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

Aqui va la introducción

1.1. Problema y Oportunidad

En la mayoría de las organizaciones que emplean sistemas en tiempo real hay una creciente necesidad por la adopción de una tecnología más flexible como lo es la incorporación de sistemas con tareas preemptive.

Aunque dicha implementación es poco investigada actualmente en la literatura, este trabajo de tesis nos brinda la oportunidad de idear las bases de un framework que facilite el diseño y/o desarrollos de aplicaciones en tiempo real, y específicamente, con tareas preemptive.

1.2. 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.*

1.3. Aproximación

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

1.4. Contribuciones

Tener las bases del diseño de un framework que permita planificar la ejecución de tareas preemptive, ya que al implementar puntos preemptive en planificación estática y dinámica se pueden disminuir los plazos vencidos de tareas con alta prioridad y mejorar el desempeño general del sistema.

1.5. Estructura de la tesis

El presente trabajo se estructura en cinco capítulos, en el capítulo **Antecedentes**, se da una introducción a los conceptos que forman partes del marco teórico, y que son necesarios para entender el contexto en el que se desenvuelve el trabajo. En el capítulo **Trabajo Relacionado**, se da un breve resumen sobre los textos que contienen información pertinente del estado del arte del tema. Posteriormente, se encuentra el capítulo **Diseño del framework** en donde se describe puntualmente la propuesta de solución. Finalmente, en el capítulo **Conclusiones y Trabajo Futuro** se recapitulan los alcances del trabajo y se mencionan los puntos que se dejaron para un trabajo futuro,

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. Sstemas embebidos heterogéneos; 6. Arquitecturas de hardware y software de tarjetas gráficas; y 7. Cómputo de propósito general 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[6] 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[7]:

- 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 qué se produce [8].

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 intervalos de tiempo se dan por el plazo límite de una tarea.

2.2.2. Esquemas de planificación

2.2.2.1. Planificación cooperativa

En en el esquema de planificación ***cooperativa*** o ***non-preemptive*** mostrado en la figura 2.1, el planificador asigna las tareas a los nodos de procesamiento disponibles y estas ocupan los recursos de cómputo durante todo su ciclo de vida.

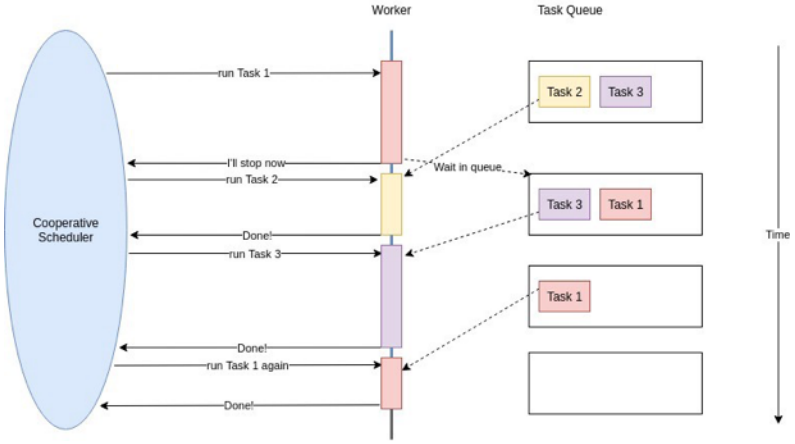

Figura 2.1. *Planificación cooperativa.[1]*

Este esquema de planificación es sencillo de implementar ya que las tareas se ejecutarán de manera secuencial, y se implementa cuando se tiene un uso predecible de los tiempos de ejecución de todo el sistema. Pero, como observamos en la figura 2.2, si una tarea ocupa los recursos un tiempo superior al contemplado no se puede interrumpir y puede generar plazos vencidos en las demás tareas.

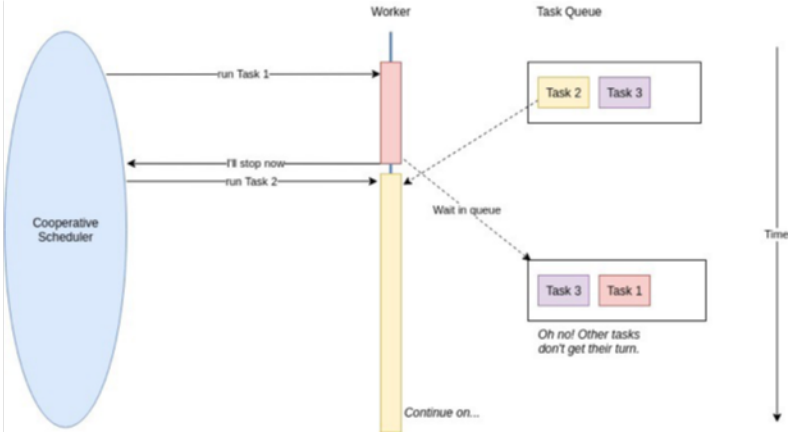

Figura 2.2. *Planificación cooperativa con plazos vencidos.[1]*

2.2.2.2. Planificación preemptive

En este esquema el planificador asigna las tareas a los recursos disponibles, y les define un tiempo de ejecución máximo, comúnmente llamado quantum[9]. Superado este punto, el planificador interrumpe la tarea para que otra sea ejecutada en su lugar, y la tarea interrumpida debe esperar hasta que le toque su turno nuevamente. Un ejemplo de este esquema, lo tenemos en la figura 2.3.

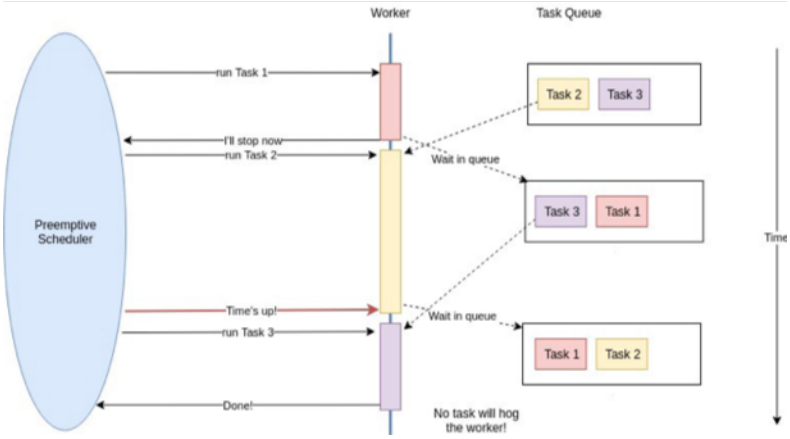

Figura 2.3. *Planificación preemptive.[1]*

La mayor diferencia entre la planificación cooperativa y la preemptive es que la primera debe ejecutar la tarea de principio a fin, y la segunda puede interrumpir las tareas si así se requiere.

Debido a que muchas veces se interrumpen tareas a la mitad de un proceso, es necesario almacenar y restaurar el contexto que se tenia antes de dicha interrupción para continuar justo en el punto en donde se quedó. Este proceso de almacenamiento, intercambio y restauración del contexto de las tareas se denomina **cambio de contexto** (**context switch**).

Existen diversas clasificaciones en que se pueden agrupar las soluciones para dar soporte a la planificación de tareas preemptive, una de ellas la podemos encontrar en el artículo [10], el cual muestra dos tipos de implementaciones:

- ***Basado en Hardware.*** Aquí se utilizan dispositivos de e/s para el cambio o drenado del contexto para implementar las políticas preemptive.
- ***Basado en Software.***

- ***Partición de Kernel.*** Aquí los kernels grandes son partidos en subkernels dividiendo sus grids en fragmentos más pequeños. Es decir, en vez de lanzar todos los Threads por Bloque (TB) de un kernel a la vez, sólo se van lanzando fracciones de éste a la vez. Este enfoque es útil para núcleos con muchos TB, donde cada TB tienen un tiempo de ejecución corto. Sin embargo, para aquellos con tiempos de ejecución largos, se pueden generar retrasos significativos y posiblemente tiempos en que haya inversión de prioridad, lo que no es aceptable en sistemas en tiempo real. Aunado a esto, este método no es aplicable a los kernels que requieren una sincronización global entre TB, ya que el lanzamiento paulatino de los componentes de un grid podrá causar plazos vencidos en las barreras entre cada fragmento.
- ***Partición en Tareas.*** Esta técnica es parecida a la anterior, pero los fragmentos se ven como tareas, se invocan tantos TB como sea posible tenerlos activos al mismo tiempo. Aquí la GPU y CPU comparten una variable que se utiliza para señalizar las solicitudes de cambio de contexto entre los threads activos y los detenidos.
- ***Entorno de Scripts.*** Esta técnica permite manejar automáticamente los kernels dependiendo de ciertos parámetros o puntos de control, liberando al programador de realizarlo manualmente. Esta aproximación funciona especialmente para entornos con kernels pequeños, ya que para aquellos que tienen una ejecución larga es necesario cuidar el nivel de granularidad o podría llegarse a plazos vencidos.

Otra clasificación la encontramos en el trabajo [11] que agrupa en tres tipos de algoritmos de planificación en tiempo real.

- ***Colas masivas en paralelo.*** Este apartado está centrado en, ya sea una o varias colas concurrentes que recopilan y distribuyen el trabajo siguiendo la regla *FIFO*, el primero en entrar, el primero en salir.
- ***Administración dinámica de memora.*** Se tiene un administrador de memoria que verifica si es posible asignar memoria para una nueva tarea, o cuál de las que

actualmente en ejecución ha superado su espacio definido.

- ***Planificación por prioridad.*** Se utilizan algoritmos de planificación para manejar dinámicamente el cambio de prioridades y maximizar el rendimiento del sistema. La mayoría de las veces este tipo es el más cercano a una implementación completa de tiempo real.

En esta clasificación también debemos agregar otro apartado:

- ***Administración dinámica de los núcleos de procesamiento.*** Aquí se limita el número de núcleos de procesamiento en tiempo de ejecución de una GPU para una tarea, y el número depende casi siempre de la prioridad de la tarea.

Ya que, aunque no esta originalmente contemplado, es necesario discriminar entre la administración de la memoria y la administración de los núcleos que una GPU nos va a prestar para realizar el procesamiento.

Finalmente, tenemos una clasificación de tipos de planificación que va más enfocada a la forma en que es implementada en el código.

- ***Modificación de código fuente.*** Es necesario que el programador modifique el código del kernel para implementar cada una de las acciones que va a seguir la tarea, desde su inicio, pasando por su interrupción, y hasta su finalización.
- ***Modificación del API.*** En este apartado, se hace una modificación a nivel de las bibliotecas o el compilador, la ventaja es que la aplicación no es modificada manualmente, pero su utilización muchas veces no está permitida por los administradores.

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. Su objetivo es adaptarse al progreso del sistema para buscar la configuración óptima de planificación.

Algoritmo	**Asignación de prioridad**	**Criterio de planificación**	**Preemptive/ Non-preemptive**	**Utilización de CPU**	**Eficiencia**
SJF	Estática	Tiempo de Ejecución	Non-preemptive	100 %	Eficiente con tareas de finalización oportuna
EDF	Dinámica	Plazo Límite	Preemptive	100 %	Eficiente en condiciones subcargadas
RM	Estática	Periodo	Preemptive	$< 100\,\%$	Eficiente en condiciones sobrecargadas
DM	Estática	Plazo Límite Relativo	Preemptive	$>$ a RM	Eficiente
LLF	Dinámica	Laxitud	Preemptive	100 %	Eficiente
GEDF	Dinámica	Plazo Límite y Tiempo de ejecución	Non-preemptive	100 %	Eficiente en ambientes Non-preemptive

Tabla 2.1. *Matriz de comparación de algoritmos de planificación.*

2.2.3.1. Shortest Job First

Shortest Job First (SJF) es el algoritmo de planificación que asigna la prioridad mayor a la tarea con el menor tiempo de ejecución. SJF es el algoritmo más utilizado cuando se comienzan a estudiar los sistemas en tiempo real debido a su simplicidad y porque minimiza la cantidad promedio de tiempo que cada tarea debe esperar hasta que se complete su ejecución [12]. Este algoritmo funciona únicamente con tareas non-preemptive, por lo que fácilmente puede llegarse a un estado de inanición de tareas que requieren mucho tiempo para completarse si se agregan continuamente tareas pequeñas.

2.2.3.2. Earliest Deadline First

Earliest Deadline First (EDF) es un algoritmo con prioridad dinámica, en el que la tarea con el plazo fijo más próximo tiene la máxima prioridad. Este algoritmo es óptimo para

implementación sobre un único procesador, y cuando el sistema se encuentra en bajos y moderados niveles de contención de recursos y datos[13]. Ya que cuando se sobrecarga el sistema, la mayoría de las tareas obtienen una alta prioridad, lo que termina en un rendimiento disminuido.

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 [14]. Por ello es necesario buscar alternativas de algoritmos que tengan un mejor desempeño en tareas específicas.

2.2.3.3. Rate Monotonic

Rate Monotonic (RM) es un algoritmo de planificación preemptive con prioridad estática para un solo procesador[13]. RM asigna la prioridad más alta a la tarea con el periodo más corto, suponiendo que los periodos sean igual a los plazos ($P_i = D_i$), esto porque si la tasa de demanda es mayor, el periodo sería más corto y por ende, la prioridad aumentaría. Por ello es optimo para usarse en tareas periódicas. La mayor limitación de su implementación, es que al utilizar tareas de prioridad fija no siempre se utiliza el 100 % del CPU, lo que conlleva al posible desperdicio de recursos[15].

2.2.3.4. Deadline Monotonic

Deadline Monotonic (DM) es el algoritmo óptimo de planificación con prioridad fija donde las prioridades son asignadas inversamente proporcionales a los plazos fijos, con esto cuando se cumple que el plazo es menor al tiempo de la tarea (D $<$ T) cuando el periodo es igual que el plazo limite (P = D) podemos ver a RM como un caso especial de DM [16]. DM ejecuta en cada instante de tiempo la tarea con el plazo más corto, por lo que si dos más tareas tienen el mismo plazo limite se debe elegir aleatoriamente la siguiente en ejecutarse.

2.2.3.5. Least Laxity First

Least Laxity First (LLF) es un algoritmo óptimo de planificación con prioridad dinámica. La laxitud de una tarea está definida como el plazo límite menos el tiempo de ejecución restante, está laxitud es la cantidad máxima de tiempo que un trabajo puede esperar cumpliendo su plazo límite. En este algoritmo, se otorga la máxima prioridad al trabajo con la menor laxitud, se permite que la tarea actualmente en ejecución sea intercambiada por otra con menor laxitud en cualquier momento [16].

El punto débil de este algoritmo se presenta cuando dos tareas presentan la misma laxitud, ya que un proceso se ejecutará durante un período corto de tiempo y luego será reemplazado por el otro y viceversa, obteniendo numerosos cambios de contexto durante la vida útil de las tareas mermando el rendimiento del sistema en general. Este algoritmo es óptimo para sistemas con tareas periódicas [17].

2.2.3.6. Group Earliest Deadline First

Group Earliest Deadline First (GEDF) está pensado para mejorar el desempeño de EDF durante condiciones de sobrecarga. La idea principal de su funcionamiento es agrupar las tareas con plazo límite similares y dentro de cada grupo planificar las tareas con SJF [17]. El parámetro rango de grupo (Gr) determina qué tarea ingresa a qué grupo el cuál es un porcentaje de la tarea al comienzo del plazo absoluto de cada cola.

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[18].

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[3]. 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[19].
  - 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.

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

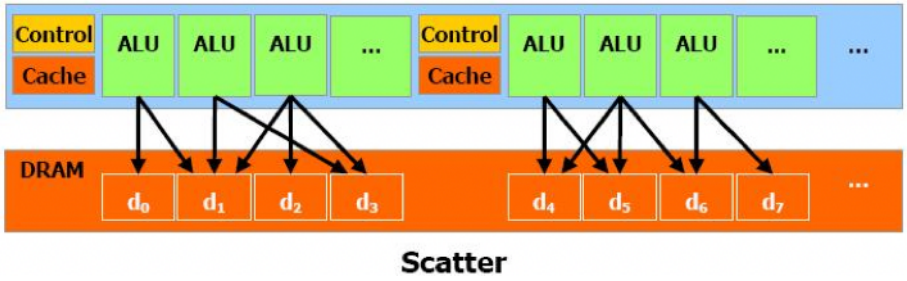

Figura 2.4. *Escritura en DRAM[2].*

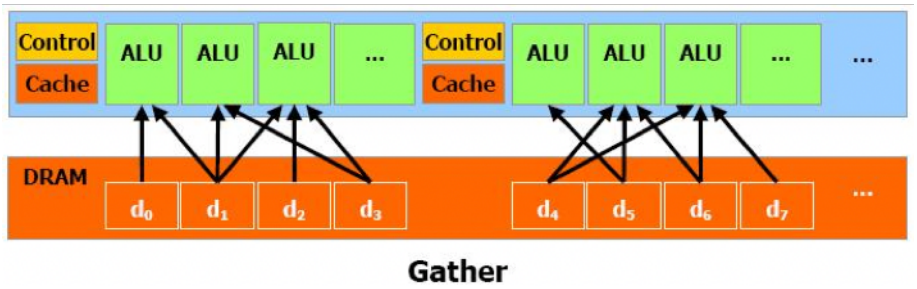

Figura 2.5. *Lectura en DRAM[2].*

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.

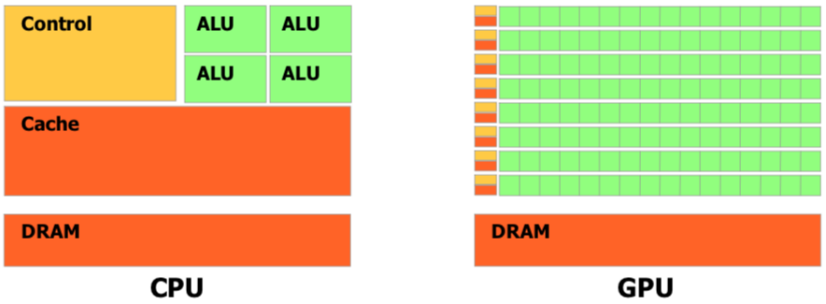

Figura 2.6. *Representación de un CPU y un GPU[2].*

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 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

La principal ventaja de la arquitectura Pascal está en su construcción ya que está implementada con transistores FinFET[20], los cuales a ser de un tamaño de 16 nanometros, permiten tener un tamaño reducido, proporcionar un rendimiento alto y obtener una gran eficiencia energética. Dicha combinación la hacen ideal para ser implementada en dispositivos embebidos que requieran ejecutar tareas híbridas.

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[21].

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[21]. 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 [22], 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[23].

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 [24].

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.

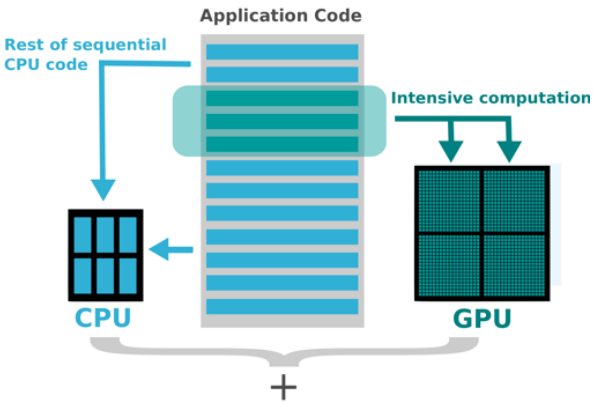

Figura 2.7. *Aceleración de programas en GPUs[3].*

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 [25].

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, dónde 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 [10], 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 [26] ya sea de planificación o memoria, lo que permite pensar en un framework que ayude a la administración de sus características.

2.6. Material de trabajo

Para realizar la presente tesis, se tuvo acceso al sistema embebido heterogéneo NVIDIA Jetson TX2, en el cual se realizaron algunas pruebas para la familiarización con esté tipo de dispositivos, así como la programación en tarjetas gráficas.

En la figura 2.8 se muestra diagrama de bloques de la arquitectura del sistema Jetson TX2.

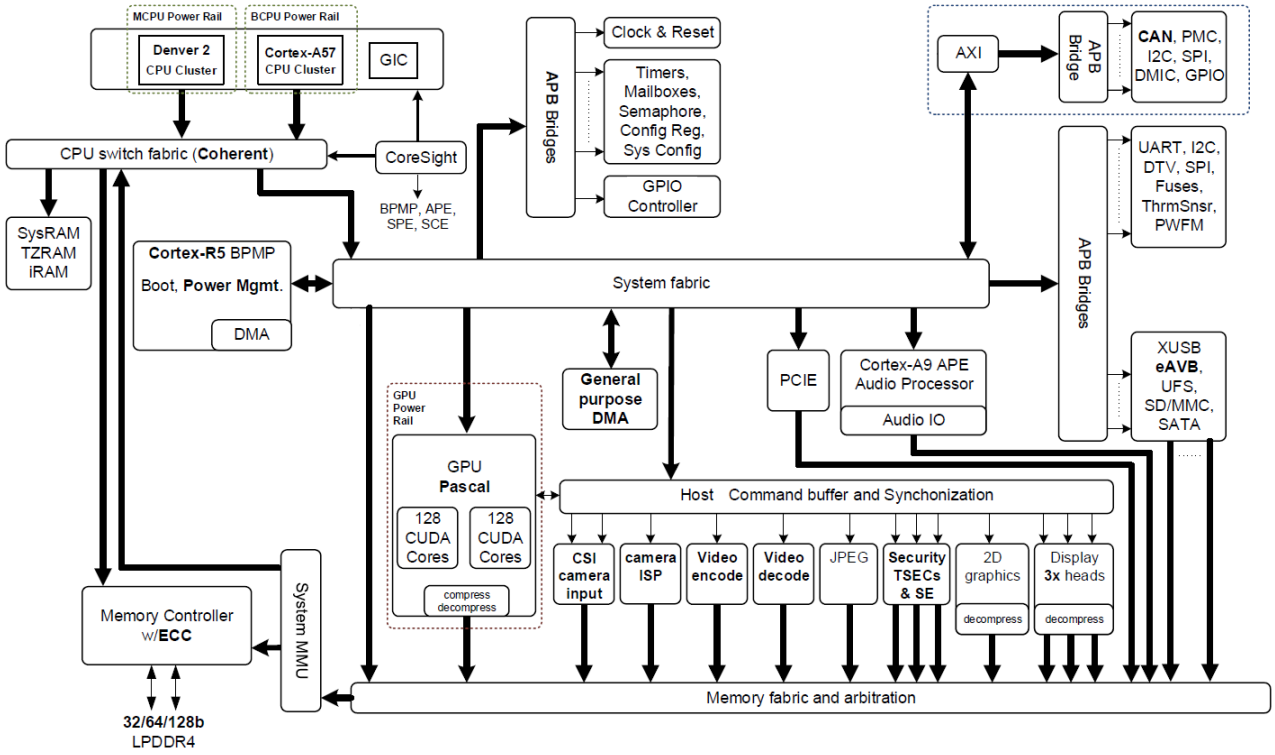

Figura 2.8. *Diagrama de la arquitectura del sistema Jetson TX2[4].*

2.6.1. Jetson TX2

Las especificaciones del sistema están descritas en la tabla 2.2.

Algunas de las tareas realizadas con el dispositivo incluyen desde la familiarización hasta la puesta a punto, como son:

- Instalación del Sistema Operativo Ubuntu 18 para procesadores ARM.
- Instalación de CUDA manager.

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 estándares H.265, H.264, VP9, VP8 VC-1, MPEG-2, y MPEG-4.
Controlador de la pantalla	eDP/DP/HDMI Multimodal	Realiza un almacenamiento multilínea 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 2.2. *Especificaciones del sistema Jetson TX2[5].*

- Actualización de bibliotecas compatibles.
- Configuración de área local y conexión a travez de computadora remota.
- Investigación e implementación de ejercicios de GPGPU.
- Realización y modificación de ejercicios para la familiarización con la arquitectura Pascal, estructura de la tarjeta y su memoria.

2.7. Resumen

mathematics 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.

2.7.1. Computación preemptive

La mayoría de los sistemas operativos modernos utilizan el cómputo preemption para planificar tareas, en una computadora que no utiliza tareas preemptive sólo se puede ejecutar un proceso a la vez con tofos los demás esperando en una cola hasta que se complete el proceso actual en ejecución. Por otra parte, una planificación de tareas preemptive elige un proceso y lo deja ejecutar durante un tiempo máximo llamado cuanto[9], al llegar ese momento, se suspende y el planificador escoge otro dependiendo de las prioridades dadas o del algoritmo.

2.7.2. 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.

Capítulo 3

Trabajo Relacionado

3.1. Clasificación de la planificación de tareas

La clasificación planificación de tareas preemptive esta compuesta de diversos tipos dependiendo de sus técnicas de implementación, cómo se describe en la sección 2.2.2.2. Las soluciones basadas en hardware son costosas, ya que debemos desarrollar y construir un dispositivo que auxilie con el cambio de contexto, por ejemplo, en el artículo [27] se utilizan extensiones de hardware a modo de registros que almacenan el contexto y en general las direcciones de memoria que contienen la información necesaria para la restauración de la ejecución de un kernel. En [28] se propone la utilización de extensiones de hardware mediante el intercambio equitativo de recursos entre los núcleos de procesamiento, esto realizando un cambio de contexto aplicando el modo preemptive en el espacio de procesamiento. En lugar de intercambiar el contexto de todo el grid, se pretende intercambiar suficientes TB de un kernel en ejecución para que haya suficientes recursos disponibles para despachar la nueva tarea. Otra solución la observamos en 8, en donde se de desarrolló un compilador y que emplea una extensión de hardware para reducir la latencia al implementar el modo preemptive, el compilador inserta puntos preemptive utilizando un análisis del ciclo de vida de los registros. Se utiliza una lógica de compresión descompresión para disminuir el tamaño del contexto de una tarea. Es decir, cuando el valor almacenado en un determinado registro es siempre igual a lo largo de la ejecución de los TB de un kernel, sólo se guardará un valor

durante el cambio de contexto.

Para aquellas basadas en software,

tenemos por una parte las que parten el kernel como

Para poder utilizar varias aplicaciones en sistemas en tiempo real complejos es necesario la utilizar técnicas de implementación preemptive. 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 [33] 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. Utiliza un divisor de carga de trabajo que divide el núcleo de la GPU en múltiples subkernels durante el tiempo de ejecución para implementar el modo preemptive. Dependiendo del estado actual de sistema y de la prioridad, el divisor de carga de trabajo decide el número y el tamaño de cada subkernel.

Se cuenta con un generador de ejecución planificada, el cual, dependiendo del estado actual de uso de los recursos del sistema y del plazo límite del la tarea, lanza una secuencia de tareas para maximizar el numero de aplicaciones cercanas al su plazo vencido.

[29] describe el framework EffiSha que se basa en un entorno de scripts que permite convertir los kernels automáticamente a modo preemptive, como se muestran en los resultados, esta solución funciona bien para kernels con ejecución pequeña, por que al tener en el sistema aquellos que salen de la media, la granularidad del TB limita el retraso mínimo preemptive que se puede lograr, resultando muy seguramente en plazos vencidos.

Es importante mencionar que el artículo [34] es el primer trabajo que genera un framework para utilizar tareas en tiempo real en tarjetas gráficas. Este trabajo entra dentro de la categoría de colas masivas en paralelo, ya que se basa en la partición en fragmentos de memoria a procesar, cada fragmento es agregado a una cola de procesamiento para ser ejecutado. Su solución es 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 GPU en cualquier momento, y así evitar interferencias de rendimiento causadas por lanzamientos concurrentes.

La primer característica de este framework es que se basa en transacciones de datos preemptive, por lo que los tiempos de bloqueo están limitados al tiempo limitado para copiar cada fragmento de dato. La segunda característica es que permite lanzar los kernels de diferentes tareas una por una basadas en su prioridad, lo que evita que las tareas con alta prioridad sean interferidas por la carga simultánea de trabajo una vez iniciadas. Sin embargo el lanzamiento del kernel puede bloquearse al haber un kernel de menor prioridad lanzado anteriormente, esto debido al probable alto uso de memoria global.

El artículo [31] propone la creación del framework schedGPU, el cual utiliza el administrador de trabajo Slurm para planificar las tareas. Este framework administra las múltiples solicitudes para acceder a las GPU de forma segura al garantizar que no se produzcan sobrecargas de memoria durante su ejecución. 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.

El artículo [32] 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*, la cual limita el número de núcleos de procesamiento de una GPU para una tarea basándose en su prioridad, esto lo realiza modificando las bibliotecas de OpenGL-ES. Así se previene que una tarea que se encuentra en segundo plano retrase a otra que se encuentra en ejecución, también se minimiza la sobrecarga de planificación al invocarse solamente dos veces, en el inicio de la tarea y en su finalización.

El articulo [30] implementa una serie de funciones para realizar particiones de kernel y de datos, esto realizando subkernels y dividiendo las transacciones de datos en fragmentos. Específicamente, se presenta una técnica de reescritura binaria para reconfigurar de manera transparente el código de los kernel. Mientras que para los kernel un poco más complejos, se desarrolló una técnica de transformación fuente a fuente que compila el código del kernel transformado en binarios CUDA. La prioridad de las tareas esta dada por colas de ejecución. GPES modifica el API de CUDA utilizando las bibliotecas de openCUDA para reconfigurar el código binario de los kernels, esto lo realiza obteniendo un máximo de blocks que se pueden ejecutar por quantum. Para realizar esto, se ayuda de dos implementaciones.

1. *Transformación fuente a fuente.* Para dar soporte al cómputo paralelo, el hardware mantiene indices continuos en un grid (*blockIdx*), por ejemplo, si un grid consta de 256 blocks, tiene indices desde 0 a 255. Entonces, para hacer que un kernel sea interrumpible en subkernels, se utiliza el concepto de *blockRange*, el cual se define como el número de blocks que se ejecutarán en un subkernel, limitado por un *blockIdx* inicial y otro final. Está granularidad de bloques ayuda a la reconfiguración de puntos de interrupción que auxilien a que las tareas de mayor prioridad puedan usar la GPU en modo preemptive.

2. *Partición de la transferencia de datos.* Aunque la transferencia de datos y el cómputo utilizan diferentes motores, la copia de memoria de una aplicación no puede realizarse simultáneamente que el lanzamiento del kernel de otro proceso ya que pertenecen a diferentes contextos. Y una GPU puede contener un contexto a la vez. Por lo tanto en un entorno multitarea, una operación de copia de memoria muy grande de baja prioridad, puede detener a las tareas de alta prioridad. Para evitarlo, GPES busca

dividir una transferencia de memoria non-preemptive en múltiples fragmentos más pequeños para volverla preemptive. En el límite de cada fragmento se inserta un punto preemptive, cada fragmento tendrá un tamaño de 4KB.

Ref.	Artículo	Clasificación	Implementación
[28]	**Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing**	Basado en Hardware	Tarjetas gráficas
[34]	**RGEM: A Responsive GPGPU Execution Model for Runtime Engines**	Colas masivas en paralelo	Tarjetas gráficas
[9]	**Preemption of a CUDA Kernel Function**		Tarjetas gráficas
[30]	**GPES: A preemptive execution system for gpgpu computing**	Modificación del API	Tarjetas gráficas
[29]	**Effisha: A software framework for enabling efficient preemptive scheduling of gpu**	Basado en Software: Entorno de Scripts	Tarjetas gráficas
[31]	**Intra-Node Memory Safe GPU Co-Scheduling**	Administración dinámica de la memoria	Tarjetas gráficas
[32]	**Priority-driven spatial resource sharing scheduling for embedded graphics processing units**		Tarjetas gráficas
[33]	**Run-Time Scheduling Framework for Event-Driven Applications on a GPU-Based Embedded System**	Basado en Software: Particion en Tareas	Sistemas embebidos
[10]	**GpuArt: An application-based limited preemptive gpu real-time scheduler for embedded systems**	Planificacion por prioridad	Sistemas embebidos

Tabla 3.1. *Matriz de clasificación de trabajos relacionados.*

3.2. Tarjetas gráficas

1. **REGM: Un modelo de ejecución GPGPU responsivo para soluciones en tiempo de ejecución (*REGM: A Responsive GPGPU Execution Model for Runtime Engines*)**

   El artículo [34] es el primer trabajo que genera un framework para utilizar tareas en tiempo real en tarjetas gráficas. Se basa en la partición en fragmentos de memoria a procesar, cada fragmento es agregado a una cola de procesamiento para ser ejecutado. Su solución es 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 GPU en cualquier momento, y así evitar interferencias de rendimiento causadas por lanzamientos concurrentes.

La primer característica de este framework es que se basa en transacciones de datos preemptive, por lo que los tiempos de bloqueo están limitados al tiempo limitado para copiar cada fragmento de dato. La segunda característica es que permite lanzar los kernels de diferentes tareas una por una basadas en su prioridad, lo que evita que las tareas con alta prioridad sean interferidas por la carga simultánea de trabajo una vez iniciadas. Sin embargo el lanzamiento del kernel puede bloquearse al haber un kernel de menor prioridad lanzado anteriormente, esto debido al probable alto uso de memoria global.

2. **Preemption de una función CUDA KERNEL (*Preemption of a CUDA Kernel Function*)**

   El artículo [9] presenta una solución basada en quantums, en donde para sacar una tarea en ejecución se verifica si ha terminado su tiempo. La prioridad de cada tarea esta dada por una estructura *FIFO*. La implementación permite que un kernel sea detenido, guarde su estado y sea reemplazado, liberando los recursos de la GPU para que se ejecuten otras tareas, lo que ayuda a aumentar la eficiencia del sistema.

   Se propone un esquema de puntos de control colocando una directiva tipo *pragma* dentro de las funciones kernel que ayudará a almacenar su estado de ejecución en la memoria del CPU en vez de la del GPU.

   La transferencia entre el CPU y GPU permite que ahorre memoria en el GPU, pero a la vez se posibilita que la ejecución se reanude casi como si el kernel nunca se hubiera detenido.

3. **GPES: Un sistema de ejecución para cómputo gpgpu (*GPES: A preemptive execution system for gpgpu computing*)**

   El articulo [30] implementa una serie de funciones para realizar particiones de kernel y de datos, esto realizando subkernels y dividiendo las transacciones de datos en fragmentos. Específicamente, se presenta una técnica de reescritura binaria para reconfigurar de manera transparente el código de los kernel. Mientras que para los kernel un poco más complejos, se desarrolló una técnica de transformación fuente a fuente

que compila el código del kernel transformado en binarios CUDA. La prioridad de las tareas esta dada por colas de ejecución. GPES modifica el API de CUDA utilizando las bibliotecas de openCUDA para reconfigurar el código binario de los kernels, esto lo realiza obteniendo un máximo de blocks que se pueden ejecutar por quantum. Para realizar esto, se ayuda de dos implementaciones.

a) *Transformación fuente a fuente.* Para dar soporte al cómputo paralelo, el hardware mantiene indices continuos en un grid (*blockIdx*), por ejemplo, si un grid consta de 256 blocks, tiene indices desde 0 a 255. Entonces, para hacer que un kernel sea interrumpible en subkernels, se utiliza el concepto de *blockRange*, el cual se define como el número de blocks que se ejecutarán en un subkernel, limitado por un *blockIdx* inicial y otro final. Está granularidad de bloques ayuda a la reconfiguración de puntos de interrupción que auxilien a que las tareas de mayor prioridad puedan usar la GPU en modo preemptive.

b) *Partición de la transferencia de datos.* Aunque la transferencia de datos y el cómputo utilizan diferentes motores, la copia de memoria de una aplicación no puede realizarse simultáneamente que el lanzamiento del kernel de otro proceso ya que pertenecen a diferentes contextos. Y una GPU puede contener un contexto a la vez. Por lo tanto en un entorno multitarea, una operación de copia de memoria muy grande de baja prioridad, puede detener a las tareas de alta prioridad. Para evitarlo, GPES busca dividir una transferencia de memoria non-preemptive en múltiples fragmentos más pequeños para volverla preemptive. En el límite de cada fragmento se inserta un punto preemptive, cada fragmento tendrá un tamaño de 4KB.

4. **Effisha: Un framework para permitir una planificación preemptive eficiente en gpu (*Effisha: A software framework for enabling efficient preemptive scheduling of gpu,*)**

5. **Planificación conjunta con GPU y aseguramiento de la memoria intra-nodo (*Intra-Node Memory Safe GPU Co-Scheduling*)** El artículo [31] propone la creación del framework schedGPU, el cual utiliza el administrador de trabajo Slurm para

planificar las tareas. Este framework administra las múltiples solicitudes para acceder a las GPU de forma segura al garantizar que no se produzcan sobrecargas de memoria durante su ejecución. 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.

6. **Planificación de recursos espaciales compartidos con prioridad para unidades de gráficos embebidos (*Priority-driven spatial resource sharing scheduling for embedded graphics processing units*)**

   El artículo [32] 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*, la cual limita el número de núcleos de procesamiento de una GPU para una tarea basándose en su prioridad, esto lo realiza modificando las bibliotecas de OpenGL-ES. Así se previene que una tarea que se encuentra en segundo plano retrase a otra que se encuentra en ejecución, también se minimiza la sobrecarga de planificación al invocarse

solamente dos veces, en el inicio de la tarea y en su finalización.

3.3. Sistemas embebidos

1. **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*)**

   Para poder utilizar varias aplicaciones en sistemas en tiempo real complejos es necesario la utilizar técnicas de implementación preemptive. 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 [33] 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. Utiliza un divisor de carga de trabajo que divide el núcleo de la GPU en múltiples subkernels durante el tiempo de ejecución para implementar el modo preemptive. Dependiendo del estado actual de sistema y de la prioridad, el divisor de carga de trabajo decide el número y el tamaño de cada subkernel.

   Se cuenta con un generador de ejecución planificada, el cual, dependiendo del estado actual de uso de los recursos del sistema y del plazo límite del la tarea, lanza una secuencia de tareas para maximizar el numero de aplicaciones cercanas al su plazo vencido.

2. **GpuArt: Un planificador de gpu en tiempo real basado en aplicaciones con preemptive limitado para sistemas embebidos (*GpuArt: An application-based limited preemptive gpu real-time scheduler for embedded systems*)**

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.

3.4. 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.

Capítulo 4

Diseño del framework

El objetivo principal de este capítulo es describir el diseño del framework propuesto para planificar tareas preemptive en sistemas embebidos heterogéneos. Se plantea la estructura del mismo junto con la descripción de su solución.

Aunque se tomó como base el sistema embebido heterogéneo NVIDIA Jetson TX2, el diseño puede ser aplicado a cualquier dispositivo, siempre y cuando cumpla con la característica de tener memoria unificada, como la descrita en la sección 2.4.3.1.

4.1. Descripción general del framework

En la Figura 4.1 se muestra el diagrama a bloques que representa los elementos del framework propuesto. El primer bloque es el módulo de Puntos Preemptive, donde se plantea la propuesta para implementar el modo preemptive en las tareas, se contempla desde las directivas que se deben implementar hasta el manejo del contexto y sus variables. Justo después se describe un módulo en el que se engloban todos los aspectos del manejo de memoria entre el CPU y el GPU. Finalmente se tienen los módulos para la la planificación y la asignación de prioridades a cada tarea, las cuales son un complemento una de la otra para la correcta implementación de alguno de los algoritmos descritos en la sección 2.2.3.

En los siguientes apartados de la tesis se analizarán y desglosarán cada uno de los módulos.

Figura 4.1. *Diagrama del framework para la planificacion de tareas preemptive en sistemas embebidos heterogeneos.*

4.2. Puntos preemptive

La mayoría de los trabajos relacionados en el estado del arte colocan puntos preemptive para formar fragmentos (chunks) de memoria.

Con esta solución se colocan puntos preemptive fijos dentro de las funciones kernel de la aplicación. Con esto se nos permite:

- Direccionar eventos.
- Manejar las interrupciones de la tarea.
- Modificar la granularidad de los subkernels.
- Determinar las directivas para detener y restaurar una tarea.

4.3. Memoria

En este apartado,

Como se mencionó en la sección 4.2, la mayoría de los trabajos relacionados en el estado del arte colocan puntos preemptive para formar fragmentos (chunks) de memoria.

La solución da prioridad a la creación de copias de memoria y transferencia entre GPU y CPU.

La arquitectura del sistema permite el soporte de memoria unificada.

No es necesario preocuparse por las transacciones de transferencia de datos.

Disminución del consumo energético.

Ahorro de recursos.

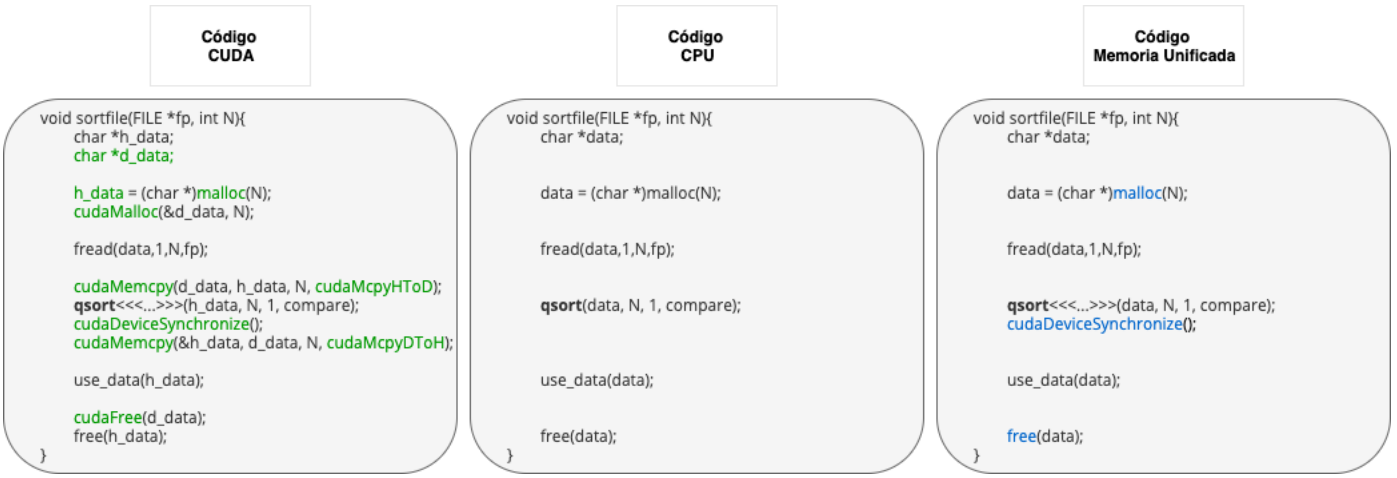

Figura 4.2. *Comparación de directivas para manejo de memoria.*

4.4. Planificación

4.4.1. Planificación estática

Sólo un artículo incluye soporte para la planificación con algoritmos en sistemas en tiempo real.

La prioridad es fija, ya que se asigna a partir de una cola FIFO.

Esta solución permite dar soporte para planificación con algoritmos de sistemas en tiempo real.

La prioridad puede ser estática o dinámica y está dada por el algoritmo de planificación.

4.4.2. Planificación dinámica

4.5. Asignación de prioridades

4.5.1. Asignación estática de prioridades

4.5.2. Asignación dinámica de prioridades

La asignación de prioridades y la implementación de planificadores dinámicos de tareas de dejarán para trabajo futuro.

4.6. Resumen

Capítulo 5

Evaluación del rendimiento

Este capítulo propone posibles métricas para evaluar el rendimiento del framework contra alguna otra solución.

5.1. Métricas de rendimiento

5.1.1. Granularidad de kernel

5.1.2. Granularidad de datos

5.1.3. Cambio de contexto

5.1.4. Rendimiento de multitarea

- Tiempo de respuesta: Tiempo transcurrido entre la creación del contexto y su destrucción.
- Tiempo de ejecución del programa: Número de intervalos de tiempo entre el lanzamiento de un subkernel hasta su finalización.
- Tiempo de ejecución de una transacción de memoria: El período de tiempo que tarda una transacción de copia a la GPU.

- Tiempo de ejecución total. Suma del tiempo de ocupación de cómputo y el tiempo de ocupación de copia.
- Tiempo de ociosidad. La diferencia entre el tiempo de respuesta y el tiempo de ejecución total.

Capítulo 6

Conclusiones y Trabajo Futuro

Este capítulo presenta un resumen del trabajo propuesto, las conclusiones a partir de los resultados obtenidos en el Capítulo 4, recapitula las contribuciones del trabajo realizado y el trabajo futuro.

Partiendo de la hipótesis presentada en el Capítulo 1:

6.1. Contribuciones

Las contribuciones del presente trabajo son:

6.2. Trabajo futuro

A continuación se muestran algunos temas de trabajos futuros:

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.
- **Kernel**: Función o fragmento de código acelerado en una GPU. No está relacionado con el kernel de un Sistema Operativo.
- **Deadline**: Tiempo límite, es el momento justo antes en que una tarea debe completar su ejecución
- **Quantum**: También llamado quantum o cuanto. El período de tiempo durante el cual se permite que un proceso se ejecute en un sistema multitarea preemptivo generalmente se denomina intervalo de tiempo o quanto.
- **DMA**: Acceso directo a memoria. Permite a ciertos dispositivos de diferentes velocidades acceder a la memoria del sistema para leerla o escribirla sin pasar por el CPU, esto sin generar una carga masiva de interrupciones.
- **Barrera**: Un método de sincronización. Cuando en el código fuente se encuentra una barrera, el grupo de procesos debe detenerse hasta que todos ellos lleguen a ella.
- **Slurm**: Simple Linux Utility for Resources Management, es un sistema de gestión de tareas y de clusters[**?**].

Bibliografía

[1] B. Priambodo, "Cooperative vs. preemptive: a quest to maximize concurrency power." [Online]. Available: https://medium.com/traveloka-engineering/cooperative-vs-preemptive-a-quest-to-maximize-concurrency-power-3b10c5a920fe

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

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

[4] ——, "Nvidia jetson tx2 delivers twice the intelligence to the edge." [Online]. Available: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/

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

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

[7] 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.

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

[9] J. Calhoun and H. J. and, "Preemption of a cuda kernel function," *13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2012.

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

[11] 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.

[12] A. S. Tanenbaum, *Modern Operating Systems.* Pearson Education, Inc., 2008.

[13] C. Liu and J. Leyland, "Scheduling algorithm for multiprogramming in a hard real-time environment," *Journal of the Association for Computing Machinery*, pp. 46–61, January 1973.

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

[15] J. Lehoczky, L. Sha, and Y. Ding, "he rate monotonic scheduling algorithm: Exact characterization and average case behavior," *IEEE International symposium on real time system*, 1989.

[16] W. Li, K. Kavi, and R. Akl, "A non-preemptive scheduling algorithm for soft real-time systems," *Computers and Electrical Engineering*, vol. vol 33, no. 1, pp. 12–29, 2007.

[17] V. Shinde and S. C. Biday, "Comparison of real time task scheduling algorithms," *International Journal of Computer Applications*, vol. 158, no. 6, pp. 37–41, enero 2017.

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

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

[20] NVIDIA, "Arquitectura pascal de nvidia computación infinita para oportunidades infinitas." [Online]. Available: https://www.nvidia.com/es-la/data-center/pascal-gpu-architecture/

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

[22] 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/

[23] 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

[24] 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

[25] 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.

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

[27] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," pp. 193–204, June 2014.

[28] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing," pp. 358–369, March 2016.

[29] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "Effisha: A software framework for enabling efficient preemptive scheduling of gpu," *AMC*, 2017.

[30] H. Zhou, G. Tong, and C. Liu, "Gpes: A preemptive execution system for gpgpu computing," *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015.

[31] 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.

[32] 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.

[33] 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.

[34] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. R. Rajkumar, "Rgem: A responsive gpgpu execution model for runtime engines," *32nd IEEE Real-Time Systems Symposium*, 2011.