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. Eistemas 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[7] 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[8]:

- 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 [9].

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. Esquemas de planificación

2.2.2.1. Planificación cooperativa

En en el esquema de planificación ***cooperativa*** o non-preemptive las tareas deciden el tiempo que utilizarán los recursos de cómputo, el trabajo del planificador es únicamente el de asignar tareas a los nodos de procesamiento disponibles.

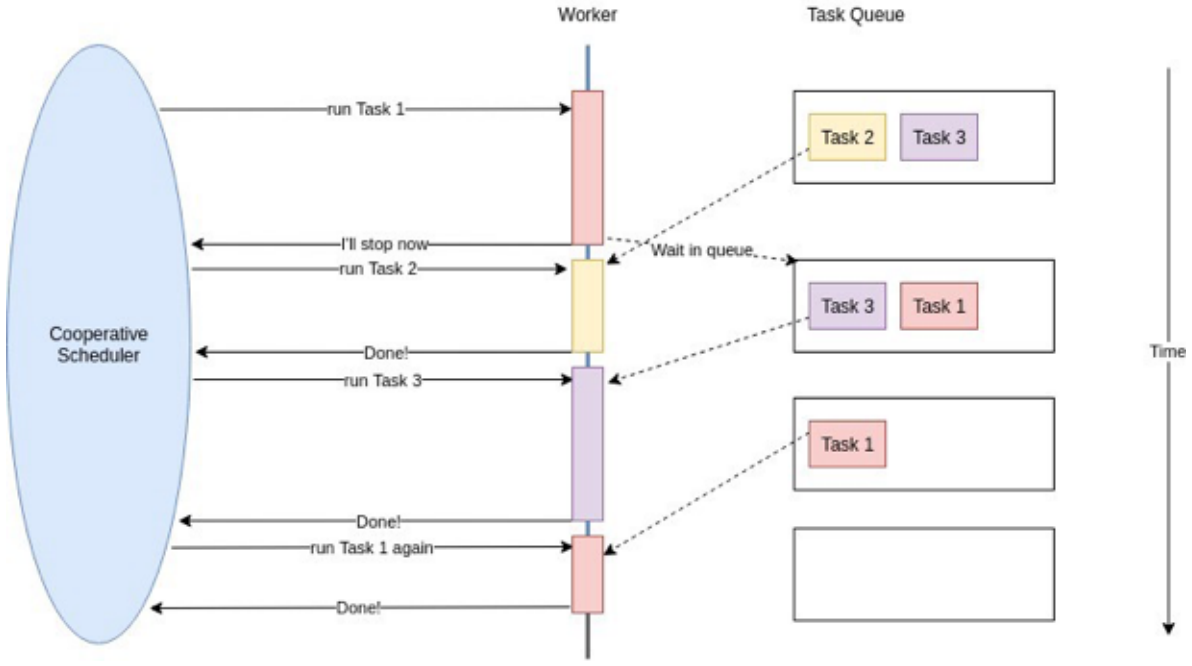

Figura 2.1. *Planificación cooperativa.[1]*

El problema de esto es que una tarea puede absorber todos los recursos del sistema generando plazos vencidos en las demás tareas.

2.2.2.2. Planificación preemptive

En este esquema, el planificador es el que tiene la batuta ante las tareas, las asigna a los recursos disponibles, y les define un tiempo de ejecución máximo, comúnmente llamado quantum[10]. Superado este punto, el planificador interrumpe (preempts) la tarea para que otra sea ejecutada en su lugar, y la tarea interrumpida debe esperar hasta que le toque su turno nuevamente.

Debido a que muchas veces se interrumpen tareas a la mitad de un proceso, es necesario

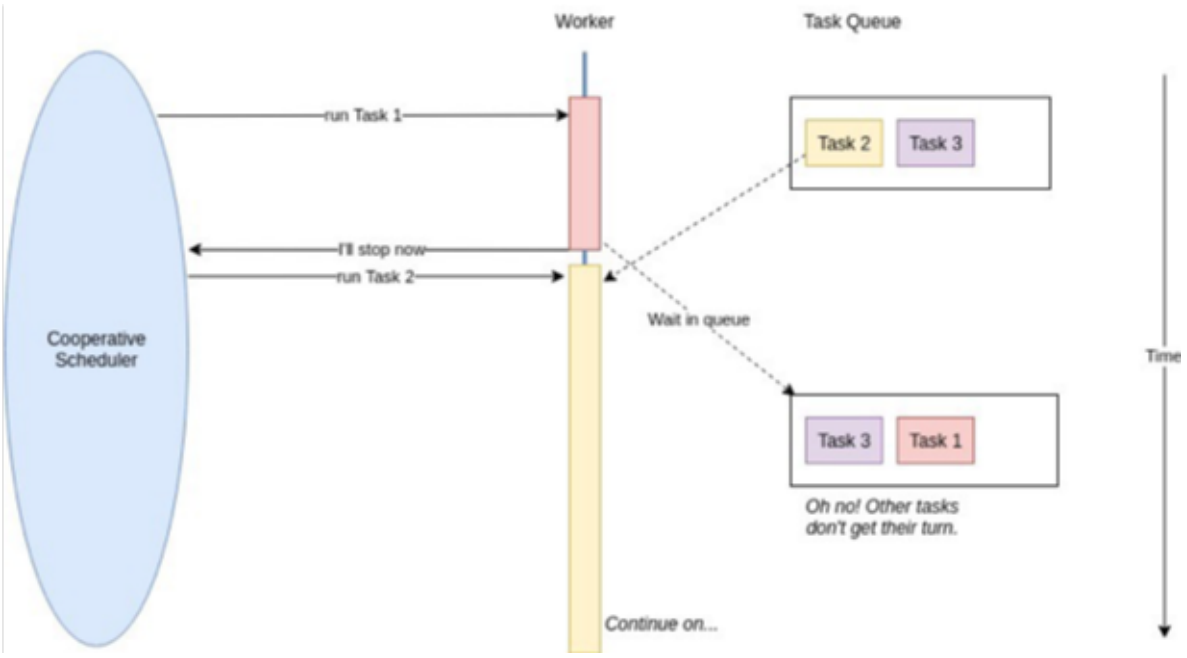

Figura 2.2. *Planificación cooperativa con plazos vencidos.[1]*

almacenar y restaurar el estado que se tenia antes de dicha interrupción para continuar justo en el punto en donde se quedó.

A este proceso de almacenamiento, intercambio y restauración del estado de las tareas se le denomina **cambio de contexto** (**context switch**).

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.

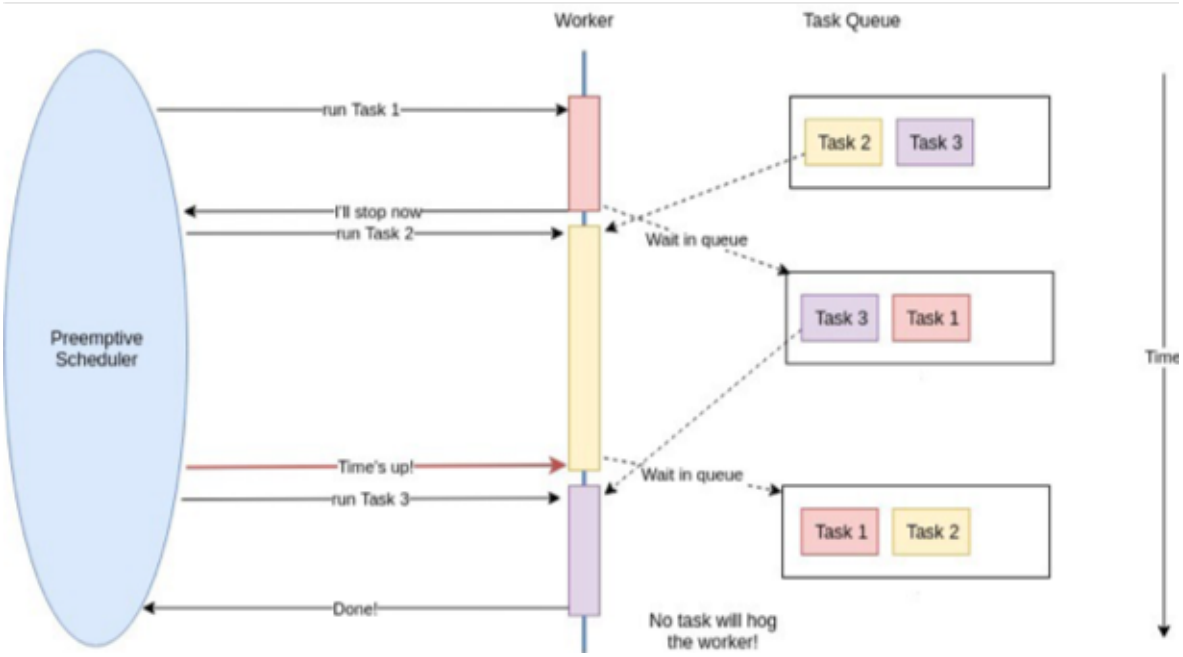

Figura 2.3. *Planificación preemptive.[1]*

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 [11]. 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.

Algorithms	Implementation	Priority Assignment	Scheduling Criteria	Preemptive/ Non-Preemptive	CPU Utilization	Efficiency
EDF	Difficult	Dynamic	Deadline	Preemptive	Full Utilization	Efficient in Underloaded Condition
RM	Simple	Static	Period	Preemptive	Less	Efficient in overloaded condition as compared to EDF
DM	Simple	Static	Relative Deadline	Preemptive	More as compared to RM	Efficient
LLF	Difficult	Dynamic	Laxity	Preemptive	Full Utilization	Efficient
GEDF	Difficult	Dynamic	Deadline and within group Shortest Execution time (SJF)	Non-Preemptive	Full Utilization	Efficient in Non-preemptive environment
GPEDF	Difficult	Dynamic	Deadline and within group SJF	Preemptive	Full Utilization	Efficient

Figura 2.4. *Matriz de comparación de algoritmos de planificación[2].*

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[12].

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[4]. 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[13].

  - 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.

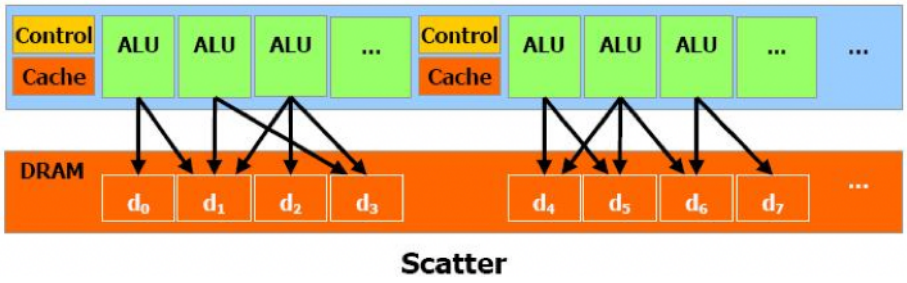

Figura 2.5. *Escritura en DRAM[3].*

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

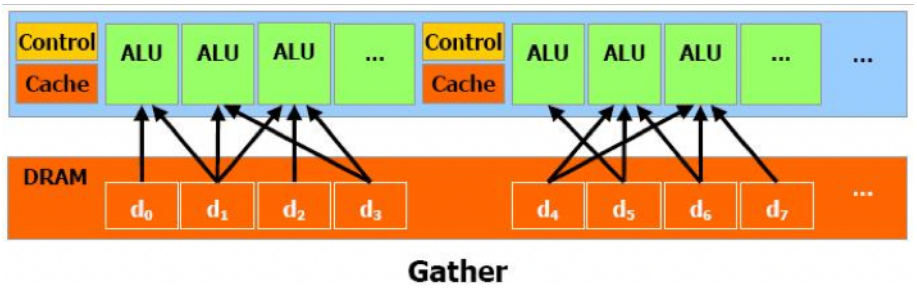

Figura 2.6. *Lectura en DRAM[3].*

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.

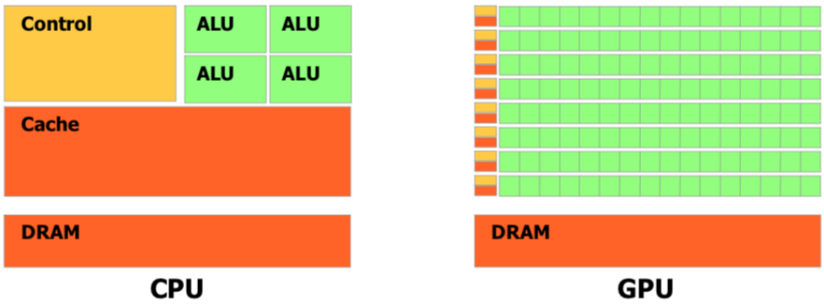

Figura 2.7. *Representación de un CPU y un GPU[3].*

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

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 simpli-

fica 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[14].

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[14]. 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 [15], 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[16].

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 [17].

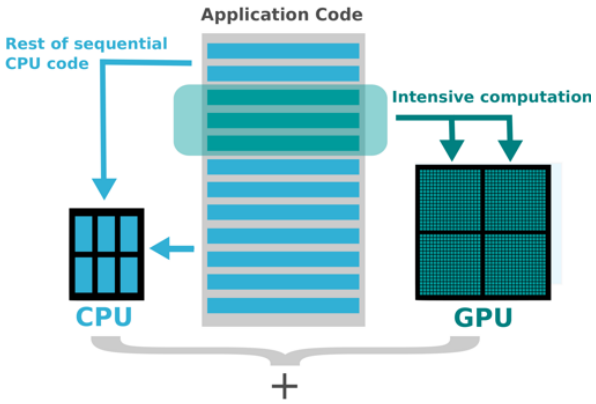

Figura 2.8. *Aceleración de programas en GPUs[4].*

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.

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 [18].

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 [19], 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 [20] 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 2.9 se muestra diagrama de bloques de la arquitectura del sistema Jetson TX2.

2.6.1. Jetson TX2

Las especificaciones del sistema están descritas en la tabla 2.1.

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.

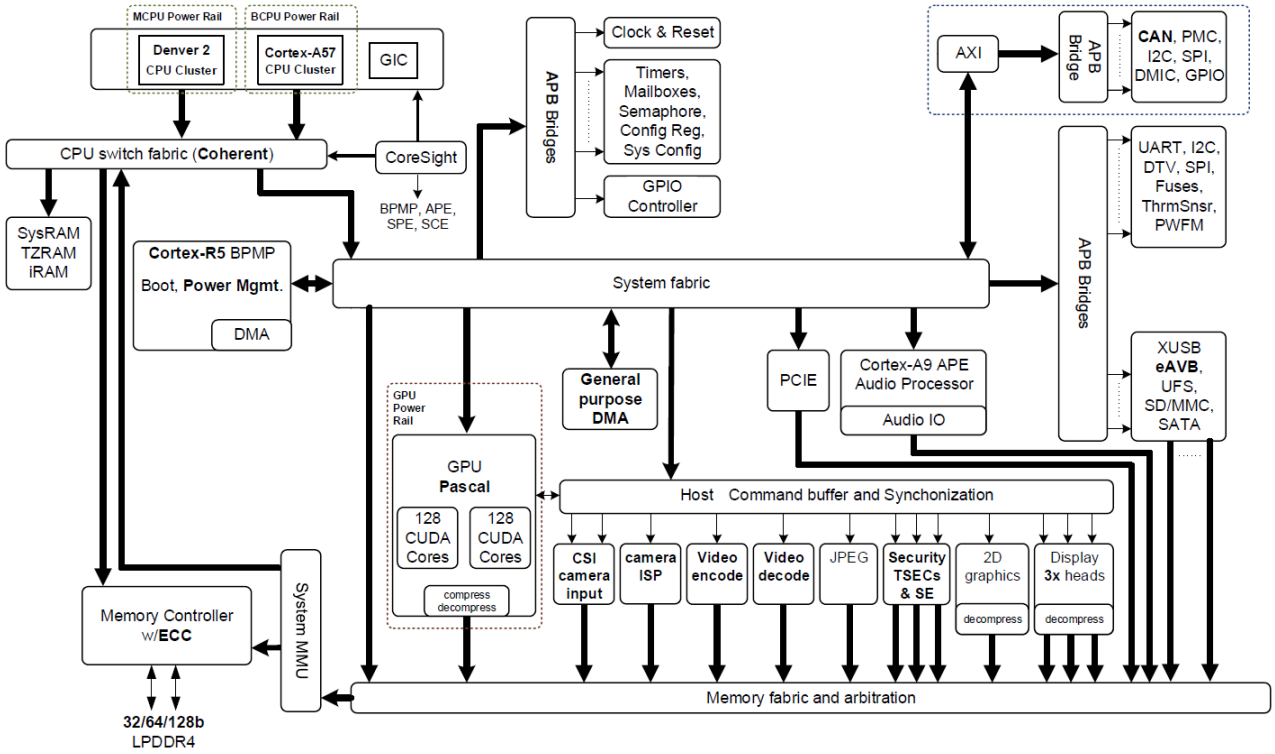

Figura 2.9. *Diagrama de la arquitectura del sistema Jetson TX2[5].*

- 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.

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 2.1. *Especificaciones del sistema Jetson TX2[6].*

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[10], 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

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"* [18] 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 [20] 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"*[21] 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"*[22] 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"*[23] 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"*[24] 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"*[25] 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.

PRUEBAS

[24] [26] [27] [19]

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. De lado izquierdo, se encuentran enmarcados los componentes Puntos Preemptive y Memoria y Prioridad y Planificación estática, los cuales serán descritos más adelante. Para los bloques del lado derecho, Asignación de prioridades y Planificación dinámica, aunque forman parte de la propuesta de diseño, su análisis de elementos se dejará para un trabajo futuro.

En el módulo de Puntos Preemptive, 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

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

tienen tres módulos para la asignación de prioridades y la planificación. Cuando se requiere implementar un planificador preemptive en sistemas con un número constante de tareas que mantienen su prioridad a lo largo del tiempo, se puede implementar algún algoritmo de planificación estático como los descritos en la sección 2.2.3, en ellos la prioridad está predefinida por los parámetros del algoritmo, por ello no es necesaria la partición en dos bloques. Pero en el momento en el que se requiera tener un método de asignación de prioridades personalizado, es necesario tener un módulo que lo permita. Aunado a esto, si por alguna razón se solicita agregar tareas dinámicamente con el sistema en ejecución, se deben tener mecanismos para manejar cualquier interrupción o actualización de información del planificador. Ambos elementos son necesarios en un framework, pero sus componentes internos se dejarán para ser resueltos como trabajo futuro.

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 pioridad a la creación de copias de memoria y transferencia entre device host.

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.

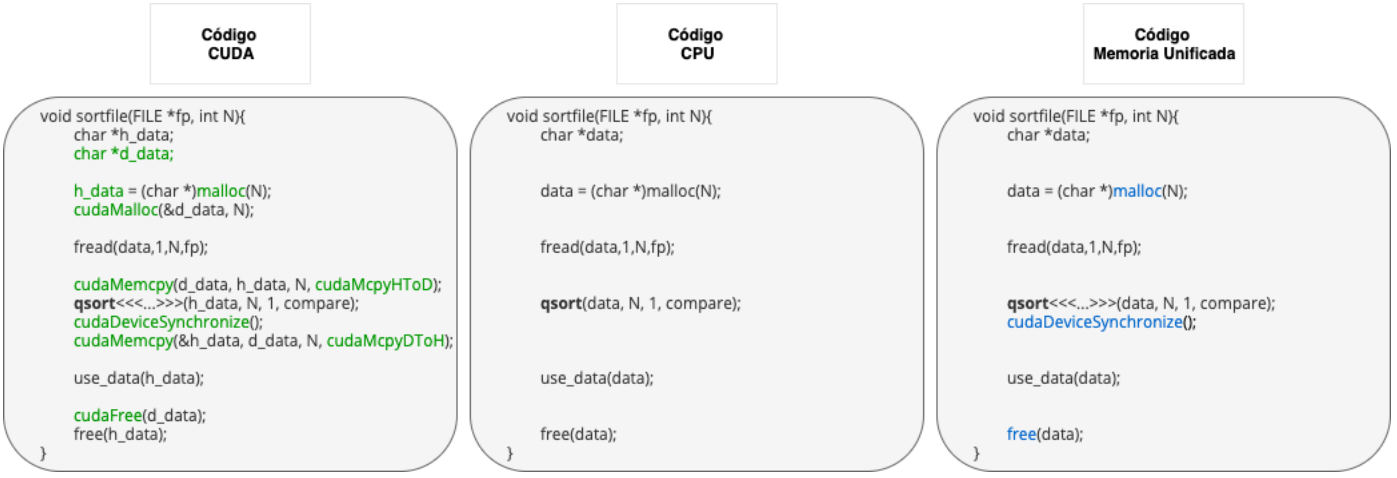

Figura 4.2. *Comparación de directivas para manejo de memoria.*

4.4. Prioridad y 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 es estática y está dada por el planificador.

4.5. Asignación de Prioridades y Planificación dinámica

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

4.6. Resumen

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.

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] 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.

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

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

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

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

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

[8] 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.

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

[10] 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.

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

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

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

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

[15] 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/

[16] 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

[17] 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

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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