y., Kim, H., 2019. Cache-aware real-time virtualization for clustered multi-core platforms. IEEE Access 7, 128628–128640. doi:10.1109/ACCESS.2019.2939859.

Liu, L., Cui, Z., Xing, M., Bao, Y., Chen, M., Wu, C., 2012. A software memory partition approach for eliminating bank-level interference in multicore systems. 2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT) , 367–375.

Lugo, T., Lozano, S., Fernandez, J., Carretero, J., 2022. A survey of techniques for reducing interference in real-time applications on multicore platforms. IEEE Access 10, 21853–21882. doi:10.1109/ACCESS.2022.3151891.

Lumpp, F., Patel, H.D., Bombieri, N., 2021. A framework for optimizing cpu-igpu communication on embedded platforms. 2021 58th ACM/IEEE Design Automation Conference (DAC) 2021-December, 685–690. doi:10.1109/DAC18074.2021.9586304.

Mancuso, R., Pellizzoni, R., Caccamo, M., Sha, L., Yun, H., 2015. Wcet(m) estimation in multi-core systems using single core equivalence. Euromicro Conference on Real-Time Systems 2015-August, 174–183. doi:10.1109/ECRTS.2015.23.

Marchi, M.D., Lumpp, F., Martini, E., Boldo, M., Aldegheri, S., Bombieri, N., 2021. Efficient ros-compliant cpu-igpu communication on embedded platforms. Journal of Low Power Electronics and Applications 11. doi:10.3390/jlpea11020024.

Milluzzi, A., George, A., 2017. Exploration of tmr fault masking with persistent threads on tegra gpu socs. IEEE Aerospace Conference , 1–7doi:10.1109/AERO.2017.7943882.

Nvidia, 2023. Cuda programming guide. URL: `https://docs.nvidia.com`. (Last accessed 2023).

Olmedo, I.S., Capodieci, N., Cavicchioli, R., 2018. A perspective on safety and real-time issues for gpu accelerated adas. IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society 1, 4071–4077. doi:10.1109/IECON.2018.8591540.

Olmedo, I.S., Capodieci, N., Martinez, J.L., Marongiu, A., Bertogna, M., 2020. Dissecting the cuda scheduling hierarchy: A performance and predictability perspective. IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2020-April, 213–225. doi:10.1109/RTAS48715.2020.000-5.

Otterness, N., Miller, V., Yang, M., Anderson, J.H., Smith, F.D., Wang, S., 2016. Gpu sharing for image processing in embedded real-time systems *. 12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2016) .

Otterness, N., Yang, M., Rust, S., Park, E., Anderson, J.H., Smith, F.D., Berg, A., Wang, S., 2017. An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) , 353–364doi:10.1109/RTAS.2017.3.

Park, J., Yeom, H., Son, Y., 2020. Page reusability-based cache partitioning for multi-core systems. IEEE Transactions on Computers 69, 812–818. doi:10.1109/TC.2020.2968066.

Pellizzoni, R., Betti, E., Bak, S., Yao, G., Criswell, J., Caccamo, M., Kegley, R., 2011. A predictable execution model for cots-based embedded systems. 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium , 269–279doi:10.1109/RTAS.2011.33.

Perez-Cerrolaza, J., Abella, J., Kosmidis, L., Calderon, A.J., Cazorla, F., Flores, J.L., 2022. Gpu devices for safety-critical systems: A survey. ACM Computing Surveys 55. doi:10.1145/3549526.

Rehm, F., Dasari, D., Hamann, A., Pressler, M., Ziegenbein, D., Seitter, J., Sañudo, I., Capodieci, N., Burgio, P., Bertogna, M., 2021. Performance modeling of heterogeneous hw platforms. Microprocessors and Microsystems 87. doi:10.1016/j.micpro.2021.104336.

Roeder, J., Rouxel, B., Grelck, C., 2021. Scheduling dags of multi-version multi-phase tasks on heterogeneous real-time systems. 2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC) , 54–61doi:10.1109/MCSoC51149.2021.00016.

Saha, S.K., Xiang, Y., Kim, H., 2019. Stgm: Spatio-temporal gpu management for real-time tasks. 2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) doi:10.1109/RTCSA.2019.8864564.

Schuh, M., Maiza, C., Goossens, J., Raymond, P., Dinechin, B.D.D., 2020. A study of predictable execution models implementation for industrial data-flow applications on a multi-core platform with shared banked memory. Real-Time Systems Symposium 2020-December, 283–295. doi:10.1109/RTSS49844.2020.00034.

Serrano, M.A., Quiñones, E., 2018. Response-time analysis of dag tasks supporting heterogeneous computing. Design Automation & Test in Europe Conference & Exhibition (DATE) Part F137710. doi:10.1145/3195970.3196104.

Singh, J., Olmedo, I.S., Capodieci, N., Marongiu, A., Caccamo, M., 2022. Reconciling qos and concurrency in nvidia gpus via warp-level scheduling. 2022 Design, Automation and Test in Europe Conference and Exhibition , 1275–1280doi:10.23919/DATE54114.2022.9774761.

Spliet, R., Mullins, R.D., 2022. Sim-d: A simd accelerator for hard real-time systems. IEEE Transactions on Computers 71, 851–865. doi:10.1109/TC.2021.3064290.

Suzuki, N., Kim, H., Niz, D.D., Andersson, B., Wrage, L., Klein, M., Rajkumar, R., 2013. Coordinated bank and cache coloring for temporal protection of memory accesses. 16th IEEE International Conference on Computational Science and Engineering, CSE 2013 , 685–692doi:10.1109/CSE.2013.106.

Suzuki, Y., Kato, S., Yamada, H., Kono, K., 2016. Gpuvm: Gpu virtualization at the hypervisor. IEEE Transactions on Computers 65, 2752–2766. doi:10.1109/TC.2015.2506582.

Wu, B., Chen, G., Li, D., Shen, X., Vetter, J., 2015. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. International Conference on Supercomputing 2015-June, 119–130. doi:10.1145/2751205.2751213.

Xu, Y., Wang, R., Li, T., Song, M., Gao, L., Luan, Z., Qian, D., 2016. Scheduling tasks with mixed timing constraints in gpu-powered real-time systems. International Conference on Supercomputing June-2016. doi:10.1145/2925426.2926265.

Yandrofski, T., Chen, J., Otterness, N., Anderson, J.H., Smith, F.D., 2022. Making powerful enemies on nvidia gpus. Real-Time Systems Symposium , 383–395doi:10.1109/RTSS55097.2022.00040.

Yang, M., Otterness, N., Amert, T., Bakita, J., Anderson, J.H., Smith, F.D., 2018. Avoiding pitfalls when using nvidia gpus for real-time tasks in autonomous systems. Leibniz International Proceedings in Informatics, LIPIcs 106, 20:1–20:21. doi:10.4230/LIPIcs.ECRTS.2018.20.

Yao, Y., Liu, S., Wu, S., Wang, J., Ni, J., Yang, G., Zhang, Y., 2022. Wamp2s: Workload-aware gpu performance model based pseudo-preemptive real-time scheduling for the airborne embedded system. IEEE Transactions on Parallel and Distributed Systems 33, 2767–2780. doi:10.1109/TPDS.2021.3134269.

Ye, Y., West, R., Cheng, Z., Li, Y., 2014. Coloris: A dynamic cache partitioning system using page coloring. Parallel Architectures and Compilation Techniques, PACT , 381–392doi:10.1145/2628071.2628104.

Yun, H., Ali, W., Gondi, S., Biswas, S., 2017. Bwlock: A dynamic memory access control framework for soft real-time applications on multicore platforms. IEEE Transactions on Computers 66, 1247–1252. doi:10.1109/TC.2016.2640961.

Yun, H., Mancuso, R., Wu, Z.P., Pellizzoni, R., 2014. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS) doi:10.1109/RTAS.2014.6925999.

Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., Sha, L., 2013. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS) , 55–64doi:10.1109/RTAS.2013.6531079.

Yurtsever, E., Lambert, J., Carballo, A., Takeda, K., 2020. A survey of autonomous driving: Common practices and emerging technologies. IEEE Access 8, 58443–58469. doi:10.1109/ACCESS.2020.2983149.

Zhang, X., Dwarkadas, S., Shen, K., 2009. Towards practical page coloring-basedmulti-core cache management. 4th ACM European conference on Computer systems, Eurosys '09' , 89–102doi:https://doi.org/10.1145/1519065.1519076.

Zhou, H., Bateni, S., Liu, C., 2018. S^3dnn: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) 66, 190–201. doi:10.1109/RTAS.2018.00028.

Zou, A., Li, J., Gill, C.D., Zhang, X., 2023. Rtgpu: Real-time gpu scheduling of hard deadline parallel tasks with fine-grain utilization. IEEE Transactions on Parallel and Distributed Systems 34, 1450–1465. doi:10.1109/TPDS.2023.3235439.