UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"Diseño de un Framework para la planificación de tareas preemptive en sistemas embebidos heterogéneos"

TESIS

QUE PARA OPTAR POR EL GRADO DE:

MAESTRO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

PRESENTA:

José Antonio Ayala Barbosa

DIRECTOR DE TESIS:

Dr. Paul Erick Méndez Monroy

Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas

Ciudad Universitaria, CDMX a Septiembre 2020

Índice general

Índice de Figuras

Índice de Tablas

Resumen

Capítulo 1

Introducción

1.1. Contexto

Para explotar completamente el poder de los sistemas heterogéneos actuales de CPU-GPU se necesitan nuevas herramientas.

1.2. Problema

El problema abordado en esta tesis es el diseño de un framework que permita la ejecución de tareas en modo preemptive en sistemas embebidos heterogéneos ya su implementación es poco relevante actualmente en la literatura, pero la creciente necesidad por su incorporación en estos sistemas brinda la oportunidad de mejorar la aplicabilidad y desarrollo.

1.3. Hipótesis

La hipótesis del presente trabajo es:

> *Es posible diseñar un framework que permita la ejecución de tareas en modo preemptive sobre sistemas embebidos heterogéneos para una mejor administración de sus recursos de cómputo.*

1.4. Aproximación

Mediante el análisis de los elementos inherentes a la estructura de ejecución de procesos híbridos (CPU + GPU) que corren sobre sistemas embebidos heterogéneos, se realiza un diseño de la arquitectura del framework que permite la utilización de tareas preemptive.

1.5. Contribuciones

Tener un planificador que permita la ejecución de tareas preemptive (tareas con privilegio), ya que al implementar puntos con privilegio en planificación dinámica puede disminuir los plazos vencidos de tareas con alta prioridad y mejorar el desempeño del sistema.

El desarrollo en sistemas embebidos heterogéneos actualmente es poco relevante en la literatura, pero la creciente necesidad por su incorporación en el mercado brinda la oportunidad de mejorar su aplicabilidad y desarrollo.

1.6. Estructura de la tesis

Capítulo 2 **Antecedentes**

Capítulo 3 **Trabajo relacionado**

Capítulo 4 **Evaluación de seguridad al implementar patrones de seguridad**

Capítulo 5 **Caso de estudio del método propuesto**

Capítulo 6 **Conclusiones**

Capítulo 2

Antecedentes

El objetivo de este capítulo es introducir los conceptos de: 1. Framework; 2. Sistemas en tiempo real; 3. Tipos de ejecución de tareas; 4. El algoritmo por defecto de los sistemas en tiempo real; 5. Eistemas embebidos heterogéneos; 6. Arquitecturas de hardware y software de tarjetas gráficas; y 7. Cómputo de propósito general en en unidades de procesamiento de gráficos.

2.1. Ingeniería de Software

El Instituto de Ingeniería Eléctrica y Electrónica (Institute of Electrical and Electronics Engineers – IEEE) define a la Ingeniería de Software como:

> *"La Ingeniería de Software[4] es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software; es decir, la aplicacion de la ingeniería al software."*

La Ingeniería de Software aplica diferentes técnicas, normas y métodos que permiten obtener mejores resultados al desarrollar y usar piezas software, al tratar con muchas de las áreas de Ciencias de la Computación es posible llegar a cumplir de manera satisfactoria con los objetivos fundamentales de la Ingeniería de Software. Entre los objetivos de la Ingeniería de Software están[5]:

- Mejorar el diseño de aplicaciones o software de tal modo que se adapten de mejor manera a las necesidades de las organizaciones o finalidades para las cuales fueron creadas.

- Promover mayor calidad al desarrollar aplicaciones complejas.

- Brindar mayor exactitud en los costos de proyectos y tiempo de desarrollo de los mismos.

- Aumentar la eficiencia de los sistemas al introducir procesos que permitan medir mediante normas específicas la calidad del software desarrollado, buscando siempre la mejor calidad posible según las necesidades y resultados que se quieren generar.

- Una mejor organización de equipos de trabajo, en el área de desarrollo y mantenimiento de software.

- Detectar a través de pruebas, posibles mejoras para un mejor funcionamiento del software desarrollado

2.1.1. Framework

Un framework o marco de trabajo es la estructura que se establece para normalizar, controlar y organizar, ya sea, una aplicación completa, o bien, una parte de ella. Esto representa una ventaja para los participantes en el desarrollo del sistema, ya que automatiza procesos y funciones habituales, además agiliza la codificación de ciertos mecanismo ya implementados al reutilizar código. Un framework puede ser considerada como un molde configurable, al que podemos añadirle atributos especiales para finalmente construir una solución completa.

La utilización de un framework siempre conlleva una curva de de aprendizaje, pero a largo plazo facilita la programación, escalabilidad, y el mantenimiento de los sistemas.

2.2. Sistemas en tiempo real

Los sistemas en tiempo real son sistemas de cómputo cuyas tareas deben actuar dentro de limitaciones de tiempo precisas ante eventos en su entorno. Por lo que el comportamiento del sistema depende, no solo del resultado del cálculo, sino también del momento (tiempo) en que se produce [6].

Un sistema en tiempo real debe responder a entradas generadas dentro de un periodo de tiempo especifico para evitar posibles fallas. El deadline o tiempo límite es el momento justo antes en que la tarea debe completar su ejecución. Existen tres tipos de plazos:

- Soft Deadline: En este tipo se pueden superar algunos tiempos límites y el sistema puede aún funcionar correctamente.
- Firm Deadline: Aquí los resultados obtenidos en los plazos vencidos no son útiles, pero los plazos son tolerados frecuentemente.
- Hard Deadline: Si una tarea no se cumple en el tiempo límite, se producirán resultados catastróficos. Este tipo de límites se utilizan comúnmente en tareas que realizan operaciones críticas.

2.2.1. Tipos de tarea

Existen tres tipos de tareas que están presentes en los sistemas en tiempo real:

- Tareas periódicas: Se ejecutan en cada intervalo fijo de tiempo conocido. Normalmente, las tareas periódicas tienen restricciones que indican sus plazos de tiempo.
- Tareas aperiódicas: Se ejecutan aleatoriamente en cualquier plazo de tiempo y no tienen una secuencia de tiempo predefinida.
- Tareas esporádicas: Son una combinación de tareas periódicas y aperiódicas, donde, en tiempo de ejecución actúan como aperiódicas pero la tasa de ejecución es de naturaleza periódica.

La mayoría del tiempo los plazos de tiempo se dan por el tiempo límite de una tarea

2.2.2. Tipos de ejecución de tareas

Existen dos tipo de ejecución de tareas, las ***preemptive***, donde es necesario interrumpir temporalmente una tarea que está realizando un sistema de cómputo, para darle la oportunidad a otra con mayor prioridad, con el compromiso de reanudar la rezagada más adelante, y las ***non-preemptive*** donde se requiere que termine la tarea actual para que posteriormente inicie una con mayor prioridad.

2.2.3. Algoritmos de planificación

Un algoritmo de planificación es una estrategia en la cual un sistema decide ejecutar una tarea en un momento dado, debe garantizar que se asigne el tiempo suficiente a todas las tareas del sistema para que puedan cumplir su tiempo límite en la medida de lo posible.

La planificación en tiempo real se puede dividir en:

- Estática: todas las prioridades se asignan en el momento del diseño del sistema y esas prioridades se mantienen constantes durante el tiempo de vida de una tarea.
- Dinámica: Se las asignan prioridades en tiempo de ejecución, en función de los parámetros de las tareas.

Earliest Deadline First (EDF) es un algoritmo óptimo de planificación para sistemas de tiempo real, y acepta tareas en modo preemptive. Es un algoritmo muy extendido en sistemas en tiempo real debido a su optimalidad teórica en el campo no-preemptive, pero al momento de implementarlo en un planificador preemptive, el resultado puede acarrear un exceso de ejecución si se toma el peor caso [7]. Por ello es necesario buscar alternativas de algoritmos que tengan un mejor desempeño en tareas específicas.

2.3. CPU

La unidad de procesamiento central o CPU es un procesador de propósito general, lo que significa que puede hacer una variedad de cálculos, pero esta diseñado para realizar el procesamiento de información en serie, consta de pocos núcleos de propósito general. Aunque se

pueden utilizar bibliotecas para realizar concurrencia y paralelismo, el hardware *per se* no tiene esa implementación.

2.3.1. Arquitectura del CPU

Un CPU está compuesto principalmente por:

- Reloj: elemento que sincroniza las acciones del CPU.
- ALU (Unidad lógica y aritmética): como su nombre lo indica, soporta pruebas lógicas y cálculos aritméticos, y puede procesar varias instrucciones a la vez.
- Unidad de Control: se encarga de sincronizar los diversos componentes del procesador.
- Registros: memorias de tamaño pequeño, del orden de bytes, y que son lo suficientemente rápidas para que el ALU manipule su contenido en cada ciclo de reloj.
- Unidad de entrada-salida (I/O): soporta la comunicación con las memoria de la computadora y permite el acceso a los periféricos.

2.4. GPU

La unidad de procesamiento gráfico o GPU es un procesador especializado para tareas que requieren de un alto grado de paralelismo. Su uso más extendido es el del procesamiento de instrucciones aplicadas a campo de imágenes 2D y 3D, realizando cálculos con pixeles y texeles[8].

La tarjeta gráfica en su interior puede contener una cantidad de núcleos de un orden de cientos hasta miles de unidades que son más pequeñas y que por ende, individualmente realizan un menor número de operaciones. Esto hace que la GPU esté optimizada para procesar cantidades enormes de datos pero con programas más específicos[2]. Lo más común al utilizar la aceleración por GPU es ejecutar una misma instrucción a múltiples datos para aprovechar su arquitectura.

2.4.1. Arquitectura del GPU

La arquitectura de las tarjetas gráficas ha ido experimentando ciertas evoluciones en su desarrollo para permitir a los programadores hacer un uso más eficiente de su poder de procesamiento. Contienen en su interior componentes no de cómputo no especializado para procesar todo tipo de información.

Una tarjeta gráfica es básicamente un multiprocesador compuesto de una gran cantidad de núcleos de procesamiento que trabajan en paralelo, junto con los componentes de un CPU, las GPU incorporan:

- Memoria: cuentan con diferentes tipos de memoria y principalente compuesta por el tipo DRAM (Memoria dinámica de acceso aleatorio).
  - Memoria global: Almacena los datos enviados desde el CPU.
  - Memoria constante de sólo lectura.
  - Memoria de texturas de sólo lectura.
  - Registros locales por núcleo de 32 bits.

  Donde las memorias constantes y de textura son de acceso más rápidas que la memoria global, ya que actúan como una especie de caché.

- Programación en streams: La arquitectura de una GPU está diseñada con base en la programación de streams, el cual involucra a múltiples cálculos en paralelo para un stream de datos[9].
  - Stream: Conjunto de elementos que tendrán un tratamiento similar.
  - Kernel: Tratamiento aplicado a cada elemento del stream.
  - Thread: Tratamiento ejecutado por procesador aplicado a un elemento del stream.

- Gather y Scatter: Cuando se aplica un kernel a un stream, este aplica todas sus instrucciones a cada elemento, por lo que cada elemento se almacena en un una posición bien definida dentro de la memoria utilizando indices que auxilian a localizarlo, a esta acción se le conoce como Scatter.

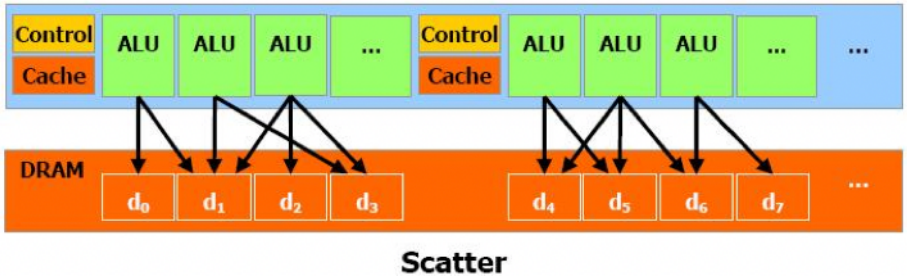

Figura 2.1. *Escritura en DRAM[1].*

En cambio el Gather es la lectura o recolección de un stream en memoria para ser procesado por una unidad de procesamiento.

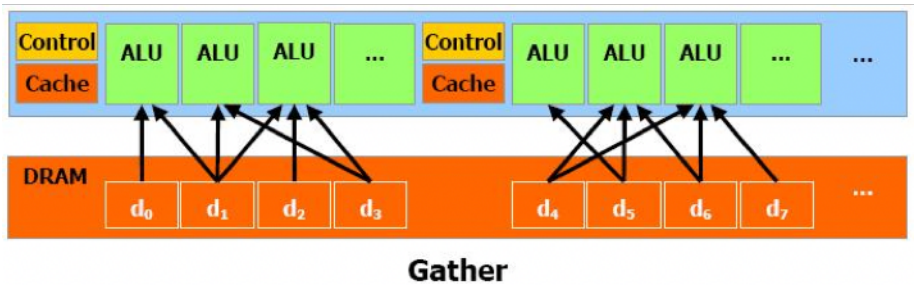

Figura 2.2. *Lectura en DRAM[1].*

2.4.2. Manycore y Multicore

Es necesario destacar que los *manycore* y los *multicore* son utilizados para etiquetar a los CPU y los GPU, pero entre ellos existen diferencias. Un core de CPU es relativamente más potente, está diseñado para realizar un control lógico muy complejo para buscar y optimizar la ejecución secuencial de programas.

En cambio un core de GPU es más ligero y está optimizado para realizar tareas de paralelismo de datos como un control lógico simple enfocándose en la tasa de transferencia (*throughput*) de los programas paralelos.

Con aplicaciones computacionales intensivas, las secciones del programa a menudo muestran una gran cantidad de paralelismo de datos. Las GPU se usan para acelerar la ejecución de esta porción código. Cuando un componente de hardware que está físicamente separado de la CPU y se utiliza para acelerar secciones computacionalmente intensivas de una aplicación, se

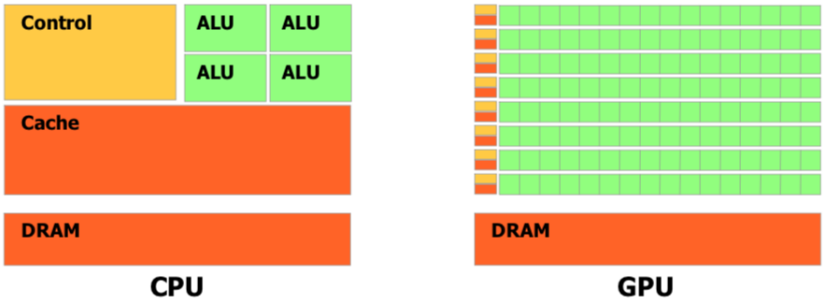

Figura 2.3. *Representación de un CPU y un GPU[1].*

le denomina acelerador de hardware. Se puede decir que las GPU son el ejemplo más común de un acelerador de hardware.

2.4.3. Arquitectura Pascal

2.4.3.1. Memoria unificada

La memoria unificada proporciona un único espacio de direcciones virtuales para la memoria de la CPU y GPU, permitiendo la migración transparente de datos entre los espacios de direcciones virtuales completos tanto de la tarjeta gráfica como del procesador. Esto simplifica la programación en GPUs y su portabilidad ya que no es necesario el preocuparse por administrar el intercambio de datos entre dos sistemas de memoria virtual diferentes[10].

2.4.3.2. Computación preemptive

Permite que las tareas de cómputo se reemplacen con granularidad a nivel de instrucción, en lugar de bloque de subprocesos, evitando el funcionamiento prolongado de aplicaciones que monopolizan el sistema y no dejan ejecutar terceras tareas[10]. Obteniendo así, que las tareas puedan ejecutarse todo el tiempo que requieran ya sea para procesar grandes volúmenes de datos o qué esperen a que ocurran varias condiciones, mientras otras aplicaciones son computadas concurrentemente.

2.4.3.3. Balanceo de carga dinámico

La arquitectura Pascal introdujo el soporte para balanceo de carga dinámico [11], ayudando a la aceleración del cómputo de tareas asíncronas.

En versiones anteriores de las tarjetas, la asignación de recursos en las colas de cálculos y de gráficos debía decidirse antes de la ejecución, por lo que, una vez que se lanzaba la tarea, no era posible reasignarla sobre la marcha. Un problema añadido que existía era, que, si una de las colas se quedaba sin trabajo antes que la otra no podía iniciar un nuevo trabajo hasta que ambas colas terminen completamente[12].

2.4.3.4. Operaciones atómicas

Las operaciones atómicas de memoria frecuentemente son importantes el cómputo de alto rendimiento ya que permiten que los hilos concurrentemente lean, escriban y modifiquen variables compartidas. La arquitectura Pascal nos permite realizar estas operaciones pero ahora con la ventaja de trabajar sobre memoria unificada.

2.4.4. GPGPU

Mientras que las GPU actuales ofrecen una gran potencia de procesamiento, a menudo es difícil aprovecharla. Por ello se han realizado esfuerzos que incluyen nuevos modelos de procesamiento con varios grados de paralelismo.

El computo de proposito general en unidades de procesamiento de graficos o GPGPU es utilizado para acelerar el procesamiento realizado tradicionalmente por la CPU unicamente, donde la GPU actua como un coprocesador que puede aumentar la velocidad del trabajo [13].

La unificación de los espacios de memoria facilita el GPGPU ya que no hay necesidad de transferencias explícitas de memoria entre el host y el dispositivo.

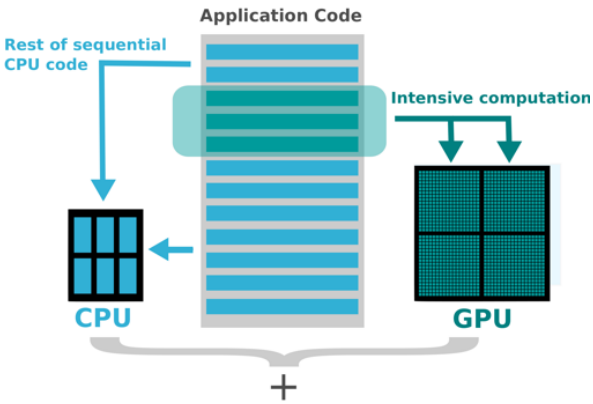

Figura 2.4. *Aceleración de programas en GPUs[2].*

2.5. Sistemas embebidos

Un sistema embebido es un sistema de cómputo diseñado para realizar tareas dedicadas, donde el mayor retos es realizar tareas específicas donde la mayoría de ellas tengan requerimientos de tiempo real [14].

2.5.1. Sistemas embebidos heterogéneos

En los últimos años los sistemas embebidos han ido demandando nuevas características debido a su rápida adopción en el mercado. Con lo que surge el desarrollo de sistemas embebidos heterogéneos, donde está contemplado realizar una gran cantidad de cómputo pero con una gran eficiencia tanto energética como en espacio.

Actualmente la empresa NVIDIA tiene en su catálogo sistemas embebidos heterogéneos con un gran soporte y bibliotecas para el cómputo de alto rendimiento. Dichos sistemas cuentan con la arquitectura Pascal de última generación [15], la cual permite compartir memoria entre CPU y GPU.

Debido a que la mayoría de las GPU en sistemas embebidos no son de naturaleza preemptive, es importante programar los recursos de GPU de manera eficiente en múltiples tareas [16] ya sea de planificación o memoria, lo que permite pensar en un framework que ayude a la administración de sus características.

2.5.1.1. Jetson TX2

Las especificaciones del sistema son:

Elemento	Componentes	Descripción
Arquitectura	NVIDIA Pascal GPU	256 núcleos Optimizados para un mejor rendimiento en sistemas embebidos.
CPU	Dual-Core Denver 2 64-bit CPUs + Quad-Core A57 Complex	Contiene dos clústers de procesamiento, el Denver 2 de 64 bits que se utiliza para tareas pesadas o de un sólo thread; y el ARMv8 Cortex-A57 Complex que actúa en tareas multi-thread y en cargas ligeras.
Memoria	8 GB L128 bit DDR4 Memory	DRAM de 128 bits que da soporte con un gran ancho de banda para una interfaz LPDDR4.
Almacenamiento	32 GB eMMC 5.1 Flash Storage	Integrada en el módulo.
Conectividad	802.11ac Wi-Fi and Bluetooth-Enabled Devices	
Ethernet	10/100/1000 BASE-T Ethernet	
Procesador de señales	1.4Gpix/s Advanced image signal processing Audio Processing Engine	Acelerador por hardware para captura de video y de imágenes. Subsistema que permite el completo soporte de audio multicanal por las diversas interfaces.
Video	Codificador avanzado de video HD Decodificador avanzado de video HD	Permite la grabación de video ultra-high-definition a 60 fps, soporta los estándares H.265 and H.264 BP/MP/HP/MVC, VP9 y VP8. Reproducciónde video ultra-high-definition a 60 fps con pixeles de 12 bits, soporta los estandares H.265, H.264, VP9, VP8 VC-1, MPEG-2, y MPEG-4.
Controlador de la pantalla	eDP/DP/HDMI Multimodal	Realiza un almacenamiento multilinea de pixeles, lo que permite mayor eficiencia de memoria al momento de aplicar operaciones de escalamiento o de búsqueda de pixeles. Permite la reducción del ancho de banda en aplicaciones móviles.

Tabla 2.1. *Especificaciones del sistema Jetson TX2[3].*

2.6. Resumen

Los CPU están diseñados para obtener el máximo rendimiento en un flujo de instrucciones ejecutando las tareas lo más rápido posible, pero un GPU está diseñado para procesar el mayor número de tareas tan rápido como sea posible en un tiempo reducido, por lo que se hace uso el cómputo paralelo en distintos dispositivos.

Capítulo 3

Trabajo Relacionado

Este capítulo presenta los trabajos relacionados con el tema de esta tesis, se analizan 1. Planificación de EDF preemptive limitado de sistemas con tareas esporádicas (*Limited Preemption EDF Scheduling of Sporadic Task Systems*); 2. Planificación de recursos espaciales compartidos con prioridad para unidades de gráficos embebidos (*Priority-driven spatial resource sharing scheduling for embedded graphics processing units*); 3. Framework para planificación en tiempo de ejecución de aplicaciones con manejo de eventos en sistemas embebidos basados en GPU (*Run-Time Scheduling Framework for Event-Driven Applications on a GPU-Based Embedded System*); 4. Sobre planificación dinámica para el GPU, sus aplicaciones en computación gráfica y más (*On Dynamic Scheduling for the GPU and its Applications in Computer Graphics and Beyond*); y 5. REGM: Un modelo de ejecución GPGPU responsivo para soluciones en tiempo de ejecución (*REGM: A Responsive GPGPU Execution Model for Runtime Engines*); 6. Planificación conjunta con GPU y aseguramiento de la memoria intra-nodo (*Intra-Node Memory Safe GPU Co-Scheduling*);

Cada sección presenta lo propuesto en el trabajo relacionado, donde se describe el problema, los objetivos y la solución a éste. Brevemente se describe la solución propuesta con los resultados obtenidos y por último se presentan las conclusiones del trabajo.

3.1. Planificación de EDF preemptive limitado de sistemas con tareas esporádicas

El algoritmo EDF es un algoritmo óptimo de planificación para sistemas de un sólo procesador, por ello está presente en la mayoría de la literatura de sistemas en tiempo real. Este algoritmo acepta tareas en modo preemptive, pero el resultado puede acarrear un exceso de ejecución, por ello siempre se toma el peor caso para la evaluación de las tareas en modo preemptive.

El comportamiento de la memoria caché puede verse significativamente afectado cada vez que se suspenda una tarea e inicie otra con mayor prioridad, lo que resulta en una mayor probabilidad de fallos en memoria. A lo anterior se le suma que las tareas a menudo necesitarán acceder a recursos compartidos, con lo que deben implementarse medidas necesarias para el arbitraje de acceso a estos recursos.

El artículo *"Limited Preemption EDF Scheduling of Sporadic Task Systems"* [14] presenta una técnica de preemption limitado, la cual mejora la implementación de EDF eliminando los puntos preemptive no necesarios para optimizar la capacidad de planificación del sistema. Esta propuesta mantiene la optimalidad teórica de EDF y también reduce la sobrecarga del sistema, manteniendo un número reducido de cambios de contexto. Con estos pequeños cambios se le da oportunidad a las tareas con poco privilegio ejecutarse y tener acceso a la memoria compartida para consumir recursos.

3.2. Planificación de recursos espaciales compartidos con prioridad para unidades de gráficos embebidos

Debido a que la mayoría de las GPU en sistemas embebidos no son de naturaleza preemptive, es importante programar los recursos de GPU de manera eficiente en múltiples tareas [16] ya sea de planificación o memoria, lo que permite pensar en un framework que ayude a la administración de sus características.

El artículo *"Priority-driven spatial resource sharing scheduling for embedded graphics processing units"*[17] presenta una técnica para la ejecución en GPUs llamada *"Planificación de recursos compartidos con reserva de presupuesto"* o por sus siglas en inglés *BR-SRS*, el cual limita el número de núcleos de procesamiento de una GPU para una tarea basándose en su prioridad. Con esto se previene que una tarea que se encuentra en segundo plano retrase a otra que se encuentra en ejecución, también se minimiza la sobre carga de planificación al invocarse solamente dos veces, en el inicio de la tarea y en su finalización.

3.3. Framework para planificación en tiempo de ejecución de aplicaciones con manejo de eventos en sistemas embebidos basados en GPU

Para poder utilizar varias aplicaciones en sistemas en tiempo real complejos es necesario la utilizar técnicas de preemption. Algunos trabajos han utilizado estas técnicas para mejorar el rendimiento de las aplicaciones gráficas en tiempo real, principalmente para la reconstrucción de imágenes en 3D y la detección de rostros.

En el artículo *"Run-Time Scheduling Framework for Event-Driven Applications on a GPU-Based Embedded System"*[18] se propone un Framework de planificación que parte los **kernels** del GPU y genera secuencias de lanzamiento en **subkernels** dinámicamente para entrar el modo preemptive con la implementación de un divisor de carga de trabajo y de un planificador de tareas.

3.4. Sobre planificación dinámica para el GPU, sus aplicaciones en computación gráfica y más

El trabajo *"On Dynamic Scheduling for the GPU and its Applications in Computer Graphics and Beyond"*[19] surgió por la necesidad que se tiene de acelerar la generación de imágenes, asignando más poder de procesamiento a las regiones más importantes de esta.

La ruta que siguió esta investigación se basa en que la naturaleza de las colas concurrentes tiene un fuerte enfoque en la exclusión mutua, que a menudo se considera clave para el rendimiento en sistemas concurrentes. Así que se propone un planificador de tareas basado en colas de trabajo que admiten programación y gestión de la memoria dinámicamente. La solución está centrada en colas concurrentes que recopilan y distribuyen el trabajo siguiendo la regla *FIFO*, el primero en entrar, el primero en salir.

Este Framework permite que las tareas accedan a la cola actual de espera, al mismo tiempo que posibilita el acceso a la estructura por medio del uso de banderas, con lo que aseguramos que el bloqueo sólo se produzca cuando la cola se quede sin elementos o sin espacio.

3.5. REGM: Un modelo de ejecución GPGPU responsivo para soluciones en tiempo de ejecución

Por la naturaleza non-preemptive asociada al copiado y transferencia de datos entre el host y el device hace necesario el administrar la ejecución de los recursos para proteger los tiempos de respuesta de tareas con alta prioridad.

El artículo *"REGM: A Responsive GPGPU Execution Model for Runtime Engines"*[**?**] presenta como solución el dividir las transacciones de copiado de memoria en varios fragmentos para insertar puntos preemptive. Esto también garantiza que sólo las tareas de mayor prioridad se ejecuten en el device en cualquier momento, y así evitar interferencias de rendimiento causadas por lanzamientos concurrentes.

La mayoría de los Frameworks actualmente acarrean ciertas limitaciones en su configuración para sistemas en tiempo real, principalmente atribuidas al hecho de que las tareas requieren realizar copias de memoria entre el device y el host para efectuar sus cálculos, esta transacción es realizada por el acceso directo a memoria (DMA), y éste al ser non-preemptive puede bloquear otros accesos con una mayor prioridad hasta que se termine la tarea actual, obteniéndose un mayor tiempo de bloqueo y un cuello de botella en la tasa de transferencia de datos.

La principal característica de RGEM es dividir las transacciones de copiado de memoria en varios fragmentos para insertar puntos preemptive que permitan limitar la ejecución de las tareas. Posteriormente lanza los kernels de diferentes tareas basandose en su prioridad, previniendo interferencias de las tareas de mayor prioridad con las que se encuentran actualmente en ejecución.

3.6. Planificación conjunta con GPU y aseguramiento de la memoria intra-nodo

El artículo *Ïntra-Node Memory Safe GPU Co-Scheduling"*[20] propone la creación del framework schedGPU, el cual utiliza el algoritmo de planificación Slurm. El marco de trabajo administra de forma segura las múltiples solicitudes de aplicaciones para acceder a las GPU al garantizar que no se produzcan sobrecargas de memoria durante la ejecución de la tarea. Este acceso es controlado mediante bloqueos de archivos, señales del sistema y exclusión mutua.

SchedGPU utiliza el patrón de diseño cliente-servidor ya que toma cada tarea que busca ser lanzada en el GPU como un cliente que está solicitando memoria a un Servidor centralizado (en el mismo nodo), el cual permite que se ejecute si hay suficiente memoria, o en caso contrario la bloquea hasta que se encuentre memoria necesaria para su funcionamiento. El servidor crea un nuevo hilo para cada cliente y mantiene una visión global de la memoria utilizada por todos los clientes a través de la biblioteca de administración de NVIDIA (**NVML**), esto para evitar la creación de un nuevo contexto que consuma memoria.

La tarea es modificada únicamente al llamar explícitamente las funciones de la biblioteca del cliente para previamente asignar la memoria requerida al GPU. Aunque esto acarrea una gran desventaja al considerar tareas donde no siempre es posible conocer la memoria requerida total de GPU, esto porque la memoria de la GPU se asigna en tiempo de ejecución. En el caso en que dos o más tareas se ejecuten al mismo tiempo y ambas aumenten gradualmente el uso de la memoria del GPU, se puede llegar a utilizar completamente la memoria disponible, con lo que podrán requerir más tiempo para completar la ejecución o directamente lanzar

un error en tiempo de ejecución.

3.7. Resumen

The earliest deadline first (EDF) scheduling algorithm is a typical representative of the dynamic priority scheduling algorithm. However, once the system is overloaded, the deadline miss rate increases and the scheduling performance deteriorates sharply, which causes a reduction in system resource utilization.

En la práctica, ambas visiones de planificación, tanto preemptive, como non-preemptive, tienen ventajas y desventajas comparadas entre sí, por lo que ninguna es superior a la otra. Pero el patrón encontrado es que es necesario en pensar en un Framework qué brinde ayuda a la ejecución de tareas y que permita guardar el contexto en un tiempo específico.

Hoy en día, los sistemas embebidos basados en GPU han empezado a considerarse esenciales debido a su alta programabilidad y capacidad de desarrollo con técnicas de alto rendimiento, sumado a su bajo consumo energético. Estos exigen una mayor potencia de cálculo y deben responder a muchos eventos, por lo que se han buscado estrategias, y ahora comparten la memoria entre el CPU y el GPU, lo que resulta en una latencia muy cercana a cero.

Se han propuesto diversos frameworks de última generación para planificación de tareas para aprovechar el rendimiento de los sistemas embebidos basados en GPU y su bajo consumo de energía.

Anexos

.1. Glosario de términos

- **Preemptive**: Tarea con privilegio.
- **Preemption**: El hecho de ser preemtive.
- **Throughput**:Tasa de transferencia.
- **Framework**: Marco de trabajo.
- **Pixel (Picture Element)**: (Elemento de imagen) Unidad mínima homogénea de color que forma una imagen.
- **Texel (Texture Element)**: (Elemento de textura) Unidad mínima de una textura aplicada a una superficie.
- **Deadline**: Tiempo límite, es el momento justo antes en que una tarea debe completar su ejecución

Bibliografía

[1] NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide*, version 2.0 ed., NVIDIA, Julio 2008.

[2] ——, "Computación acelerada: Supera los desafíos más importantes del mundo." [Online]. Available: https://la.nvidia.com/object/what-is-gpu-computing-la.html

[3] ——, "Jetson tx2 developer kit." [Online]. Available: https://developer.nvidia.com/embedded/jetson-tx2-developer-kit

[4] "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 1–84, Dec 1990.

[5] S. SÁNCHEZ ALONSO, M. Á. SICILIA URBAN, and D. RODRIGUEZ GARCIA, *Ingeniería del software - un enfoque desde la guía swebok.* Alfaomega Grupo Editor, S.A. de C.V, 2012.

[6] G. C. Butazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications.* Springer Science Business Media, 2011.

[7] S. Heath, *Embedded systems design.* EDN Series For Design Engineers, 2003.

[8] A. STANCU, E. CODRES, and M. M. Guerrero, *Jetson TX2 and CUDA Programming*, second edition ed., NVIDIA, 2018.

[9] S. Rennich, "Cuda c/c++ streams and concurrency," NVIDIA, 2011. [Online]. Available: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

[10] NVIDIA, *NVIDIA Tesla P100 The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU*, NVIDIA Corporation, 2016.

[11] J. P. HURTADO V., "Análisis a fondo: Arquitectura gpu nvidia pascal – diseñada para la velocidad," 2016. [Online]. Available: https://www.ozeros.com/2016/05/analisis-a-fondo-arquitectura-gpu-nvidia-pascal-disenada-para-la-velocidad/

[12] R. Smith, "The nvidia geforce gtx 1080 gtx 1070 founders editions review: Kicking off the finfet generation." [Online]. Available: https://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/9

[13] NVIDIA, "Nvidia sobre la computación de gpu y la diferencia entre gpu y cpu," 2018. [Online]. Available: https://la.nvidia.com/object/what-is-gpu-computing-la.html

[14] M. Bertogna and S. Baruah, "Limited preemption edf scheduling of sporadic task systems," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 579–591, 2010.

[15] C. Hartmann and U. Margull, "Gpuart - an application-based limited preemptive gpu real-time scheduler for embedded systems," *Journal of Systems Architecture*, 2018.

[16] NVIDIA, "Nvidia jetson tx2: High performance ai at the edge." [Online]. Available: www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/

[17] Y. Kang, W. Joo, S. Lee, and D. Shin, "Priority-driven spatial resource sharing scheduling for embedded graphics processing units," *Journal of Systems Architecture*, vol. 76, pp. 17–27, mayo 2017.

[18] H. Lee and M. A. A. Faruque, "Run-time scheduling framework for event-driven applications on a gpu-based embedded system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1956–1967, 2016.

[19] M. Steinberger, "On dynamic scheduling for the gpu and its applications in computer graphics and beyond," *IEEE Computer Graphics and Applications*, vol. 38, no. 3, pp. 119–130, 2018.

[20] C. Reaño, F. Silla, D. S. Nikolopoulos, and B. Varghese, "Intra-node memory safe gpu co-scheduling," *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, vol. 29, no. 5, 2018.