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
- Unidad Mérida -

Ciudad Universitaria, CDMX diciembre 2020

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Posgrado en Ciencia e Ingeniería de la Computación,
Universidad Nacional Autónoma de México

José Antonio Ayala Barbosa

Maestría en Ciencia e Ingeniería de la Computación
1 de diciembre de 2020

Este trabajo fue apoyado por los proyectos UNAM-PAPIIT IA104218, IA102620

Agradecimientos

A mis abuelas y abuelo, a mi madre, a mi padre y a mi hermano, que siempre me apoyaron incondicionalmente.

A la profesora Rosalía, amigos y compañeros, por su ayuda en la culminación de este trabajo.

Al Dr. Paul Erick, por su visión crítica, por las enseñanzas, consejos, y sobre todo, por la paciencia que me brindó.

A PROTECO y al profesor César Govantes, por su apoyo y motivación para incursionar en el campo de la investigación.

A mis profesores, al PCIC, a la UNAM y al CONACYT, por todas las bondades y ayuda que me han obsequiado durante mi formación académica y profesional.

Índice general

Índice de figuras

Índice de tablas

Índice de Algoritmos

CAPÍTULO
1

Introducción

Los sistemas embebidos tienen el fin de cumplir con tareas específicas, están programados generalmente en lenguajes nativos para satisfacer necesidades con una pronta respuesta, por ello son los candidatos principales para la integración de aplicaciones en tiempo real. Éstas deben reaccionar dentro de límites de tiempo precisos para garantizar una correcta funcionalidad, satisfacer los criterios de calidad o evitar daños críticos. Por esta razón, actualmente las organizaciones recurren a los sistemas embebidos en vez de emplear computadoras de propósito general.

Las tecnologías emergentes requieren de soluciones cada vez más intensivas, por ello existe una creciente migración de paradigma en las empresas para acelerar sus aplicaciones embebidas mediante la utilización de unidades de procesamiento gráfico de propósito general ($GPGPU$) con el fin de solventar las demandas de recursos[8].

Las GPU modernas se adaptan ampliamente a entornos multitarea desde centros de datos hasta teléfonos inteligentes. Sin embargo, el soporte actual para su programación está limitado, formando una barrera para que la tarjeta gráfica satisfaga las necesidades de las organizaciones.

A pesar del gran esfuerzo que las empresas fabricantes de tarjetas gráficas están haciendo por generalizar el uso de sus plataformas, no es suficiente debido a que la tecnología está en constante cambio y a la privacidad con que se manejan, no se conoce completamente la arquitectura de operación. Por esta razón se requiere buscar nuevas formas de administrar los sistemas y aplicaciones que se ejecutarán en ellas.

El objetivo principal de esta tesis es definir el diseño de un *framework* que facilite la planificación de tareas *preemptive*, específicamente, en sistemas embebidos heterogéneos, es decir, aquellos que integran tanto una Unidad Central de Procesamiento (CPU) y una Unidad de Procesamiento Gráfico (GPU). La solución que se presenta está orientada a aplicaciones cuyos requisitos de ejecución cumplen con las características de procesamiento embebido y pueden ser manejadas por un planificador de sistemas en tiempo real. La implementación del modelo se dejará como trabajo futuro.

El problema a la hora de implementar aplicaciones embebidas heterogéneas es la documentación limitada con la que actualmente cuenta la literatura. Este trabajo de tesis nos brinda la oportunidad de idear las bases de un *framework* que facilite el diseño y/o desarrollo de aplicaciones en tiempo real que utilicen tareas con suspensión *preemptive*.

Tener las bases del diseño de un *framework* que permita planificar la ejecución de tareas *preemptive* facilitará la disminución de los plazos vencidos de tareas con alta prioridad y mejorará el desempeño general del sistema.

1.1 Estructura de la tesis

El presente trabajo se estructura en seis capítulos. En el capítulo **Antecedentes**, se da una introducción a los conceptos que forman parte del marco teórico, y que son necesarios para entender el contexto en el que se desenvuelve el trabajo. Enseguida, 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** donde se describe puntualmente la propuesta de solución. Una vez presentado el diseño, en el capítulo **Métricas de Rendimiento** se anexan algunas métricas de rendimiento que se le pueden aplicar al *framework* cuando, en un futuro, sea implementado. Finalmente. En el capítulo **Conclusiones y Trabajo Futuro** se recapitulan los alcances del trabajo y se mencionan los puntos posibles a desarrollar para un trabajo futuro.

CAPÍTULO
2

Antecedentes

En este capítulo se presentan conceptos que forman parte del marco teórico, y son importantes para entender el contexto en el que se desenvuelve el trabajo.

Se inicia mostrando la tendencia de organizar los componentes de un sistema y la necesidad de creación de estructuras metodologías para facilitar el desarrollo de software.

Se continúa con los sistemas en tiempo real, su clasificación, componentes y la planificación de tareas.

Enseguida se definen las características y arquitectura del procesador y la tarjeta gráfica para introducir el concepto de cómputo de propósito general en unidades de procesamiento gráfico y su utilización en sistemas embebidos heterogéneos.

Finalmente, se presenta el material de trabajo con el que se realizó el análisis y las propuestas del presente trabajo.

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 es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software; es decir, la aplicación de la ingeniería al software.[9]"*

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[10]:

- 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 mayor calidad posible de acuerdo a las necesidades y los resultados que se deseen 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 y metodología 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 mecanismos ya implementados al reutilizar código. Un *framework* puede ser considerada como un molde configurable, al que se pueden añadir atributos especiales para finalmente construir una solución integrada.

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

El objetivo de este trabajo es el diseño del *framework* para planificar tareas *preemptive* en sistemas embebidos heterogéneos.

2.1.2 Métricas de software

A la hora de probar la corrección de un software es necesario realizar pruebas de software que nos permitan observar si se están cumpliendo requisitos mínimos de implementación.

Comúnmente se dividen en dos grupos[11]:

- Pruebas funcionales: Se prueba la funcionalidad que se requiere implementar en el producto de software. Se valida y verifica que el software cumple con lo especificado y hace lo que debe hacer y cómo debe hacerlo.
- Pruebas no funcionales: Se prueban los aspectos relacionados con el comportamiento del producto de software, no con el uso final.

Las pruebas más conocidas y que se utilizan comúnmente en la industria son[12]:

- Compatibilidad: Mide la capacidad de convivir y trabajar en conjunto, ya sea entre sus propios componentes o con las aplicaciones vecinas.
- Adaptabilidad: Mide la tolerancia a los cambios del entorno actual.
- Seguridad: Mide e identifica los riesgos de infiltración y vulnerabilidades presentes en el software.
- Protección crítica: Mide el grado de corrección de los resultados del software y el grado de peligro al que se enfrentan los usuarios si algo sale mal. Es decir, permite prever escenarios desastrosos como daños al medio ambiente, lesiones graves a personas e incluso, la muerte.
- Estabilidad: Mide la eficiencia y la capacidad del software para funcionar continuamente durante un periodo determinado de software. Se mide si en ese periodo de uso normal falla o se presentan errores.
- Rendimiento: Mide el desempeño que puede alcanzar un producto de software con distintas configuraciones o parámetros de uso, involucra tanto tiempos de respuesta como nivel de ocupación de los recursos de hardware.

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 sólo del resultado del cálculo, sino también del momento (tiempo) en que se produce [13].

Un sistema en tiempo real debe responder a entradas generadas dentro de un periodo de tiempo específico para evitar posibles fallas. Éste se vale de un planificador, el cual asigna las tareas pendientes de ejecución a los recursos de procesamiento, también brinda una jerarquía de prioridad y define un tiempo de ejecución máximo.

Un sistema en tiempo real esta formado en su forma básica por tareas de las aplicaciones y la planificador. Las tareas pueden ser descritas por un tiempo de consumo, período o evento y plazo límite.

El *deadline* o plazo límite es el momento crítico en que la tarea debe completar su ejecución. Existen tres tipos de plazos límite[13]:

- *Soft deadline*: En esta tarea se pueden superar algunos tiempos límites y el sistema aún puede funcionar correctamente.
- *Firm deadline*: Aquí los resultados obtenidos en los plazos vencidos no son útiles, pero los plazos vencidos son tolerados frecuentemente.
- *Hard deadline*: Si una tarea no se cumple en el plazo 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 con base a su ejecución que están presentes en los sistemas en tiempo real[14]:

- Tareas periódicas: Se ejecutan en un intervalo fijo de tiempo conocido. Normalmente, las tareas periódicas tienen restricciones que indican sus plazos de tiempo.
- Tareas aperiódicas: Se ejecutan aleatoriamente en cualquier instante 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.

En general, los intervalos de tiempo se dan por el plazo límite de una tarea.

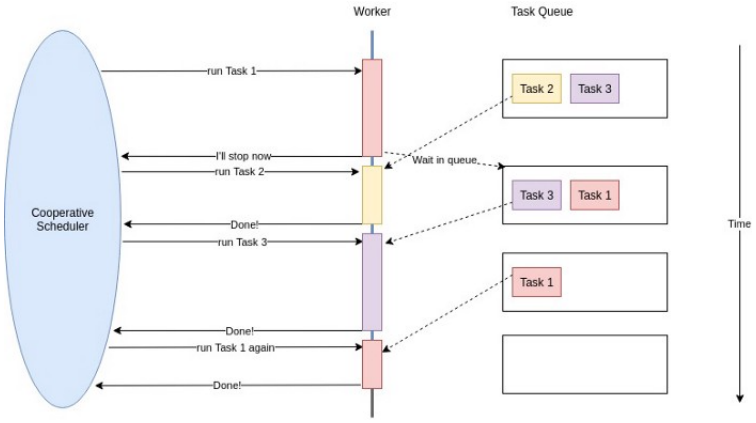

Figura 2.1: Planificación cooperativa.[1]

2.2.2 Esquemas de planificación

2.2.2.1 Planificación cooperativa

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

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

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*[15]. Superado este punto, el planificador interrumpe la tarea para que otra de mayor prioridad se ejecute en su lugar, y la tarea interrumpida espera hasta que le toque su turno nuevamente. Un ejemplo de este esquema, se muestra en la figura 2.3.

La mayor diferencia entre la planificación cooperativa y la *preemptive* radica en que la primera debe ejecutar la tarea de principio a fin, y la segunda puede interrumpir las tareas si así es requerido por el planificador.

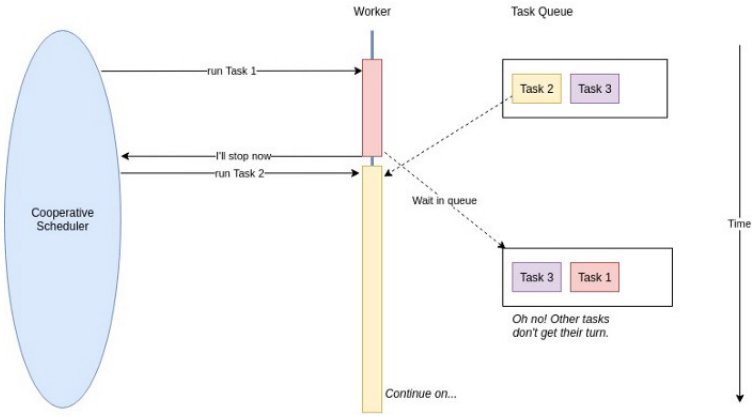

Figura 2.2: Planificación cooperativa con plazos vencidos.[1]

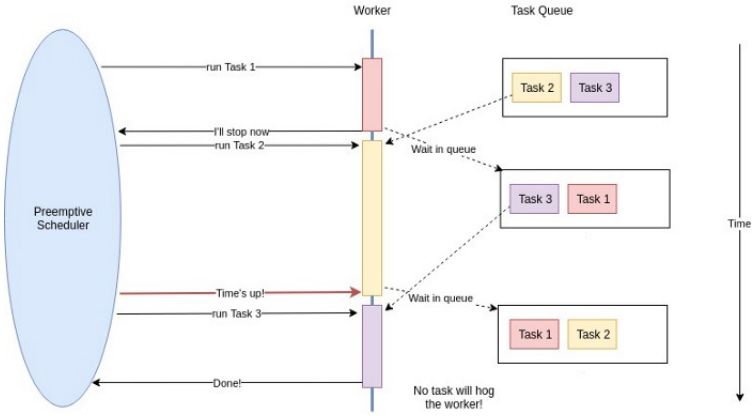

Figura 2.3: Planificación preemptive.[1]

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*.

Como se ha observado, la planificación *preemptive* se ha convertido en un requisito para la implementación de los sistemas en tiempo real. Por lo que ha surgido una clasificación dependiendo de su implementación[16].

- ***Planificación preemptive completa***. Inmediatamente después de terminar el *quantum*, se saca de ejecución la tarea actual.

- ***Planificación preemptive limitada***. En la mayoría de los casos, un planificador totalmente *preemptive* produce suspensiones innecesarias. Para reducir la sobrecarga en tiempo de ejecución, se han propuesto diversas soluciones[16]:

  - ***Planificación de umbrales preemptive***. Esta solución permite que una tarea deshabilite la suspensión *preemptive* dependiendo del nivel de prioridad. Por lo tanto, a cada tarea se le asigna una prioridad y un umbral *preemptive*. Por lo que la suspensión *preemptive* se activa cuando la prioridad de la tarea que llega a la cola de ejecución es mayor que el umbral de la tarea en ejecución.

  - ***Planificación de suspensiones preemptive diferidas***. Cada tarea puede ser ejecutada en un periodo cooperativo. Aquí cada suspensión se pospone por un periodo determinado de tiempo, en vez de estar específicamente en un lugar en el código. Dependiendo de su implementación puede encontrarse en dos clasificaciones:

    - ***Modelo flotante***. En este modelo, las regiones *preemptive* son definidas por el programador insertando primitivas específicas en el código que habilitan y deshabilitan las suspensiones. Debido a que el tiempo de activación de cada región no está predefinido, se considera que los puntos *preemptive* están flotando en el código.
    - ***Modelo de activación por disparadores***: Se divide en dos regiones, la región cooperativa es activada por la llegada de una tarea con mayor prioridad y planificadas por un temporizador para durar exactamente su *quantum* (a menos que terminen antes), después de lo cual se habilita el modo *preemptive*. Si la tarea en la cola es de menor prioridad, no se interrumpe la que se encuentra en ejecución hasta que termine el siguiente *quantum*, con lo que se decide si se suspende o continua en ejecución.

  - ***Puntos preemptive fijos***. Una tarea se ejecuta implícitamente en modo cooperativo y la suspensión sólo se permite dentro de ubicaciones predefinidas dentro del código de la tarea, llamadas puntos *preemptive*. De esta manera una tarea se divide en varios fragmentos cooperativos. Si llega una tarea de mayor prioridad entre dos puntos de la que está actualmente en ejecución, la suspensión se pospone hasta el siguiente punto *preemptive*.

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 plazo límite en la medida de lo posible. Cuando a un algoritmo que pasa las métricas de la función de utilidad (ver capítulo 5) y todas las tareas son planificables, se dice que es un algoritmo óptimo ya que consigue la máxima utilización del procesador [13].

La planificación en tiempo real se puede dividir en:

- Estática: Todas las prioridades se asignan al diseñar el sistema y éstas se mantienen constantes durante el tiempo de vida de una tarea.
- Dinámica: Se asignan prioridades en el 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.

Como parte de la revisión de la literatura del estado del arte se generó la tabla 2.1 que contiene los principales algoritmos de planificación, así como algunas de sus características de operación.

Algoritmo	Asignación de prioridad	Criterio de planificación	Preemptive/ Cooperativa	Utilización de CPU	Eficiencia
SJF	Estática	Tiempo de Ejecución	Cooperativo	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	Cooperativo	100 %	Eficiente en ambientes cooperativos

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,

ya que minimiza la cantidad promedio de tiempo que cada tarea debe esperar hasta que se complete su ejecución [14]. Este algoritmo funciona únicamente con tareas cooperativas, por lo que fácilmente puede llegarse a un estado de inanición (estado donde no puede realizar ninguna acción), 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 límite más próximo tiene la mayor 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[17].

Es un algoritmo muy extendido en sistemas en tiempo real debido a su optimalidad teórica en el campo cooperativo, aunque al momento de implementarlo en un planificador con soporte *preemptive* el resultado puede acarrear un exceso de ejecución si se toma el peor caso [18]. Es el algoritmo más extendido al momento de realizar los primeros bosquejos de un sistema en tiempo real.

2.2.3.3 **Rate Monotonic**

Rate Monotonic (RM) es un algoritmo de planificación *preemptive* con prioridad estática para un solo procesador[17]. RM asigna la prioridad más alta a la tarea con el periodo más corto, suponiendo que los periodos sean igual a los plazos límites ($P_i = D_i$). En caso de que la tasa de demanda sea mayor el periodo sería más corto y, por ende, la prioridad aumentaría. Por ello es óptimo 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[19].

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, es decir, cuando se cumple que el plazo es menor al tiempo de la tarea ($P_i < T_i$) y el periodo es igual al plazo limite ($P_i = D_i$) se puede ver a RM como un caso especial de DM [20]. DM ejecuta en cada instante de tiempo la tarea con el plazo más corto, por lo que si dos o más tareas tienen el mismo plazo límite, la siguiente en ejecutarse debe elegirse aleatoriamente.

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, esta laxitud es la cantidad máxima de tiempo que un trabajo puede esperar cumpliendo su plazo límite. En este algoritmo, que tiene soporte *preemptive*, se otorga la máxima prioridad al trabajo con la menor laxitud y se permite que la tarea actualmente en ejecución sea intercambiada por otra con menor laxitud en cualquier momento [20].

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. Con ello, se obtienen numerosos cambios de contexto durante la vida útil de las tareas y el rendimiento del sistema en general se ve reducido. Este algoritmo es óptimo para sistemas con tareas periódicas [21].

2.2.3.6 Gang Earliest Deadline First

Gang Earliest Deadline First (GEDF) está pensado para mejorar el desempeño de EDF durante condiciones de sobrecarga[22]. La idea principal de su funcionamiento es agrupar las tareas con plazo límite similares y dentro de cada grupo planificar las tareas con SJF [21]. El parámetro rango de grupo (Gr) es un porcentaje de la tarea al comienzo del plazo absoluto de cada cola que determina el ingreso de la tarea a un grupo. Este algoritmo tiene soporte multiprocesador, por lo que en este ambiente resulta más efectivo[22].

2.3 CPU

La unidad de procesamiento central o CPU es un procesador de propósito general, lo cual significa que puede hacer una gran variedad de cálculos pero está diseñado para realizar el procesamiento de información en serie y 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 memorias de la computadora y permite el acceso a los periféricos.

2.3.2 Núcleos de procesamiento CPU y GPU

Es necesario destacar que los núcleos de un CPU y un GPU en principio son similares, pero entre ellos existen diferencias. Un núcleo 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 núcleo de GPU es más ligero y está optimizado para realizar tareas de paralelismo de datos con un control lógico simple enfocándose en la tasa de transferencia de los programas paralelos.

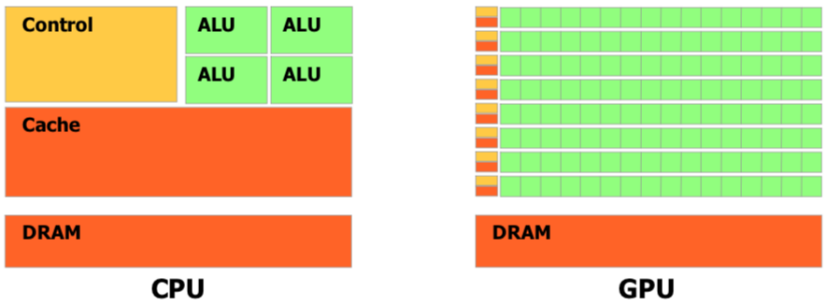

Figura 2.4: Representación de un CPU y un GPU[2].

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 del código. Cuando un componente de hardware que está físicamente separado del CPU 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 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 procesamiento de instrucciones aplicadas al campo de imágenes 2D y 3D, realizando cálculos con pixeles y texeles[23].

La tarjeta gráfica en su interior puede contener una cantidad de núcleos del 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[5]. 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 GPU

La arquitectura de las tarjetas gráficas ha experimentado ciertas evoluciones en su desarrollo para permitir a los programadores hacer un uso más eficiente de su poder de procesamiento. Las GPU incorporan:

- Memoria: cuentan con diferentes tipos de memoria y principalmente 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 flujos: La arquitectura de una GPU está diseñada con base en la programación de flujos, el cual involucra múltiples cálculos en paralelo para un flujo de datos[24].

- Flujo: Conjunto de elementos que tendrán un tratamiento similar.
- Kernel: Tratamiento aplicado a cada elemento del flujo.
- Thread: Tratamiento ejecutado por procesador aplicado a un elemento del flujo.

2.4.2 Arquitectura CUDA

CUDA es el acrónimo en inglés de Compute Unified Device Architecture (Arquitectura Unificada de Dispositivos de Cómputo), el cual es una arquitectura de hardware y de software que permite ejecutar programas en las tarjetas gráficas de la marca NVIDIA[3].

CUDA C es una extensión del estándar ANSI C con varios complementos del lenguaje para utilizar la programación heterogénea, añadiendo APIs sencillas para administrar los dispositivos e/s, memoria y otras tareas. También es un modelo de programación escalable que permite a los programas trabajar transparentemente con un número variable de núcleos de procesamiento.

Un programa en CUDA consiste en la mezcla de dos códigos: el código de anfitrión, que tiene que ver con lo realizado por el CPU, y el código de dispositivo, que ejecutará la GPU. El compilador de NVIDIA (nvcc) separa ambos códigos durante el proceso de compilación y durante la etapa de enlace las bibliotecas de CUDA se agregan a las funciones que irán al dispositivo para poder manipular completamente la tarjeta gráfica.

A la hora de hablar de su arquitectura se pueden identificar dos perspectivas dependiendo del nivel de abstracción que se requiera, la del programador o la del hardware.

2.4.2.1 Perspectiva del programador

La tabla 2.2 muestra los componentes principales de CUDA desde la perspectiva del programador, y la figura 2.5 muestra esquemáticamente como se conforma el *grid* de un *kernel*.

Muchas veces es necesario conocer el *id* tanto del *thread* como del *block* con los que se está trabajando. En la figura 2.5 se observa la configuración de la posición

Kernel	Funciones paralelas escritas en el programa que indican que operaciones se realizarán en la GPU.
Thread	Unidad mínima que ejecuta una instancia de un *kernel*. Tiene su propio *id*, su propio contador de programa, registros, memoria privada, entradas y salidas.
Block	La agrupación de *threads* que utilizan memoria compartida.
Grid	Arreglo de *blocks* que ejecutan el mismo *kernel*, leen y escriben datos en memoria global.

Tabla 2.2: Componentes de CUDA para el programador.

secuencial de cada elemento.

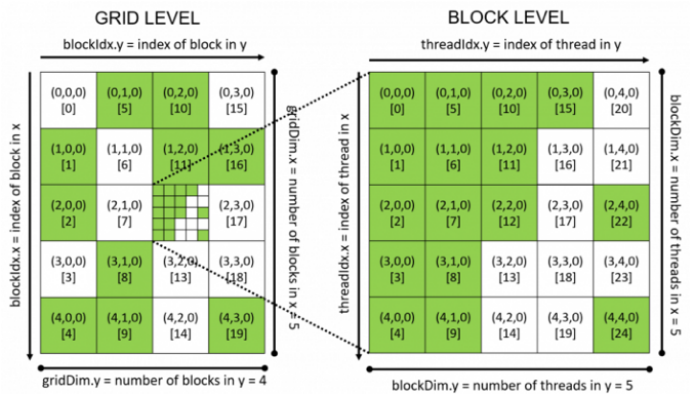

Figura 2.5: Representación de los componentes de un *grid*[3].

Para obtener ambos datos se aplica el algoritmo 2.1:

```
id_block = blockIdx.y * gridDim.x + blockId.x

id_thread = threadIdx.y * blockDim.x + threadIdx.x

```

Algoritmo 2.1: Transformación para obtener el *id* del *threads* y del *block*.

Existe una jerarquía de memoria para las variables que se utilicen, la tabla 2.3 muestra el lugar donde se almacenan y el alcance que tienen.

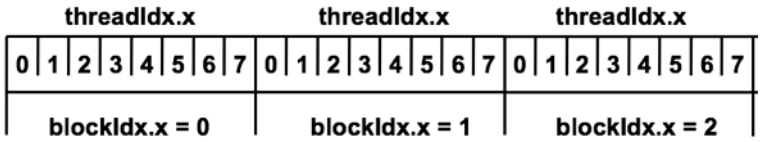

Figura 2.6: Orden de *threads* y *blocks* dentro de un *grid*[3].

Declaración	Memoria	Alcance	Tiempo de vida
int x	registro	thread	thread
int arreglo_x	local	thread	thread
__shared__ int shared_x	shared	block	block
__device__ int global_x	global	grid	aplicación

Tabla 2.3: Jerarquía de almacenamiento en dispositivo.

2.4.2.2 Perspectiva del hardware

Cada modelo de tarjeta gráfica contiene una cantidad variable de *SM* dependiendo de su tamaño y/o propósito. Cada uno puede ejecutar un número limitado de *TB* en concurrente (ver tabla 2.4. Cada vez que un *SM* ejecuta un *TB*, cada uno de sus *threads* se ejecuta al mismo tiempo, por lo tanto, para liberar los recursos ocupados y darle la oportunidad a un nuevo *kernel* de la cola de ejecución, es fundamental que todos los *threads* de ese *TB* concluyan completamente su ejecución [25].

Un *warp* es la entidad mínima ejecutable dentro de la GPU y es un conjunto de 32 *threads* dentro de un *TB*, por lo que todos los *threads* en su interior ejecutan la misma instrucción, y son seleccionados secuencialmente por el *SM* que los ejecutará. [26]

Warp	Agrupación de 32 *threads* que ejecutan la misma instrucción.
Thread Block (TB)	Conjunto de *warps* consecutivos.
Streaming Multiprocessor (SM)	División del GPU que ejecuta un conjunto de *TB*.
GPU	Conjunto de *SM* específico de la tarjeta GPU.

Tabla 2.4: Componentes de CUDA para el hardware.

Una vez que se lanza un *TB* en un *SM*, todos sus *warps* permanecen en ejecución

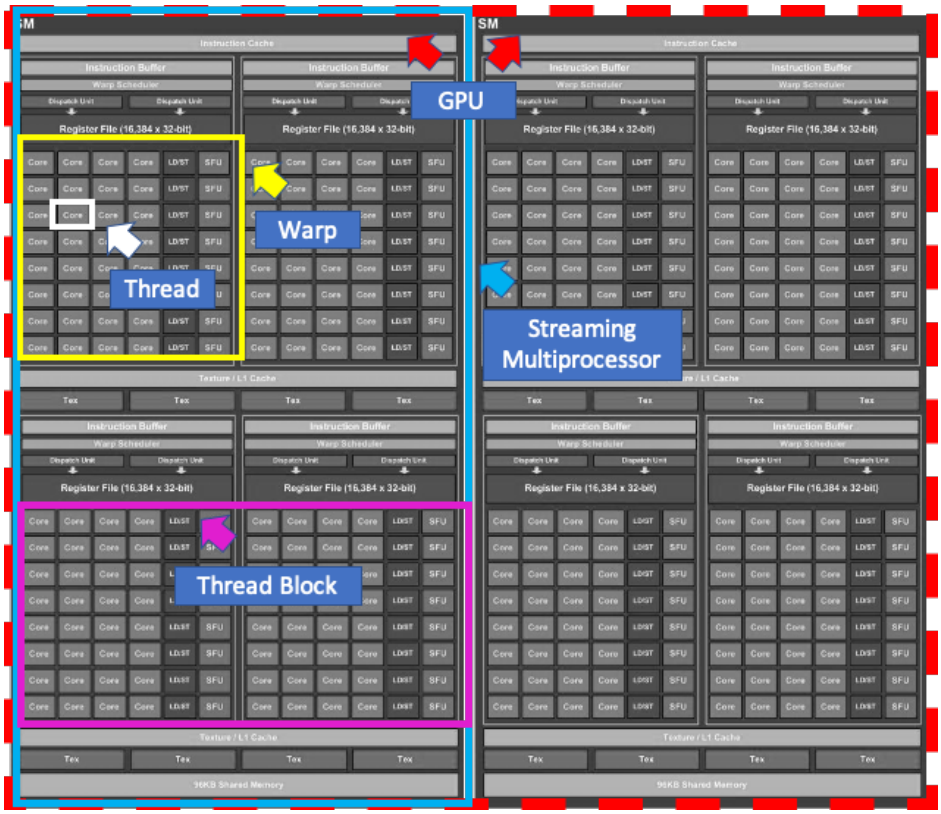

Figura 2.7: Componentes de la arquitectura en perspectiva del hardware (Basada en [4]).

hasta que todos sus *threads* completen su ejecución. Por lo que un *TB* nuevo no puede iniciarse dentro de un *SM* hasta que no haya un número suficiente de *warps* libres para ejecutar el nuevo *TB*.

La figura 2.7 muestra el diagrama de la arquitectura de una tarjeta gráfica desde la perspectiva del hardware. La tabla 2.4 enlista los componentes de la arquitectura.

2.4.3 Arquitectura Pascal

Pascal es el nombre de una versión de la arquitectura de tarjetas gráficas manufacturadas por la empresa NVIDIA.

Esta arquitectura tiene tres apartados significativos que ayudan a mejorar el ren-

dimiento de los sistemas. El primero es la memoria unificada, la cual proporciona un único espacio de direcciones virtuales para la memoria de el 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, como programador, no es necesario el preocuparse por administrar el intercambio de datos entre dos sistemas de memoria virtual diferentes[27].

En la figura 2.8 se observan tres tipos de código: el central es el código original que se ejecuta normalmente en un CPU. A la izquierda se presenta su versión en CUDA, y a la derecha se puede observar la versión en CUDA con memoria unificada. Con ello se puede observar que se optimiza el código para el manejo de la memoria entre CPU y GPU, además de que intrínsecamente se reduce el tiempo de transferencia de datos.

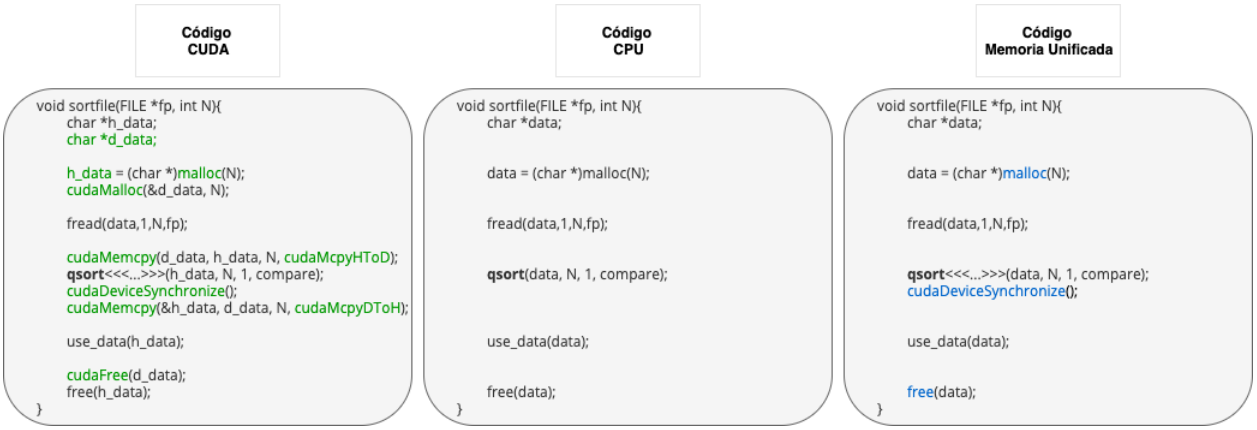

Figura 2.8: Comparación de directivas para manejo de memoria.

El segundo es la utilización de operaciones atómicas de memoria. Brindan a un *thread* la posibilidad de leer, modificar, y escribir memoria, mientras cualquier otra operación concurrente que intenta acceder a esa memoria es suspendida para evitar condiciones de carrera. Esta suspensión es muy costosa en tiempo de ejecución porque los accesos concurrentes se secuencializan y los caches se invalidan.

El tercero es la incorporación de computación *preemptive* a nivel de instrucción, donde el propio sistema evita que se monopolice el uso de la tarjeta por las instrucciones de algún *kernel*. Puede permitir que las tareas se ejecuten por un tiempo prolongado o esperar hasta que ocurra alguna condición para iniciarla. El programador no tiene acceso a esta función y el hardware es el encargado de administrarla[27].

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 cómputo de propósito general en unidades de procesamiento gráfico o *GPGPU* es utilizado para acelerar el parte del procesamiento intensivo realizado tradicionalmente por el CPU (ver figura 2.9), donde la GPU actúa como un coprocesador que puede aumentar la velocidad del trabajo [28].

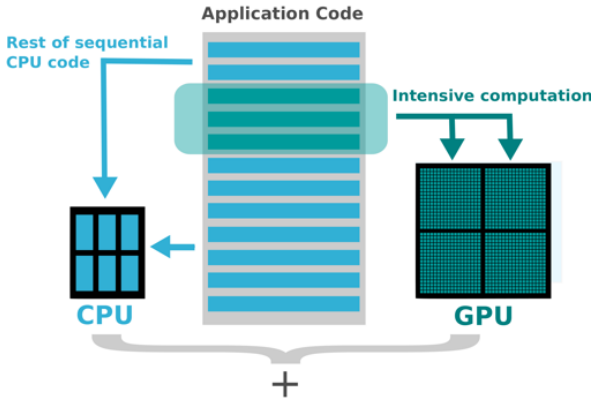

Figura 2.9: Aceleración de programas en GPU[5].

En el caso de la arquitectura Pascal la unificación de los espacios de memoria facilita el *GPGPU* ya que no hay necesidad de transferencias explícitas de memoria entre el anfitrión y el dispositivo.

2.4.5 Clasificación de soporte preemptive para GPU

Existen diversas clasificaciones en las que se pueden agrupar las soluciones para dar soporte a la planificación de tareas *preemptive* [29, 30, 31, 32], una de ellas se basa en el tipo de implementación:

- ***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 *sub-kernels* dividiendo sus *grids* en fragmentos más pequeños. Es decir, en vez de lanzar todos los *Thread por Block* (*TB*) de un *kernel* a la vez, sólo se van lanzando fracciones de éste. Este enfoque es útil para núcleos con muchos *TB*, donde cada *TB* tienen un tiempo de ejecución corto. Cuando se tienen diversas especificaciones a resolver por un *kernel* y estas son necesariamente tareas secuenciales por la dependencia de resultados, se implementan puntos *preemptive* para dividir el *kernel* en tareas a completar.
- ***Partición en Trabajos.*** Esta técnica es parecida a la anterior, pero los fragmentos se ven como trabajos, se invocan tantos *TB* como sea posible tenerlos activos al mismo tiempo. Aquí 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 se obtiene de acuerdo a la forma en que se planifican las tareas:

- ***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 memoria.*** Se tiene un administrador de memoria que verifica si es posible asignar memoria para una nueva tarea, o cual de las que actualmente se encuentra en ejecución ha superado su espacio definido.
- ***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.
- ***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.

Finalmente, se puede encontrar una clasificación enfocada en la forma en que se implementa la planificación 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; sin embargo, su utilización muchas veces no está permitida por los administradores.

2.5 Sistemas embebidos

Un sistema embebido es un sistema de cómputo diseñado para realizar tareas dedicadas, su mayor reto es realizar tareas específicas donde la mayoría de ellas tengan requerimientos de tiempo real [33].

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 eficiencia tanto energética como espacial.

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 [34] ya sea de planificación o memoria, lo que permite proponer un *framework* que ayude a la administración de sus características.

2.6 Caso de Estudio: Jetson TX2

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 este tipo de dispositivos, así como la programación en arquitectura Pascal.

En la figura 2.10 se muestra el diagrama de bloques de la arquitectura del sistema Jetson TX2.

2.6.1 Jetson TX2

Las especificaciones del sistema están descritas en la tabla 2.5.

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.
- Actualización de bibliotecas compatibles.
- Configuración de área local y conexión a través de una 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.

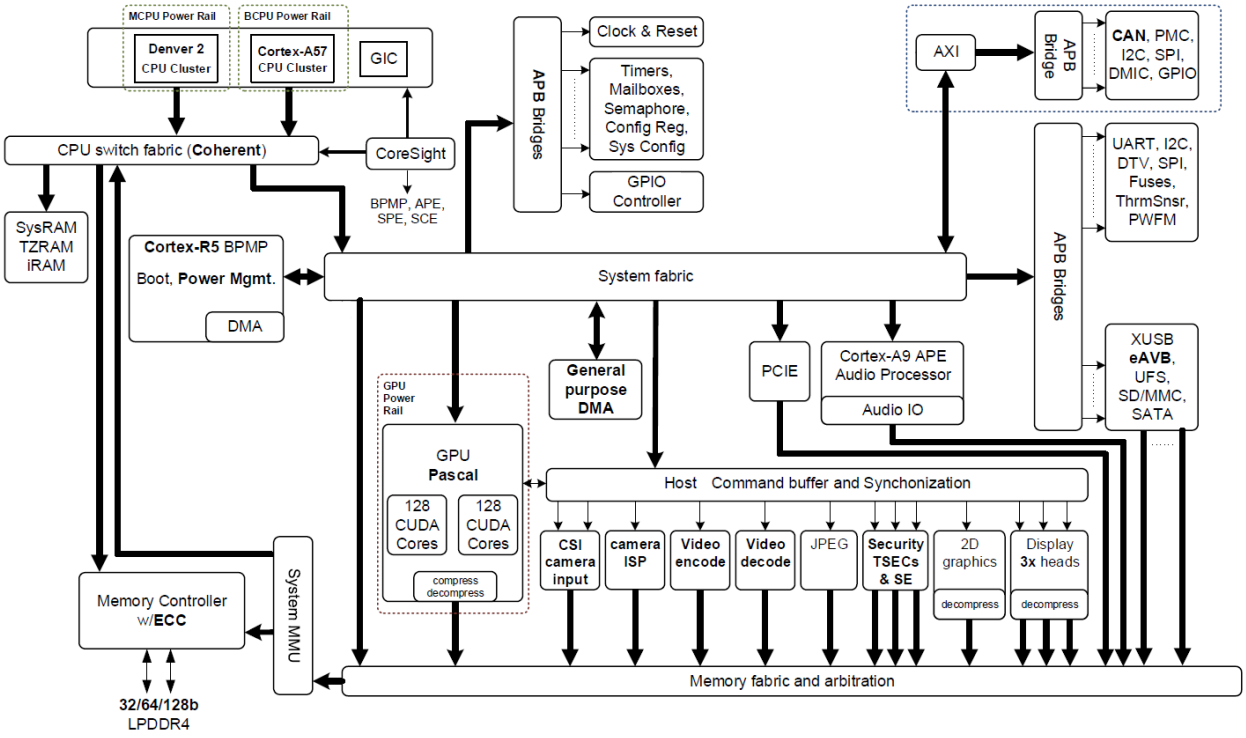

Figura 2.10: Diagrama de la arquitectura del sistema Jetson TX2[6].

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 clústeres 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 128 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 Almacenamiento Flash	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. Reproducción de 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 2.5: Especificaciones del sistema Jetson TX2[7].

CAPÍTULO
3

Trabajo relacionado

Durante la revisión de la literatura referida a este estado del arte, se encontró que existe muy poca bibliografía sobre el tema debido a que las empresas fabricantes de las tarjetas gráficas no liberan suficiente información al público en general, ya que sus diseños no están documentados o se describen a tan alto nivel que no revelan los detalles técnicos cruciales.

Uno de los trabajos mas relevantes lo realizó Amert et al. [35] con base en la documentación y pruebas de caja negra sobre la arquitectura Pascal. Como caso de estudio utilizaron el sistema NVIDIA Jetson TX2, que es precisamente el que se tuvo disponible al realizar esta tesis. El estudio tiene como objetivo dar las pautas para realizar normas oficiales de seguridad a los sistemas embebidos heterogéneos y así poder certificar su uso en ambientes críticos.

Cada GPU contiene un arreglo de *Streaming Multiprocesor* (*SM*) dependiendo del modelo o arquitectura, en donde cada *SM* puede procesar concurrentemente una misma cantidad de *threads*. Como ya se mencionó anteriormente, las empresas que producen las tarjetas no revelan mucho sobre las peculiaridades de cómo se realiza el reparto de recursos para ejecutar las peticiones. Únicamente se conoce que la GPU distribuye los *blocks* pendientes dentro de los *SM* en grupos de 32 *threads*, estos grupos son llamados *warps*. Un *warp* es una entidad planificable de un *SM*. Un *block* puede ser asignado únicamente a un *SM* si este tiene suficientes recursos para recibirlo, si no, esperará hasta que pueda ser lanzado. El cómo realiza esta espera tampoco es revelado por las compañías.

El sistema Jetson TX2 cuenta con 2 *SM*, y cada uno con 128 cores [36]. Cada *SM* puede procesar un máximo de 64 *warps* en concurrente[37] o:

$$64\ warps * 32\ threads = 2048\ threads\ por\ SM \tag{3.1}$$

Lo que resulta en un total de 4096 *threads* concurrentes en ambos *SM*.

Por ejemplo, en la tabla 3.1 se muestra la información de diversos escenarios en los que se desea agregar *kernels* para procesar en la tarjeta gráfica.

Escenario	Kernel	Blocks	Threads por block	Threads Totales	Blocks SM0	Warps SM0	Blocks SM1	Warps SM1	Threads utilizados
A	K1	4	512	2048	4	64	0	0	2048
B	K2	16	265	4096	8	64	8	64	4096
C	K3	9	350	3150	5	55	4	44	3168

Tabla 3.1: Asignación de un solo *kernel* a los *SM*.

En el escenario **(A)** (ver figura 3.1) se tiene el *kernel* **K2** con 4 *blocks* y cada uno de 512 *threads*. La primera condición para poder agregarlo recae en que la totalidad del *kernel* debe poder ejecutarse en concurrente en la tarjeta gráfica durante un instante de tiempo. En este caso, recordando que un *SM* tiene un máximo de 64 *warps* disponibles, y el *kernel* justamente está constituido por esa cantidad, por lo que el planificador de hardware de la tarjeta asigna secuencialmente los *blocks* a las localidades.

En el escenario **(B)** se agrega un *kernel* que comprende 16 *blocks* y cada uno consta de 256 *threads* por *block*. En este caso, al lanzarlo se utilizará la totalidad de los recursos de la GPU y los *blocks* se irán asignando secuencialmente empezando desde la primer localidad del **SM0** y así sucesivamente hasta que se terminen sus recursos. Cuando esto suceda, se continuará con las primeras localidades de **SM1** hasta terminar de lanzar todos los *blocks* del *kernel*. En caso de que existan recursos para colocar *blocks* de un *kernel* al final de **SM0** pero no los suficientes para colocarlo al inicio de **SM1**, el *kernel* no se podrá lanzar, y el planificador por hardware saltará la tarea y asignará otra que ocupe menos recursos.

Finalmente, en el escenario **(C)** deberían ser 54.6 *warps* en el **SM0** utilizando 1750 *threads*, pero en realidad se utilizan 55 *warps*, con lo que el trabajo de 10 *threads* y 9 *warps* se desperdician. En el **SM1**, deberían ocuparse 43.75 *warps*, pero se usan

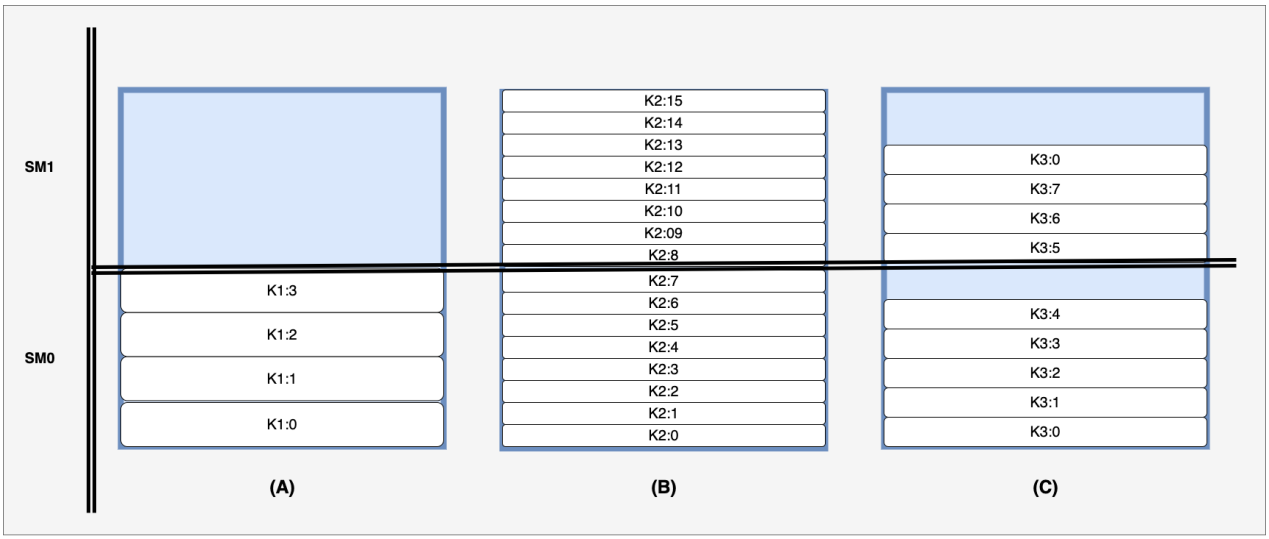

Figura 3.1: Diagrama de asignación de un *kernel* a los *SM*.

44, dejando 8 *threads* y 20 *warps* sin poderse utilizar.

Habitualmente en sistemas *GPGPU* se requiere lanzar más de un *kernel* a la vez, en la tabla 3.2 se tienen 3 situaciones en las que esto podría ocurrir.

Escenario	Kernel	Blocks	Threads por block	Threads por kernel	Threads en concu-rrente	Warps	Warps apor-tados a SM0	Warps SM0	Warps apor-tados a SM1	Warps SM1
D	K1 K2	3 10	512 256	1536 2560	4096	48 80	48 16	64	0 64	64
E	K1 K2 K3	1 1 2	1024 768 256	1024 768 512	2304	32 24 16	32 24 8	64	0 0 8	8
F	K1 K2 K3	1 1 1	768 1536 768	768 1536 768	3072	24 48 24	24 0 0	24	0 48 0	48

Tabla 3.2: Asignación de varios *kernels* a los *SM*.

En el escenario **(D)** (ver figura 3.2) se desea asignar dos *kernels*, uno de 3 *blocks* y otro de 10. Se debe tener en cuenta la restricción que marca que un *kernel* sólo puede ser asignado si existen suficientes recursos para *warps* en secuencia. La primer tarea utiliza 48 *warps*, y la segunda 80, por lo que en **SM0** se pueden asignar los 48

warps del primer *kernel* más 16 del segundo, y en el **SM1** se colocan los 64 restantes, ocupando así la totalidad de localidades de la GPU.

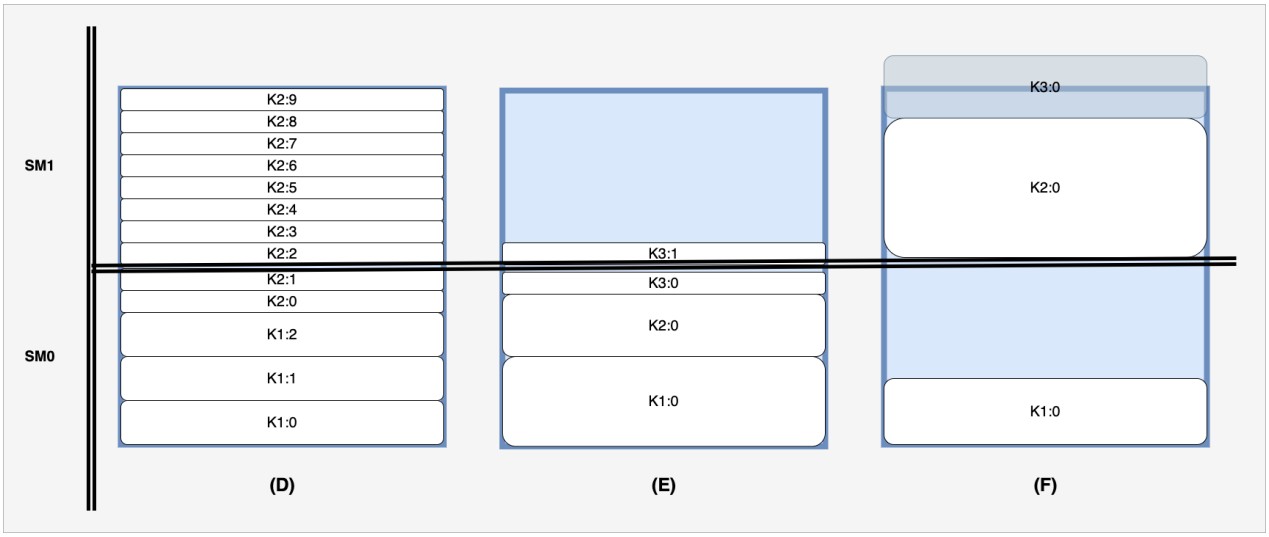

Figura 3.2: Diagrama de asignación de varios *kernels* a los *SM*.

Continuando con el caso **(E)**, ahora se tienen 3 *kernels*, dos de un *block* con 1024 *threads* y otro con 768, respectivamente y uno con 2 *blocks* de 256 *threads*. En conjunto ocupan 72 *warps* en total, por lo que se pueden asignar los 32 *warps* del **K1** al **SM0**, justo después los 24 del **K2**. Con esto aún sobran 8 *warps* que sirven perfectamente para ejecutar la mitad de *warps* del **K3** y, como las localidades del inicio de **SM1** están disponibles, se sigue con la asignación de los 8 restantes.

El escenario **(F)** presenta un caso en que se tienen 3 *kernel* que en conjunto representan 3072 *threads*, lo que nos hace pensar que muy bien pueden ser ejecutados en concurrente dentro de la GPU, pero el orden de lanzamiento influye de primera mano en la repartición de los recursos. Primero se lanza el *kernel* **K1** que constituye 24 *warps*, después se recibe la petición de **K2** con 48, pero como no es posible asignarlo en el **SM0**, se le asignan las localidades de **SM1**. Finalmente, desea despachar a **K3** con 24 *warps*, pero como la asignación de recursos se realiza de forma secuencial, se empieza a preguntar justo en la siguiente localidad de la última asignada, con lo que ya no es posible lanzar esa tarea, y se deberá esperar a que terminen su ejecución las tareas actuales. Aunque en el **SM0** sobran recursos que bien podrían aprovecharse, la documentación actual no permite realizarlo, por ello se recurrió a una solución en

la sección 4.5.1.

Ahora bien, teniendo una referencia por medio de pruebas de software, se puede conocer un poco sobre el funcionamiento de la asignación de *kernels* a la tarjeta gráfica. Se puede afirmar que es posible gestionar las peticiones de recursos a la GPU, y el programador debe cuidar que tamaño de los *blocks* de *threads* sean múltiplos de 32 o al menos potencia de 2 y mayor que 32 para optimizar los recursos de la tarjeta gráfica y así ejecutar la mayor cantidad posible de tareas concurrentemente y optimizar el uso del recurso.

Para poder utilizar varias aplicaciones con sistemas en tiempo real complejos es necesario utilizar técnicas de implementación *preemptive*[32]. 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[30].

La clasificación de la planificación de tareas *preemptive* esta compuesta de diversos tipos dependiendo de sus técnicas de implementación, cómo se describe en la sección 2.4.5.

Las soluciones basadas en hardware son costosas, ya que se debe desarrollar y construir un dispositivo que auxilie con el cambio de contexto. Por ejemplo, en el artículo Tanasic et al. [38] 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*.

Wang et al. [39] ha propuesto la utilización de extensiones de hardware mediante el intercambio equitativo de recursos entre los núcleos de procesamiento, realizando un cambio de contexto al aplicar 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 las nuevas tareas.

Otra solución la presentan presentan Li, Nyland y Zhou [40], donde se desarrolla un compilador 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.

Zhou, Tong y Liu [31] implementan *GPES*, una serie de funciones para realizar particiones de *kernel* y de datos, esto realizando *sub-kernels* 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 está 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 llevarlo a cabo se realiza una transformación fuente a fuente apoyándose en la partición de la transferencia de datos.

Lee et al. [41] propone un *framework* de planificación que parte los *kernels* de la GPU y genera secuencias de lanzamiento en *sub-kernels* dinámicamente para entrar en el modo *preemptive* con la implementación de un divisor de carga de trabajo y de un planificador de tareas. El divisor de carga de trabajo fracciona el *kernel* GPU en múltiples *sub-kernels* en tiempo de ejecución para implementar el modo *preemptive*. Dependiendo del estado actual del sistema y de la prioridad, el divisor de carga de trabajo decide el número y el tamaño de cada *sub-kernel*.

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

Lin et al. [42] describe el *framework* EffiSha que se basa en un entorno de scripts que permite convertir los *kernels* automáticamente a modo *preemptive*. Esta solución consta de componentes que funcionan tanto en tiempo de compilación como en el de ejecución. En tiempo de compilación realiza una transformación de fuente a fuente que transforma un programa para la gestión y planificación oportuna de su tiempo de ejecución. En el código del CPU, reemplaza las llamadas a función de la GPU con las del API de EffiSha, así modifica los *kernel* GPU para que puedan acelerar el cambio o drenado de contexto durante el tiempo de ejecución. También se analizan e identifican aquellos datos que no se volverán a utilizar después de la restauración de contexto, con lo que ahorra el tiempo de las transacciones de memoria innecesarias.

La fase de ejecución consiste en un daemon en el lado del anfitrión y un proxy de éste en el lado de la dispositivo. Dicho daemon gestiona el momento en que los *kernels* deben comenzar, reanudarse o detenerse en la GPU, y dependiendo de la acción se notifica al proceso del CPU que fue lanzado.

Como muestran los resultados del trabajo EffiSha, esta solución funciona bien para *kernels* con tiempo de ejecución corta porque 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 trabajo presentado por Kato et al. [32] es el primero 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 y 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 la GPU en cualquier momento, y así evitar interferencias de rendimiento causadas por lanzamientos concurrentes.

La primera característica de este *framework* es que se basa en transacciones de datos *preemptive*, por lo que los tiempos de bloqueo están limitados 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 uso elevado de memoria global.

El trabajo de Calhoun et al. [15] se basa en preguntar continuamente si ha terminado el *quantum* de una tarea, en caso afirmativo detiene la ejecución e ingresa la siguiente. Las tareas son almacenadas en una cola, por lo que todas tienen la misma prioridad durante la vida del sistema. Se propone un esquema de puntos de control donde se almacena el estado de un *kernel* en ejecución en la memoria del CPU en vez de la GPU. Para ello, se apoya de una estructura donde se almacena el contexto completo de la tarea. Para disminuir la latencia entre cada punto *preemptive*, se le avisa al *framework* que se debe tener preparada la estructura de seguridad con directivas *pragma*, antes y después de la ejecución parcial de un *kernel*. Para ello fue

necesario implementar un analizador sintáctico que ayudara al compilador a verificar las modificaciones del código.

Reano et al. [43] 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 la 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 la 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 *thread* 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*)[44], esto para evitar la creación de un nuevo contexto que consuma memoria.

La tarea es modificada únicamente al llamar de manera explícita las funciones de la biblioteca del cliente para previamente asignar la memoria requerida al GPU. Esto acarrea una gran desventaja al considerar tareas donde no siempre es posible conocer la memoria requerida total de GPU, ya que 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 de la GPU se puede llegar a utilizar completamente, con lo que podrán requerir más tiempo para completar la ejecución o directamente lanzar un error de desbordamiento de memoria en tiempo de ejecución.

El trabajo de Kang et al. [45] presenta una técnica para la ejecución en GPUs llamada *"Planificación de recursos espaciales 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 trabajo de Nicola et al. [46] utiliza sistemas embebidos heterogéneos muy popu-

lares en el campo de los coches autónomos. Realiza una modificación a nivel de API para que el hipervisor de la arquitectura de programación (CUDA y OpenGL) directamente realice una planificación (apoyada con EDF) de *kernels* a nivel de hardware. Inicia escuchando los canales del anfitrión en búsqueda de peticiones de lanzamiento, una vez que arma una lista de peticiones, de manera asíncrona implementa políticas para permitir un tiempo de ejecución a cada tarea y así implementar el modo *preemptive* a las peticiones de recursos. Este trabajo se apoya en la computación *preemptive* presente en la arquitectura Pascal (ver sección 2.4.3) ya que el hipervisor puede ejecutar las instrucciones de los diferentes *kernels* con el orden obtenido de la planificación y dependiendo de la granularidad de las mismas.

El segundo trabajo que utiliza algoritmos para la planificación de tareas en tiempo real, hasta el momento de la revisión del estado del arte es GPUart presentada por Hartmann et al. [29]. Permite la implementación *preemptive* dentro de los *TB* y cada uno de estos *sub-kernels* se pueden planificar bajo las políticas de EDF y de aquellos algoritmos que mantengan la prioridad de las tareas fijas.

GPUart se centra en las GPUs embebidas, es decir, en las GPU que se colocan en la misma placa que el CPU. Esto porque permiten tener cero copias de memoria, lo que hace que las transferencias entre CPU y GPU sean nulas al compartir físicamente una memoria común. Por ello, GPUart no considera la planificación de transferencias de memoria a través del acceso directo a memoria (DMA).

A continuación se muestra la tabla 3.3 con los trabajos relacionados y la clasificación en la que entran dependiendo de las características con que se ejecutan, así como su referencia en el bibliografía.

Ref.	Artículo	Clasificación por implementación	Clasificación por planificación	Clasificación por Modificación
[38]	**Enabling preemptive multiprogramming on GPUs**	Basado en Hardware: Añade registros para almacenar contexto	Colas masivas en paralelo	Modificación del API
[39]	**Simultaneous Multi-kernel GPU: Multi-tasking throughput processors via fine-grained sharing**	Basado en Hardware: Selector de núcleos de procesamiento	Administración dinámica de los núcleos de procesamiento	Modificación del API
[40]	**Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching**	Basado en Hardware: Analizador del ciclo de vida de registros	Administración dinámica de memora	Modificación del API
[31]	**GPES: A preemptive execution system for gpgpu computing**	Basado en Software: Partición de *kernel*	Administración dinámica de memora	Modificación del API
[41]	**Run-Time Scheduling Framework for Event-Driven Applications on a GPU-Based Embedded System***	Basado en Software: Partición en trabajos	Administración dinámica de la memoria	Modificación del código fuente
[42]	**Effisha: A software framework for enabling efficient preemptive scheduling of GPU**	Basado en Software: Entorno de scripts	Colas masivas en paralelo	Modificación del API y código fuente
[32]	**RGEM: A Responsive GPGPU Execution Model for Runtime Engines**	Basado en Software: Partición de *kernel*	Colas masivas en paralelo	Modificación de código fuente
[15]	**Preemption of a CUDA Kernel Function**	Basado en Software: Partición de kernel	Colas masivas en paralelo	Modificación del API
[43]	**Intra-Node Memory Safe GPU Co-Scheduling**	Basado en Software: Partición de kernel	Administración dinámica de la memoria	Modificación del API
[45]	**Priority-driven spatial resource sharing scheduling for embedded graphics processing units**	Basado en Software: Partición en trabajos	Administración dinámica de los núcleos de procesamiento	Modificación del API
[46]	**Deadline-Based Scheduling for GPU with Preemption Support***	Basado en Software: Partición en trabajos	Planificación por prioridad	Modificación del API
[29]	**GpuArt: An application-based limited preemptive gpu real-time scheduler for embedded systems***	Basado en Software: Partición de kernel	Planificación por prioridad	Modificación de código fuente

Tabla 3.3: Matriz de clasificación de trabajos relacionados.

** Artículos que fueron diseñados específicamente para sistemas embebidos.*

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. Actualmente existe una gran variedad de planificadores de tareas sobre CPU, la propuesta de esta tesis es presentar el diseño de un *framework* que ayude a planificar tareas *preemptive* a ejecutarse sobre la GPU. El estudio de la planificación de las tareas del CPU queda fuera del alcance de esta tesis.

Aunque se tomó como base el sistema embebido heterogéneo NVIDIA Jetson TX2, el diseño puede ser aplicado a otros dispositivos, siempre y cuando cumplan con la característica descritas en la sección 2.4.3.

4.1 Descripción general

La solución propuesta se basa en las clasificaciones:

- **Por implementación**: *Partición de kernel basada en software.*
- **Por planificación**: *Planificación por prioridad.*
- **Por modificación**: *Modificación de código fuente.*

En la Figura 4.1 se muestra un diagrama de bloques sobre la arquitectura del *framework* propuesto. Cada uno de los *blocks* agrupa las bases necesarias para el funcionamiento del *framework*.

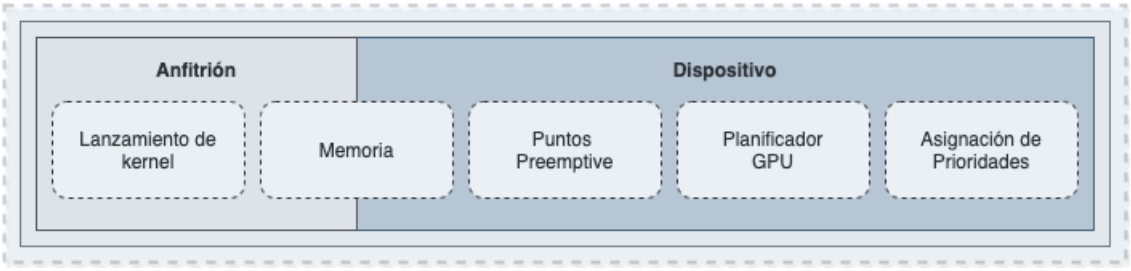

Figura 4.1: Esquema del *framework* para la planificación de tareas *preemptive* en sistemas embebidos heterogéneos.

El *framework* está dividido en dos zonas de implementación, la zona anfitrión contempla aquellas actividades que son propias del CPU, como lo es el protocolo de **Lanzamiento de kernel** (ver sección 4.2). También contempla el manejo de memoria, en específico de la tarjeta Jetson TX2 que utiliza una arquitectura Pascal (ver sección 2.4.3), éste cuenta con una memoria unificada lo cual elimina las copias de memoria entre el anfitrión y el dispositivo, cuya propuesta del módulo **Memoria** (ver sección 4.3) pertenece a ambas zonas, donde la parte relevante del *framework* es el cambio de contextos de cada tarea cuando se realiza su suspensión.

En la zona dispositivo se encuentra el módulo **Puntos Preemptive** (ver sección 4.4). Como su nombre lo indica, se plantea la forma en que el *framework* implementa las suspensiones y reactivaciones de las tareas una vez alcanzado cada uno de los puntos.

Un componente fundamental del *framework* es el módulo **Planificador GPU** (ver sección 4.5), éste proporcionan las pautas para el encolado de las tareas que se ejecutarán en un determinado momento en el dispositivo. Pero para poder realizar dichas pautas, se plantea el módulo **Asignación de prioridades**, (ver sección 4.6) el cual se encargará de ordenar un conjunto de tareas definiendo sus prioridades dentro de la cola.

Como se detallará más adelante, el *framework* como solución no es transparente al programador y es necesaria la modificación del código fuente.

El diseño de la asignación de prioridades por el planificador está fuera del alcance de esta tesis (ver sección 4.6), pero se puede implementar con la mayoría de la gama de algoritmos del estado del arte (ver sección 2.2.3). Para el estudio de este trabajo

se usarán indistintamente los términos *kernel* y tarea.

4.1.1 Precondiciones necesarias

- La precondición más importante radica en que el *framework* debe ser implementado en una aplicación que funciona correctamente, ya que se realizará una modificación en su código fuente para la implementación del modo *preemptive*.
- No se permite la memoria dinámica ni compartida entre *kernels*.
- No se permite el llamado a funciones no rastreables (que no existen).
- El *quantum* deberá ser un parámetro de diseño, es decir establecido para todo el sistema embebido en un archivo de configuración.
- El número de *threads* por *blocks* debe ser menor o igual a la cantidad de *threads* disponibles en cada *SM*.
- Los contextos (conjuntos de variables) de cada *kernel* deben coexistir en la memoria al mismo tiempo para que se puedan ejecutar y suspender en cada punto *preemptive*.
- Todas las tareas que soliciten recursos al planificador deben ser *preemptive* para que en dado caso puedan ser suspendidas, si así lo requiere el balanceo de carga.
- Para evitar plazos vencidos, el conjunto de tareas de la aplicación debe ser planificable.

4.2 Lanzamiento del kernel

Dentro de las precondiciones antes mencionadas, existen dos de gran importancia para este módulo. La primera, es que el *framework* planificará un número estático de *kernels* conocidos desde el inicio y, la segunda, es que el código fuente esté disponible para su adecuación al *framework*.

Teniendo en cuenta que habrá aplicaciones en ejecución en forma concurrente (Figura 4.3), todas las tareas deben alcanzar un punto de petición de recursos de GPU para que el planificador permita dar el orden de lanzamientos.

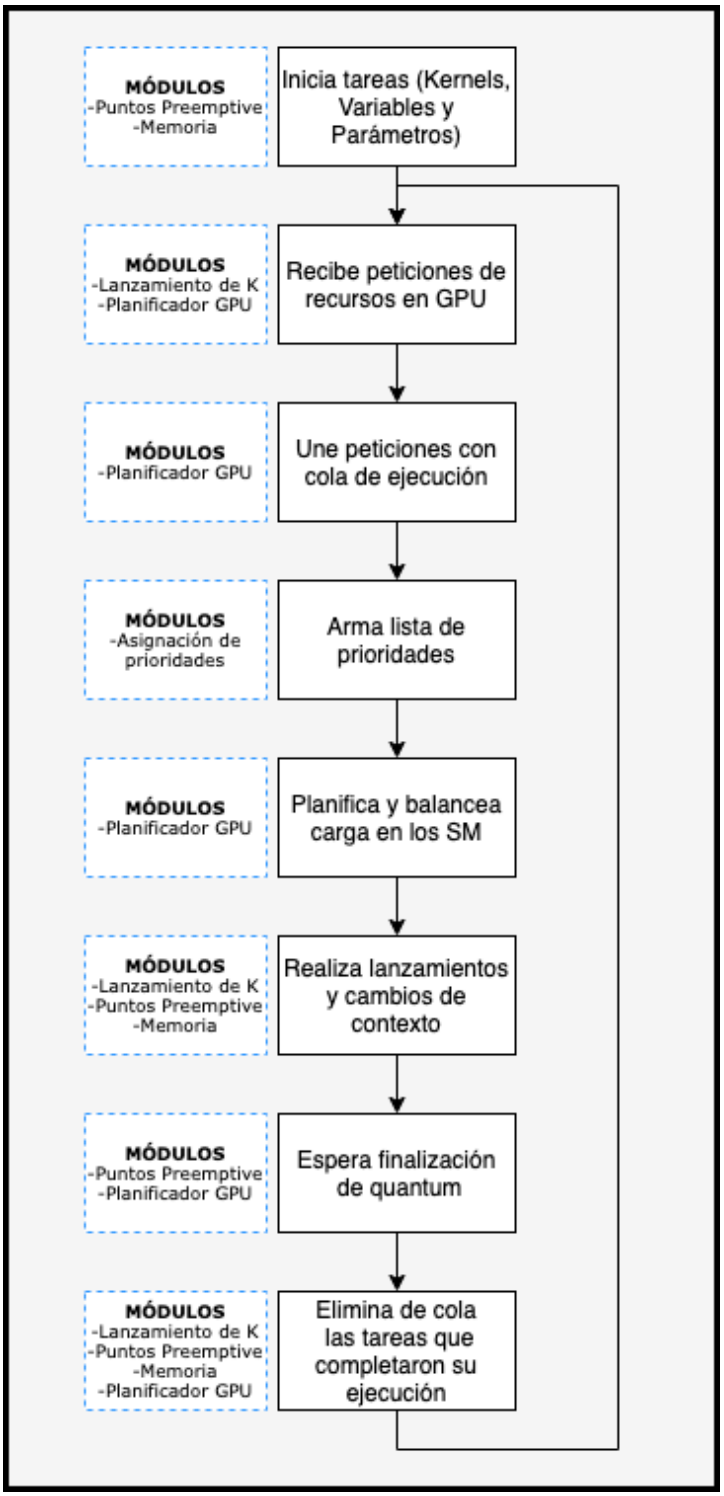

Figura 4.2: Diagrama del flujo del *framework*.

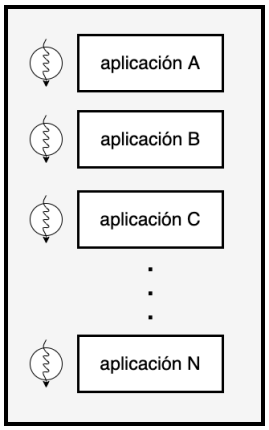

Figura 4.3: Aplicaciones en ejecución concurrente en el CPU.

Al código de cada una de las aplicaciones (ver algoritmo 4.1) se le debe añadir una serie de variables y parámetros extra para que el contexto pueda ser almacenado al alcanzar una suspensión *preemptive* y el planificador GPU eventualmente programe su ejecución.

Entre las variables a incorporar están la definición de una estructura *backup* específica del *kernel*, también se debe indicar la duración del *quantum* con *quantum_ time*, una bandera de control para verificar si ya ha expirado el *quantum* y una bandera que indicará si ya se ha ejecutado completamente el *kernel*. Finalmente, el planificador dará permiso de que se ejecute el *kernel* con la variable de control *continuar_ eje*.

Ahora bien, una vez que se han definido las variables de control, se debe implementar un ciclo que terminará hasta que el *kernel* sea completado. Dentro se inicializa *quantum_ expirado* en ***false*** para indicar que se tiene tiempo de ejecución.

La bandera *continuar_ eje* será modificada por el planificador para indicar la ejecución o suspensión del *kernel*. Una vez que sea planificado para su ejecución, se lanzará el *kernel* y se esperará el tiempo definido para el *quantum*.

Finalizado el plazo del *quantum*, para poder determinar si un *kernel* ha terminado completamente su procesamiento, se auxilia de la función *kc* (Algoritmo 4.2). Este revisa el parámetro *estado* de la estructura *backup* y se pregunta si el estado de todos los *blocks* es ***TERMINADO*** (ver sección 4.4). Si todos han terminado *kc* regresa ***true***, el *kernel* termina saliendo del ciclo *while*.

```
. . .
  //Declaración del backup
  Backup backup;
  //Duración máxima del quantum en microsegundos
  int quantum_time = #Duración del quantum
  bool* suspension_pree;
  bool* continuar_eje;
  bool* kernel_completado;
  //Solicita recursos en GPU
  agrega_kernel_planificador(blockDim,gridDim,continuar_eje,
   kernel_completado);

  /* Ejecución del kernel */
  while(!kernel_completado){
    quantum_expirado = false;

    if(continuar_eje == true){

      kernelFunction<<<blockDim,gridDim >>>(a, ... , resultados,
                            backup,continuar_eje);
      /*Espera el tiempo del quantum */
      usleep(quantum_time);
      quantum_expirado = true;
      cudaDeviceSynchronize();
      kernel_completado=kc(backup.estado);
    }

  }
. . .
```

Algoritmo 4.1: Algoritmo para lanzamiento del *kernel* en el lado del anfitrión.

```
bool kc(int* estado){
  for (int i = 0; i < gridDim; i++)
    if (estado[i] != TERMINADO)
      return false;
  return true;
}
```

Algoritmo 4.2: Función *kernel* completo.

4.3 Memoria

4.3.1 Almacenamiento del contexto

Es necesario crear una estructura de datos que guarde las copias de seguridad de los datos pertinentes que en conjunto formen el contexto de un *kernel*.

Todos los parámetros y variables que se encuentren dentro de una función *kernel* deben almacenarse en memoria, por lo que para cada *kernel*, se debe crear una estructura *ad hoc*.

```
1  struct Backup{
2
3    /* Variables locales */
4    int temp1[blockDim*gridDim];
5    int temp2[blockDim*gridDim];
6    . . .
7    int tempn[blockDim*gridDim];
8
9    /*Contadores*/
10   int i[gridDim];
11   int j[gridDim];
12   . . .
13   int k[gridDim];
14
15   /* Último punto de control */
16   int estado[gridDim];
17
18 };
```

Algoritmo 4.3: Estructura *backup* para almacenar el contexto.

La estructura *backup* (ver algoritmo 4.3) almacena tres tipos de variables, primero todas aquellas variables locales necesarias para resolver el problema original del *kernel*. Debido a que estas variables son individuales por *thread*, debe guardarse una copia de cada *thread* por cada *blocks*. Esta solución es muy costosa, por lo que se recomienda que la utilización de estas variables sea mínima o nula. En muchos casos, podría almacenarse su contenido directamente en alguna de las variables *resultado* que se pasaron como parámetro.

El segundo tipo de variables es de tipo contador. Dependiendo del cálculo que se esté realizando, muchas veces se deberán paralelizar *estructuras for* sin dependencia

de datos. Por esta razón, es posible que después de un cierto número de iteraciones se pregunte por el estado del *quantum* y, en ese momento, se realice la suspensión *preemptive* para todos los *threads* de un *blocks*. Como todos llegaron a ese punto, simplemente, se puede guardar un valor del contador. En caso de que se estén utilizando contadores que son propiamente controlados por un punto de verificación de *quantum*, se deberá utilizar el formato de variable local.

Finalmente, se incluyen variables adicionales que nos ayudan a almacenar el estado general de un *block*, este valor es calculado por el *thread0* de cada *block*.

4.3.2 Variables compartidas

Al momento de realizar una solución de *GPGPU*, se tiene que considerar que existirán variables que deben mantenerse visibles tanto para el CPU como para el GPU. En el algoritmo 4.1 de la sección 4.2 se tienen ciertas variables que son compartidas entre ambas unidades de procesamiento.

Aparte de los parámetros que originalmente tiene la función *kernel*, se agregan dos más: una estructura *backup*, que almacena el contexto cuando se presenta una suspensión *preemptive* y la bandera *quantum_ expirado*, que nos indica si ya terminó el tiempo máximo de ejecución. Como se tiene la memoria unificada, ambos parámetros existirán en la memoria global para que estén disponibles para ambos procesadores.

4.4 Puntos preemptive

La estrategia general en una aplicación acelerada por cómputo gráfico es implementar más de una función *kernel*. Pero en el momento de ejecutar varias aplicaciones en la GPU puede haber algunas que se mantengan monopolizando los recursos, causando así un retraso en la ejecución general de todo el sistema y sus aplicaciones. Una maniobra de diseño es implementar puntos de control *preemptive* para tener tiempos de ejecución uniformes independientemente de la duración de cada uno de los diferentes *kernels*.

Este módulo permite gestionar la actividad de un *kernel* a nivel de aplicación, aquí se marca la pauta de los puntos exactos donde se podrá realizar la administración del contexto de una tarea en ejecución y contará con tres casos principales: si se está iniciando, si está en ejecución o si ha terminado, con esto se podrán liberar

threads del GPU para dar oportunidad a otros *kernels* de consumir recursos.

Se propone una serie de puntos de control *preemptive* que se incluirán explícitamente dentro del código de las aplicaciones que se desean implementar en modo *preemptive*, definiendo tres estados iterativos del ciclo de vida de un *kernel* a)inicio, b)ejecución y c)finalización.

El objetivo de este módulo es resguardar una copia del contexto actual en una estructura de datos llamada *backup* cada que se alcance algún punto de control *preemptive* dentro de un *kernel* y sea necesario detener su ejecución. De esta manera, cuando se presente una nueva oportunidad de ejecución sea reanudado justo como si no se hubiera detenido.

Una vez que una tarea, independientemente del momento de su ciclo de vida en que se encuentre, seguirá ejecutándose en la GPU mientras el planificador del anfitrión lo mantenga en la lista de ejecución.

Al momento de lanzar la siguiente tarea en la lista de ejecución, y ésta regrese de suspensión *preemptive*, se inicializan todas las variables necesarias para rearmar el contexto por medio de la estructura *backup*. En caso contrario, las variables se inicializarán directamente en el código de la aplicación. En el algoritmo 4.5 se presenta la inicialización del *kernel* mediante la sintaxis CUDA.

Al inicio del algoritmo 4.4 la función *kernel* se encuentra ligeramente modificada en sus parámetros, ya que es necesario que reciba el *backup* donde se almacenará el contexto cuando se presente una suspensión *preemptive* y también se recibe el apuntador al estado del *quantum*, dicho valor arrojará ***true*** cuando se haya concluido el tiempo del *quantum*.

Como se mencionó anteriormente, esta solución se basa completamente en software, por lo que se debe estructurar la función *kernel* para mantener una convención que ayude a mitigar posibles problemas. Todas las declaraciones de variables deberán realizarse en la primera fase.

Las únicas declaraciones con inicialización permitidas en esta fase son aquellas que designan la posición tanto de los *threads* como de los *blocks* dentro de un *grid*, esto porque su información es necesaria en cada una de las siguientes fases.

```
__global__ void kernelFunction(int* a,.., int* resultados,..,
                        struct Backup* backup, bool* continuar_eje){
  /* Fase de declaración de variables */

  /* Variable posición de thread y block */
  int id_block = blockIdx.y * gridDim.x + blockId.x;
  int id_thread = threadIdx.y * blockDim.x + threadIdx.x;

  /* Variables locales */
  int temp1;
  int temp2;
  . . .
  int tempn;
  . . .

  /*Contadores*/
  int i;
  int j;
  . . .
  int k;
    . . .
```

Algoritmo 4.4: Fase de declaración de variables.

Enseguida se pasa a la fase de la inicialización (ver algoritmo 4.4) de cada una de estas variables y como se muestra en el algoritmo 4.5 se auxilia de una estructura *switch-case* con tres casos dependiendo del estado de cada *block*. Para seleccionar cada uno de los casos se debe leer el valor que se encuentra en la estructura *backup*, esto porque hay que recordar que el *kernel* por si sólo no sabe si se ejecuta por primera vez o está regresando de una suspensión *preemptive* con un cambio de contexto dentro del sistema.

Los tres estados son:

- ***INICIO***: Se presenta la primera vez que es lanzado un *kernel*, por lo que el valor inicial debe ser almacenado tanto en la variable local como en su espacio correspondiente en la estructura de copia de seguridad.

- ***EJECUCION***: Se presenta una vez inicializadas las variables en el estado anterior, o cuando el planificador da oportunidad de ejecución al *kernel* en suspensión para terminar el procesamiento. Aquí se realiza el cambio de contexto desde el *backup* extrayendo la variables locales para trabajar con la información como si nunca se hubiera suspendido el *kernel*.

- ***TERMINADO***: Debido a que muchas veces dentro de un *kernel* hay *blocks* que finalizan su procesamiento antes que otros, es necesario indicar que ese *block* ya terminó y no requiere hacer ningún cálculo.

```
  . . .
/* Fase de inicialización */
switch(backup.estado[id_block]){
  case INICIO:
    //Inicialización de variables locales
    temp1 = 0;
    temp2 = 0;
    . . .
    tempn = 0;

    //Inicialización de contadores
    i = 0;
    j = 0;
    . . .
    i = 0;
    break;
  case EJECUCION:
    //Inicialización de variables con respecto al backup
    temp1 = backup->temp1[id_block * blockDim.x + id_thread];
    temp2 = backup->temp2[id_block * blockDim.x + id_thread];
    . . .
    tempn = backup->tempn[id_block * blockDim.x + id_thread];

    //Inicialización de contadores con respecto al backup
    i = backup->i[id_block];
    j = backup->j[id_block];
    . . .
    k = backup->k[id_block];
    break;
  case TERMINADO:
    break;
}
. . .
```

Algoritmo 4.5: Fase de inicialización.

Una vez inicializadas todas las variables se puede realizar el procesamiento objetivo del *kernel* (ver algoritmo 4.6). Para ello, nuevamente se pregunta a la estructura *backup* el estado individual de cada *block*, dependiendo de lo que responda cada uno, se realiza lo siguiente:

```
. . .

  /* Fase de procesamiento*/
  switch(backup.estado[id_block]){
    case INICIO:
      if(id_thread == 0)
        backup->estado[id_block] = EJECUCION;
      _syncthreads(); // Todos esperan a que se modifique el estado
      // No hay break para que continue al siguiente caso
    case EJECUCION:
      //Ejecución del kernel
      do {
        //Procesa
        resultados = #paso_de_procesamiento;
        /*Si termina el thread, debe avisar a thread0 para saber
        estado general del block*/
        if(thread_completo)
          #avisa a thread0

        if(id_thread == 0 && #todos los threads terminarion)
          block_completo = true;

      } while(continuar_eje && !block_completo);

      /* Si se realizó la ejecución completamente */
      if(block_completo){
        if(id_thread == 0)
          backup->estado[id_block] = TERMINADO;
        break;
      }
      _syncthreads();//TB sincronizan para llegar al mismo valor de contador

            /* Si se indicó una suspesión, almacena contexto en
    backup */
      //Variables locales
      backup->temp1[id_block * blockDim.x + id_thread] = temp1;
      backup->temp2[id_block * blockDim.x + id_thread] = temp2;
      . . .
      backup->tempn[id_block * blockDim.x + id_thread] = tempn;
      //Contador
      if(id_thread == 0){
        backup->i[id_block]=i;
        backup->j[id_block]=j;
        backup->k[id_block]=k;
      }
```

```
45          break;
46        case TERMINADO:
47          break;
48      }
49
50  . . .
```

Algoritmo 4.6: Fase de procesamiento.

- ***INICIO***: Como se acaba de lanzar el *kernel* por primera vez, únicamente se cambia el estado del *block* a *EJECUCION*, y, como ahora se tiene un nuevo valor se puede ingresar al siguiente estado dentro del mismo *switch*.
- ***EJECUCION***: Al entrar en este caso, en primera instancia se realiza el paso de procesamiento para calcular el resultado de la aplicación, esto dentro de una estructura *do-while* para que al menos se realice una vez antes de que se presente una suspensión (variable *continuar_eje*), o se haya completado el procesamiento. En el instante en el que termine el procesamiento en un *thread*, éste le debe avisar inmediatamente al *thread0* para que se tenga la certeza del estado general del sistema. Ahora bien, si el *kernel* se ha completado, el estado del *block* en *backup* se modifica a *TERMINADO* y finaliza ese *block* sin realizar copia de seguridad para ahorrar tiempo de procesamiento.
- ***TERMINADO***: Cuando se llega a este estado simplemente se termina la ejecución de los *threads* de procesamiento.

4.4.1 Condición de carrera

La fase de procesamiento (ver algoritmo 4.6) es un procedimiento en el que hay que poner especial atención, ya que es donde se concentra el núcleo de las operaciones del *kernel*, además es donde se escriben variables compartidas por todo el *grid*. Por ello, hay que estar conscientes de que se debe evitar la condición de carrera.

Por esta razón, en el *case INICIO* únicamente el *thread0* de cada *block* está habilitado para modificar el estado que se guarda en el *backup*. Justo después del cambio de estado se debe esperar en una barrera para que todos los *threads* conozcan la actualización y no terminen abruptamente su procesamiento.

Lo anterior se repite en el *case EJECUCION*, cuando se termina el procesamiento, nuevamente sólo el *thread0* está autorizado para editar el contenido del arreglo

estado en la estructura de copia de información.

Finalmente, si el procesamiento se realiza con ayuda de contadores, al momento de que expire el *quantum*, todos los *threads* deberán suspenderse cuando lleguen al mismo valor, así que, lo más conveniente (en términos de memoria) es guardar sólo una copia de dicho contador. Entonces, una vez más el *thread0* será quien almacene la información en su correspondiente lugar dentro de *thread0* (línea 32).

4.5 Planificador GPU

El módulo principal del *framework* es el **Planificador GPU** (ver figura 4.4) porque es donde se realiza toda la planificación de los *kernels* a ser ejecutados en la GPU. Trabaja muy estrechamente con el módulo de asignación de prioridades, el cual da la pauta para elegir el orden de los *kernels* en un momento determinado de tiempo.

```
1 struct Task{
2   int id;
3   int costo;
4   int posicionSM;
5   bool* continuar_eje;
6   bool* kernel_completado;
7 }
```

Algoritmo 4.7: Estructura task.

Para poder planificar un conjunto de *kernels* en la GPU, es necesario conocer las particularidades de la arquitectura del sistema embebido en el que se implementará el *framework*. Esto es especialmente necesario para conocer el número máximo de *threads* que pueden estar en ejecución concurrentemente.

Cada uno de los *kernels* que soliciten recursos de cómputo en la tarjeta gráfica al planificador deberá ser mapeado a una estructura (algoritmo 4.7), la cual tendrá almacenados los parámetros necesarios para su ejecución, así como información relevante para la asignación de su prioridad en un instante de tiempo.

El algoritmo 4.8 muestra la manera en que deberá guiarse el programador para implementar el planificador, primero se obtiene la lista de tareas *sol* que solicitan los recursos del GPU y las une con aquellas que ya se tenían en la cola de ejecución R, esta cola contiene tanto las tareas que en una ejecución fueron beneficiadas de

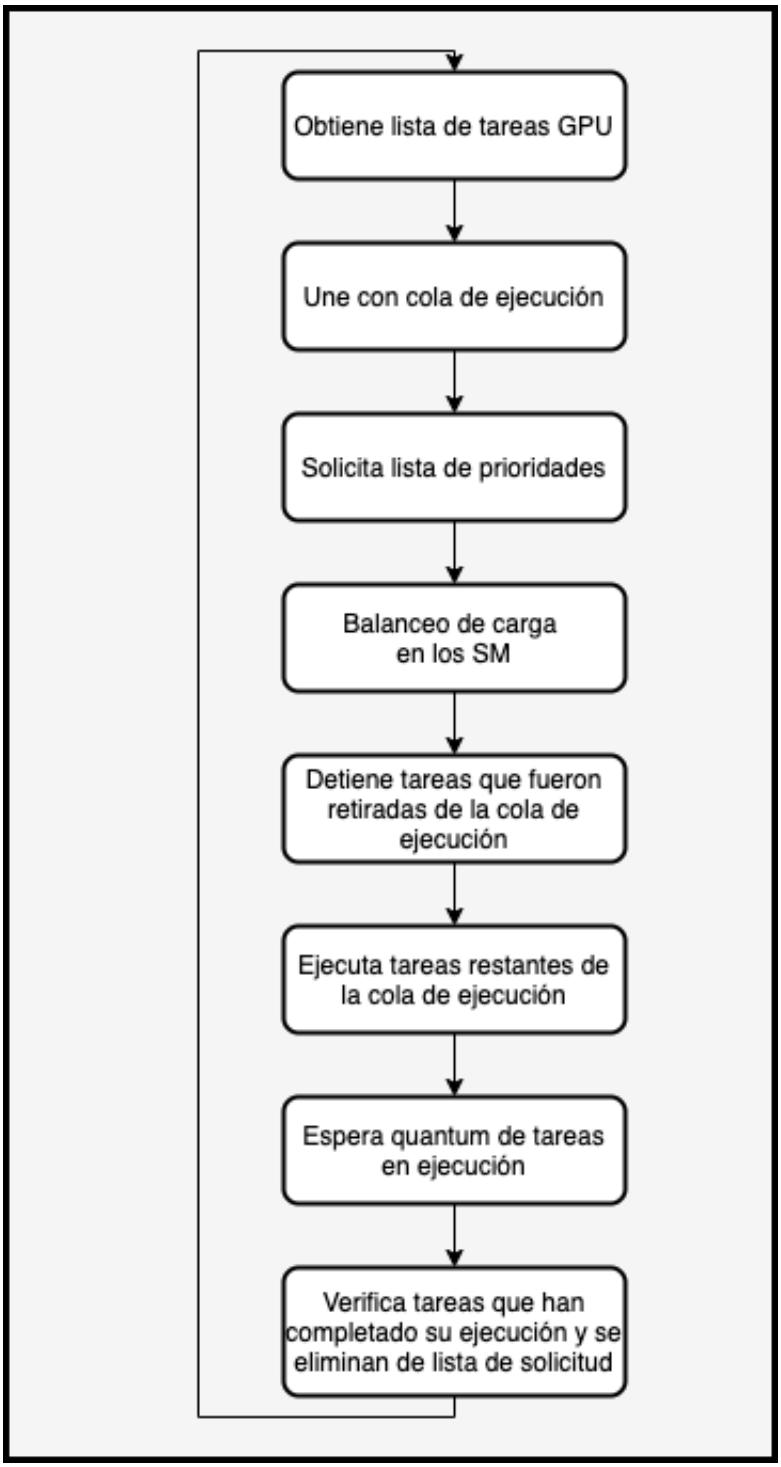

Figura 4.4: Diagrama de flujo del planificador.

consumir recursos como aquellas que se mantienen en estado de espera.

Se llama al módulo Asignación de Prioridad (ver sección 4.6) para que devuelva la cola ordenada por prioridad r (el orden depende del algoritmo de asignación de prioridades en tiempo real seleccionado).

La cola r se envía a un balanceador de carga que ayudará a la planificación de tareas ejecutables en un *quantum* optimizando la repartición de recursos. Posteriormente, detendrá la ejecución de aquellas que deban resguardar su contexto en la iteración actual j para darle lugar a *kernels* con mayor prioridad.

```
#define SM 2; //Número de SM
#define TSM 2048 //Número de threads por SM

sol[] = NULL //Set de kernels que solicitan recursos
R[] = NULL //Kernels en cola de ejecución
r = NULL //Set de kernels ordenados por prioridad

while(sol != NULL && R != NULL ){
  sol[] = getListaSolRecursos();

  R.instert(sol);

  r = asigPrioridad(R);

  balanceoCarga(r);

  detieneTareas(r);

  iniciaTareasFaltantesEjec(r);

  /*Espera quantum de tareas en ejecución*/

  tareasCompletadas(R);
}
```

Algoritmo 4.8: Función planificador.

Se esperará un determinado número de *quantums* (el estudio del *quantum* más apropiado queda fuera del alcance de esta tesis) y una vez alcanzado el plazo límite, se eliminarán de la cola R aquellas tareas que completaron su ejecución en la iteración j.

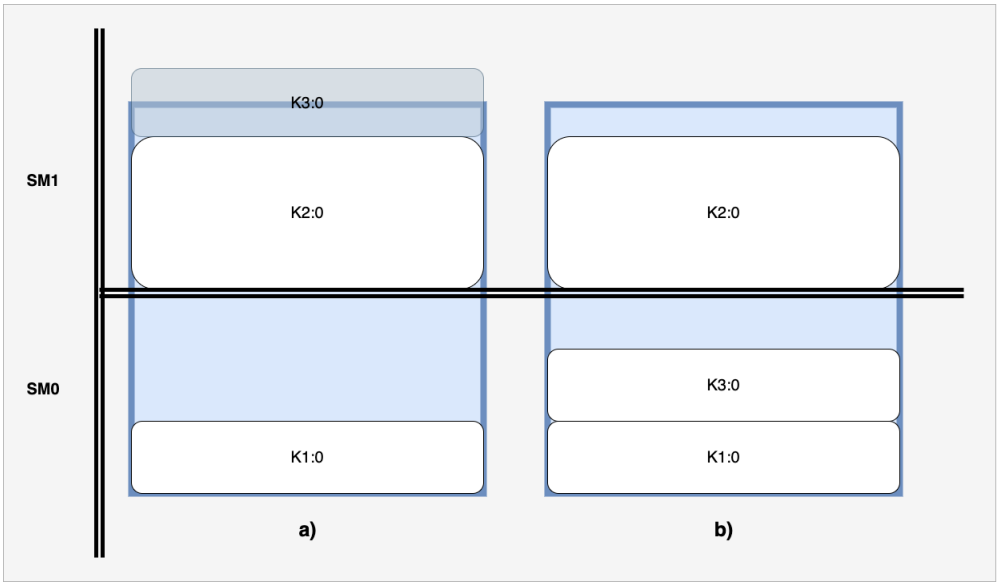

Figura 4.5: Diagrama de balanceo de carga de *kernels* en los *SM*.

4.5.1 Balanceador de carga

El balanceador de carga realiza la selección de tareas a ejecutar dentro del *quantum* y su orden a los *SM* para optimizar el uso de sus recursos, de acuerdo a una lista de ejecución y los *kernels* ejecutados en un *quantum* previo y su asignación dentro de los *SM*.

Para la asignación de carga de trabajo se tiene la suposición que todas las tareas son *preemptive* para cubrir las limitaciones, que por decisiones inherentes del hardware, muchas veces no podrán asignarse *kernels* planificables dentro del *quantum* aunque haya espacio para su ejecución.

Como se mencionó en el capítulo 3 el orden de lanzamiento de los *kernel* a la GPU afecta la forma en que serán asignados a los *SM*. Por ejemplo, en el escenario **a)** (ver figura 4.5). Primero se lanza **K1**, seguido de **K2** y al final **K3**, dando como resultado el desperdicio de recursos y que no sea posible ejecutar el *kernel* **K3**. Sin embargo, si se modifica el orden de asignación a los *SM* (ahora **K1**, **K3** y **K2**, se puede optimizar el reparto de tareas en las localidades, como se ve en el ejemplo **b)**.

Por esta razón, se diseñó un balanceador de carga (ver algoritmo 4.9) que permitirá la planificación de tareas de alta prioridad en un *quantum* rellenando los *SM*

hasta que se asigne un conjunto maximal de tareas para su operación en concurrente.

```
balanceoCarga(r){
  SM0[]=//Arreglo de tareas balanceadas en SM0
  SM1[]=//Arreglo de tareas balanceadas en SM1
  . . .
  SMn[]=//Arreglo de tareas balanceadas en SMn

  tareasEnRango(r);//Obtiene tareas en rango

  //Balancea carga en los SM
  //Para tareas fuera de ejecución
  for(i=0;i<fuera.length;i++){

    if(Hay warps disponibles para su ejecución)
      SMn.colocaEnPosicion(r[i]);
    else {
      #Libera espacio
      if(Hay warps disponibles para su ejecución)
        SMn.colocaEnPosicion(r[i]);
      else
        #Tarea no se ejecuta en quantum
    }
  }
}
```

Algoritmo 4.9: Balanceador de carga.

Como se ha visto a lo largo de este trabajo, la asignación de recursos para la ejecución de *kernels* en la tarjeta gráfica no es algo trivial, por lo que fue necesario diseñar un balanceador de carga (ver figura 4.6) que tomara en cuenta las características propias del sistema utilizado como caso de estudio.

La lógica de asignación de tareas a los *SM* se basa en colocar un conjunto maximal de *kernels* con las prioridades más altas en concurrente pero sin realizar muchos cambios de contexto.

El balanceador obtendrá de la lista de prioridades **r** el rango de tareas que en caso hipotético (con base a la capacidad de *threads* de la tarjeta GPU) pueden ser ejecutadas en los *SM* en concurrente.

La lista de prioridades ahora estará dividida en 3 secciones. La primera estará

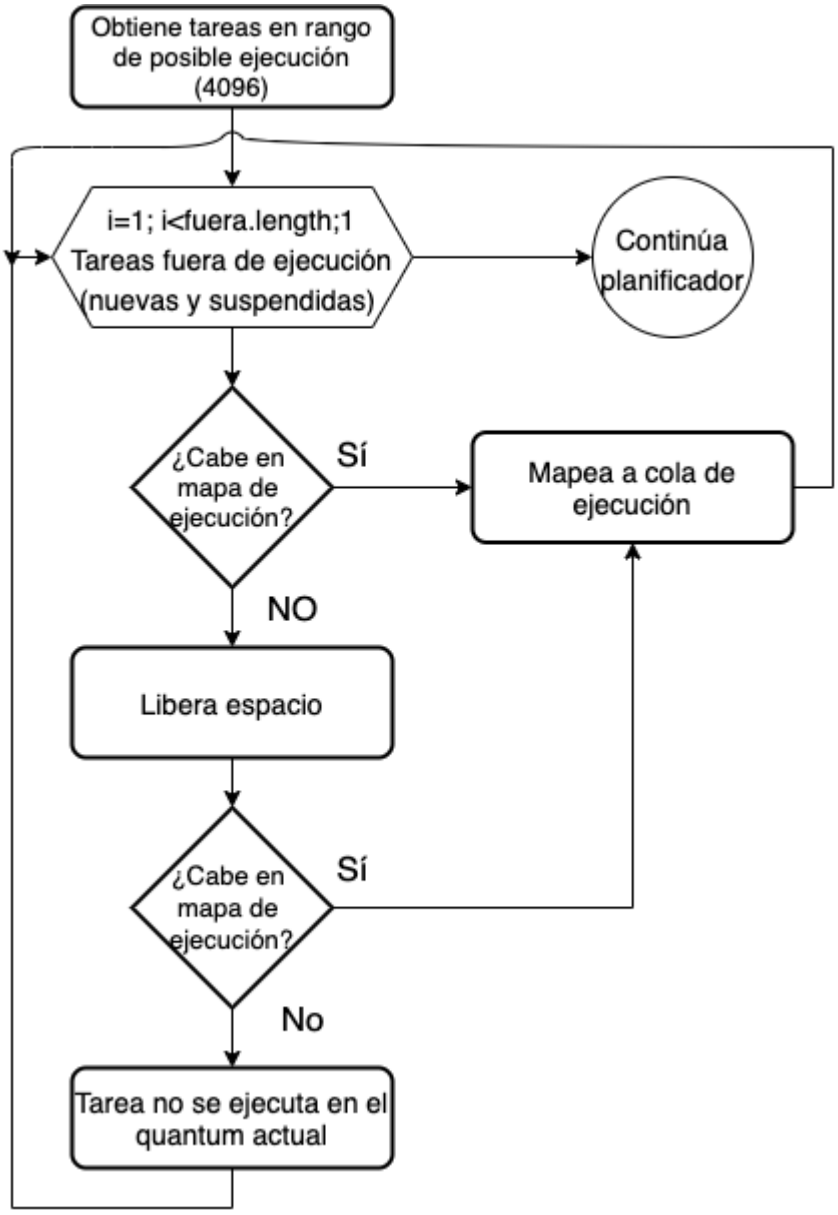

Figura 4.6: Diagrama de flujo del balanceador de carga.

compuesta por las **n** tareas que encabezan la lista, donde **n** es un parámetro de diseño que dependiendo de las características deseadas propias del sistema a implementar será seleccionada por el programador. La segunda sección contiene las tareas de mediana prioridad, aquellas que no se encuentran dentro de las primeras **n**, pero que sí dentro del rango de posible ejecución (para el caso de la tarjeta Jetson TX2 4096). Finalmente, las tareas de baja prioridad, aquellas que quedaron fuera del rango.

Debido a que la lista de ejecución posee una bandera con el estado de ejecución de cada *kernel*, el balanceador buscará recursos para las tareas que actualmente se encuentren fuera del mapeo de asignación (tareas recién llegadas, tareas en suspensión) para evitar en la medida de lo posible los cambios de contexto, ya que estos consumen una gran cantidad de tiempo.

En caso que no haya recursos suficiente se deberá librear espacio. El bloque *Libera espacio* del diagrama 4.6 busca asignar la tarea en turno de manera jerárquica auxiliado por el mapeo *SM* que se actualiza en cada asignación de la manera siguiente:

1. Asignar en el espacio libre dentro del mapeo *SM*, empezando por el *SM0*.
2. Sustituirla por una tarea de menor prioridad en posible suspensión con tamaño igual, si no existe, una mayor.
3. Liberar el espacio con las tareas en posible suspensión. Primero las que sean de mayor tamaño y contiguas para asegurar un mayor espacio.
4. Si después de esto no se puede asignar, la tarea se retendrá hasta el siguiente *quantum*.

El mapeo final establece el orden de llamado de los nuevos *kernels* y la suspensión de aquellas que fueron remapeadas o suspendidas durante el balanceo.

A continuación se muestran algunos de los diversos escenarios en forma de casos de estudio que pueden aparecer durante la planificación. Cada caso ilustra el balanceo de cargas a ejecutar en un *quantum*.

La tabla 4.1 muestra una descripción de lo que cada caso pretende ilustrar. Para los ejemplos se tiene $\mathbf{n} = \mathbf{2}$ tareas de alta prioridad.

El diagrama cuenta con la lista de las tareas que se encuentran actualmente mapeadas en el GPU de lado izquierdo, y del lado derecho la lista de prioridades actual ordenada y dividida en las tres secciones de prioridad, y al final de la segunda sección

Caso I	El planificador permite que las tareas continúen con su ejecución otro *quantum* sin ser interrumpidas.
Caso II	El balanceo de cargas requiere que se suspendan tareas.
Caso III	*Kernels* han completado su ejecución y deben ser sacados de la lista de prioridades.
Caso IV	*Kernels* de alta prioridad tienen preferencia sobre los de mediana prioridad.
Caso V	*Kernels* de alta prioridad que no estaban originalmente en ejecución realizan una gran cantidad de movimientos para asegurar su ejecución.

Tabla 4.1: Descripción de casos del balanceador de carga.

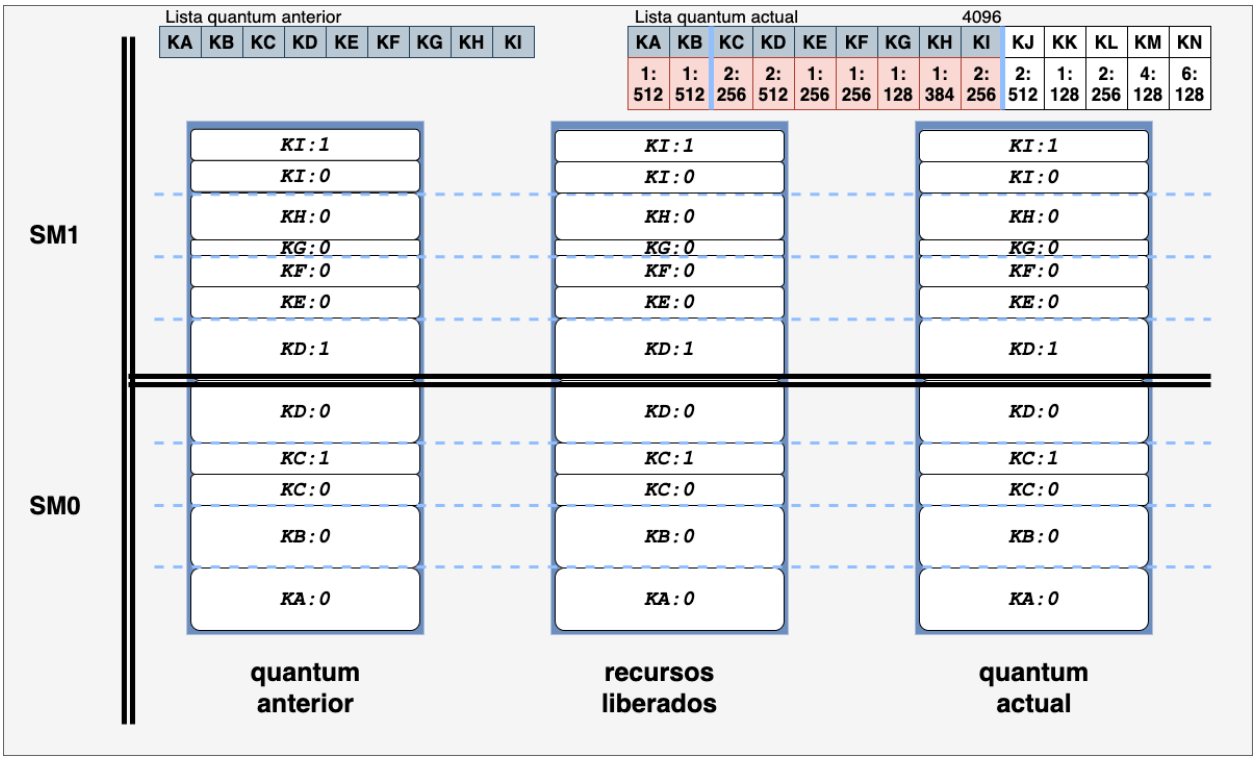

Figura 4.7: Caso I. Balanceo de cargas sin tareas en suspensión aún con cambio de prioridades.

la cuenta de *threads* que entraron en el rango de posible ejecución.

Se utiliza un código de color para facilitar la visualización de los eventos ocurri-

dos, se sombrean de gris aquellas tareas que originalmente estaban en ejecución y resultaron nuevamente como posible ejecución, en color verde aquellas que entraron como posible ejecución pero no lo estaban en el *quantum* anterior, y como amarillo aquellas que salieron del rango de posible ejecución para el nuevo *quantum*. También se sombrearon de rojo aquellas tareas que después del balanceo de carga lograrán ejecutarse en el *quantum* actual.

El **Caso I** (ver figura 4.7), representa un escenario en donde la totalidad de las tareas que estaban ejecutándose en el *quantum* anterior no terminaron su ejecución, pero vuelven a quedar en la posición de mayor prioridad en la lista. Puesto que ninguna tarea tuvo que ser suspendida, se les dará oportunidad de un siguiente *quantum* sin necesidad de suspenderse, guardar su contexto y relanzarse.

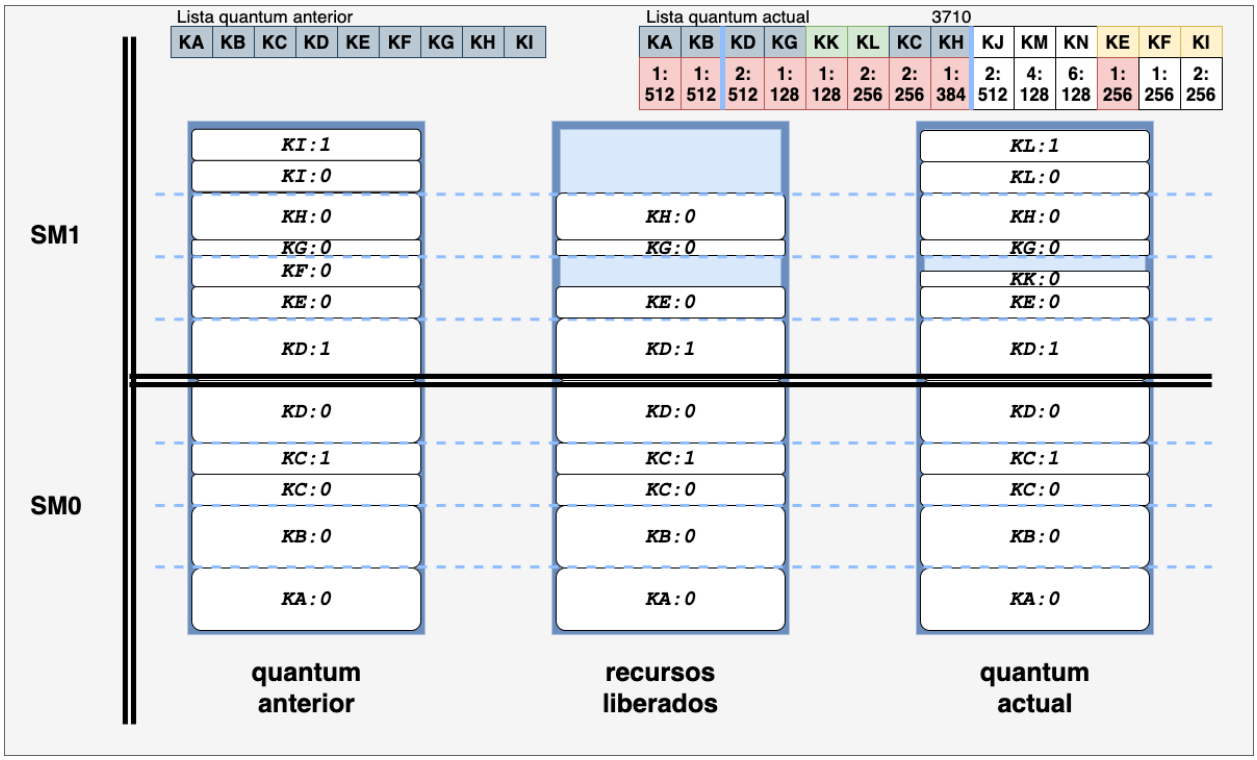

Figura 4.8: Caso II. Balanceo con tareas en suspensión.

Una vez terminado el *quantum* y recibiendo la lista de prioridades, se presenta el **Caso II** (ver figura 4.8), donde algunas de las tareas ahora cambiaron el orden en

la lista de prioridades. El balanceo inicia recibiendo que la lista de posible ejecución donde **KK** y **KL** entraron a la sección de posible ejecución, mientras que **KE**, **KF** y **KI** pueden ser suspendidas. El *kernel* **KK** comienza a buscar un lugar dentro de los *SM* para ser ejecutado. Sin embargo, la tarea de baja prioridad que más se ajusta a sus características es **KF**, por lo que se mapean sus recursos en su nueva posición. La siguiente tarea en busca de recursos ahora es **KL**, y justo empata con la posición de **KI**, intercambiando así el mapeo de sus recursos.

Finalmente, **KE** como no tuvo que ser movida de su posición para ingresar una de más alta prioridad, tiene oportunidad de ejecutarse nuevamente en este *quantum* a pesar de ser de baja prioridad.

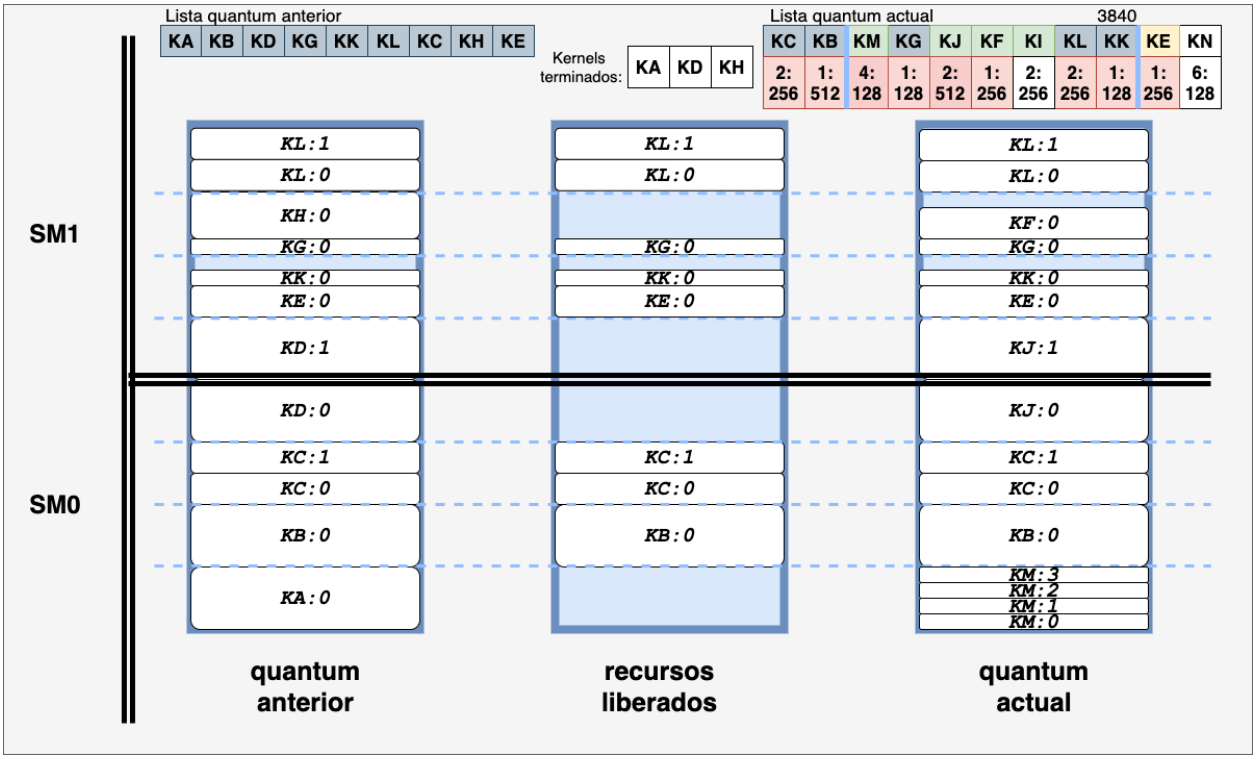

Figura 4.9: Caso III. Balanceo con tareas nuevas y terminadas.

Continuando con el ciclo de vida del planificador, se presenta un posible **Caso III** (ver figura 4.9) donde las tareas **KA**, **KD** y **KH** han completado su ejecución, por lo que son eliminadas de la lista de ejecución y del mapeo de los *SM*. Con esto,

la cola de ejecución ha sido modificada tanto en orden como en longitud. Las tareas **KM**, **KJ**, **KF** y **KI** ahora figuran dentro del rango de las posibles ejecuciones, **KE** como una tarea en posible suspensión y **KF** como **KI** con tareas suspendidas que intentan regresar a ejecución. La tarea **KM** se ajusta perfectamente en los recursos liberados por **KA** por lo que es mapeada sin problemas. Ocurre lo mismo con **KJ** ya que quedan libres los *warps* que ocupaba **KD** y también **KF** regresa de suspensión ocupando los *threads* que originalmente pertenecían a **KH**. Pero no así con **KI**, ya que después de no encontrar un espacio libre para su ejecución (ya que no tener espacio suficiente si se suspendiera la única tarea **KE** en posible suspensión), queda en lista de espera para este *quantum*.

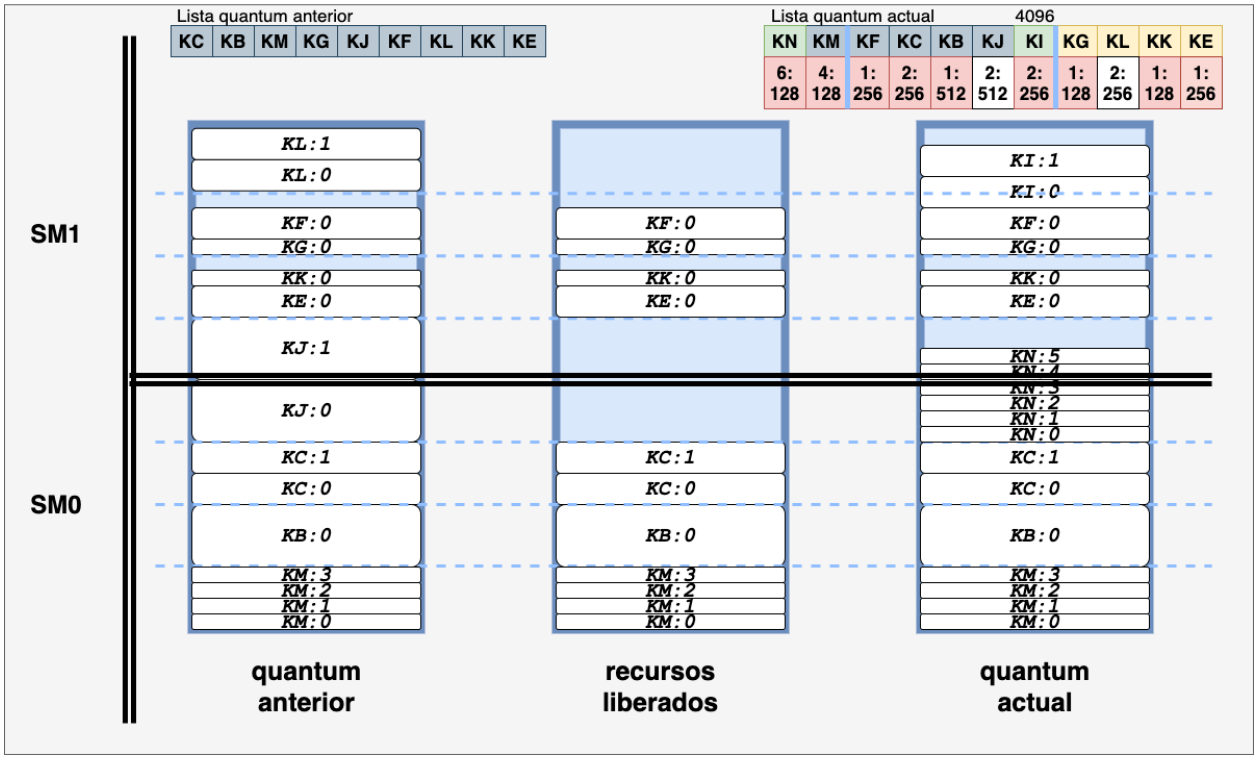

Figura 4.10: Caso IV. *Kernels* de alta prioridad tienen preferencia.

En el caso **Caso IV** (ver figura 4.10) la tarea **KN** es nueva dentro del rango de posible ejecución y con la más alta prioridad, **KI** intenta nuevamente regresar de suspensión y las tareas **KG**, **KL**, **KK**y **KE** serán posiblemente suspendidas con base en la capacidad se *warps* de los *SM*.

Así, debido a que **KN** es de alta prioridad, es la primera en verificar el espacio libre, y no encuentra disponible, procede a buscar una tarea de baja prioridad en posible suspensión que sea de tamaño igual o mayor pero tampoco hay, finalmente busca liberar espacio con las tareas en posible suspensión en conjunto, teniendo dos posibilidades ***KE-KK-KG*** (640 *threads*) o **KL** y el espacio libre contiguo (640 *threads*), sin embargo **KN** requiere 768 *threads*. Por ser la tarea de más alta prioridad, esta sigue buscando en las tareas con posibilidad de ejecución, la siguiente en posible suspensión es **KJ** que libera (1024 *threads*) suficiente para asignar **KN** y solicitar la suspensión de **KJ**.

La siguiente tarea por asignar es **KJ**, pero después de verificar todas las posibilidades, aunque pudiera suspender **KK**, **KL**y **KE**, ni todas juntas suspendidas logran suficiente espacio contiguo, por lo que no puede ser reasignada y se suspende al siguiente *quantum*. La última tarea por reasignar es **KI**, primero busca espacio libre pero no hay suficiente, lo siguiente es buscar por un tamaño similar dentro de las tareas en posible suspensión que es **KL**, con la que se intercambia para su ejecución.

El **Caso V** es algo particular (ver figura 4.11), donde el *quantum* anterior tiene *kernels* muy dispersos por los *SM*. En la lista de prioridades las tareas **KT** y **KU** son nuevas y encabezan la cola, ambas siendo de alta prioridad y ambas ocupando 1024 *threads* respectivamente.

Siguiendo con la metodología de balanceo de carga, **KT** busca liberar el espacio con las tareas de menor prioridad, por ello pregunta inicialmente con la tarea de prioridad más baja **KP**, una vez en posible suspensión no hay recursos necesarios para su ejecución.

Las siguientes tareas en suspender son **KR** y **KY**, ambas junto con el espacio vacío, suman los *threads* que se requiere. En este caso no se utilizan los últimos *threads* del *SM0* ya que para lanzar un *kernel* es necesario que los *threads* de un mismo *block* estén continuos. La siguiente tarea a ser mapeada es **KU**, al momento de realizar la liberación del espacio, se pueden sustituir los recursos de las tareas de baja prioridad **KS** y **KW**. Ahora bien, una vez asignadas las tareas de más alta prioridad se procede a rellenar los espacios con las tareas que tienen posibilidad de ejecutarse de acuerdo a la capacidad de los *SM*, incluso aquellas que fueron descartadas por las de más alta prioridad. A este instante, las tareas realmente suspendidas son **KR**, **KY**, **KS** y **KW**, y como posibles **KP** ya que no fue ocupado su lugar. Así **KW** es la primera

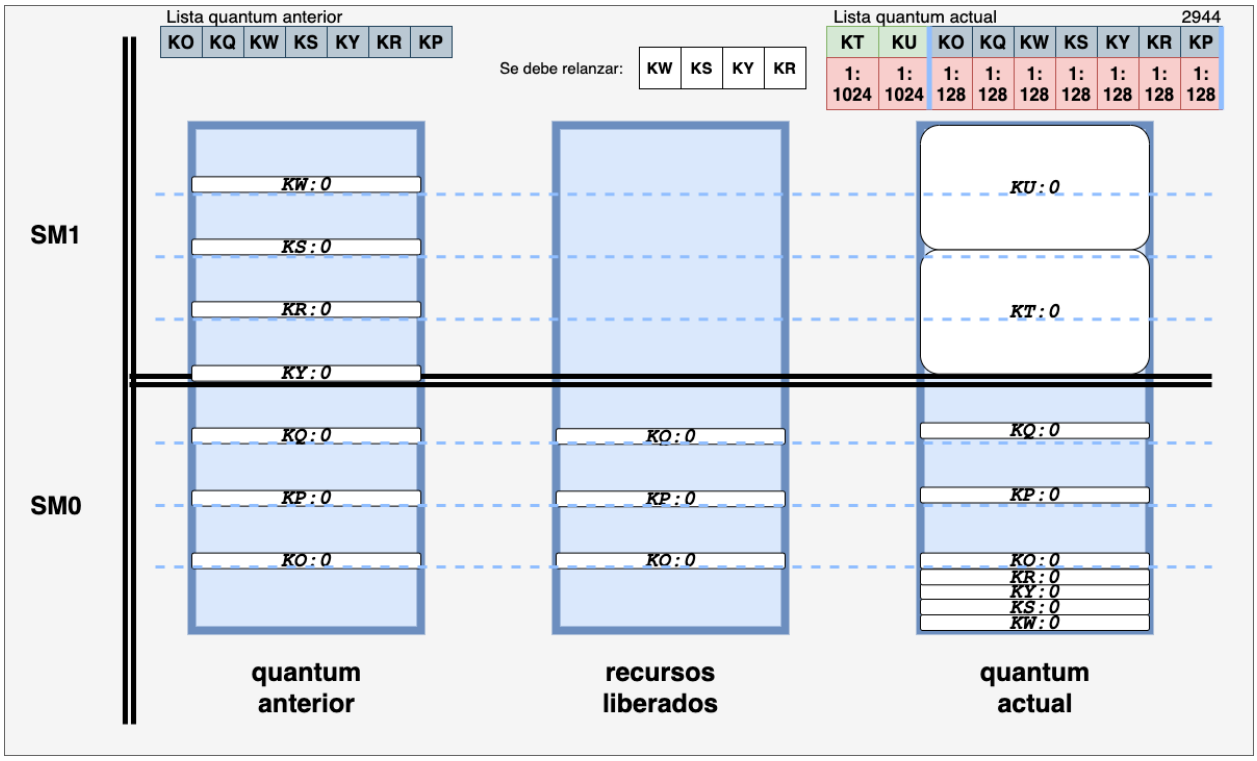

Figura 4.11: Caso V. *Kernels* de alta prioridad que no estaban en ejecución.

en buscar espacio.

En este caso el balanceador, aunque tuvo que mover de posición algunas tareas, lo que implica realizar cambios de contexto, pudo mapear la totalidad de tarea en la lista de ejecución, y en especial las de más alta prioridad.

Finalmente, se presentará en la siguiente sección el procedimiento para asignar la prioridad a las tareas y generar la lista de prioridades que recibe el Módulo Planificador GPU.

4.6 Asignación de prioridades

Al tener como módulos separados tanto el planificador como la asignación de prioridades, nos da la flexibilidad de poder implementar diferentes algoritmos de tiempo real (ver sección 2.2.3).

Para el diseño de este *framework*, se tomó como base el uso de algoritmos de asignación de prioridades en tiempo real de soporte monoprocesador, aunque no se descarta la posible implementación de aquellos que trabajan con multiprocesadores y subconjuntos de tareas. En este caso se asegura la ejecución de al menos dos tareas en concurrente en la tarjeta gráfica.

Dependiendo de las particularidades de cada algoritmo, se requerirá diferente información sobre la tarea a planificar, dichos parámetros podrán modificarse en la estructura (algoritmo 4.7) que mapea a las tareas.

Este módulo generará una lista de prioridades ordenando las tareas de mayor a menor, según sea el caso.

A cada iteración del planificador se asigna una prioridad a las tareas que actualmente están solicitando recursos de la GPU. La planificación de las tareas del CPU pudieran ser manejadas por el sistema operativo o algún otro componente, pero su estudio está fuera del alcance de esta tesis.

El ejemplo de un posible algoritmo para la asignación de la prioridad de las tareas se muestra en el capítulo 5.

CAPÍTULO
5

Métricas de rendimiento

Como se vio en el capítulo 2, existen diversas métricas que se le pueden aplicar a un producto de software, en este capítulo proponen algunas métricas para evaluar el rendimiento del *framework*.

Para poder realizar cualquier evaluación del rendimiento es necesario obtener datos importantes sobre las ejecuciones de un *kernel*, con dicha información se podrá implementar una serie de gráficas que permitan valorar la tendencia de los resultados.

Como ejemplo se toma a *Earliest Deadline First* (algoritmo 5.1), un algoritmo de planificación dinámica que ordena un conjunto de tareas $R = \{\tau_1, \tau_2, ..., \tau_n\}$ de acuerdo a sus plazos absolutos d. Las tareas con plazos más próximos se ejecutarán con una prioridad más alta[13]. Debido a que el plazo límite absoluto de una tarea periódica τ_i depende de la actual instancia τ_j, plazo límite relativo D, un periodo de activación T y una fase o tiempo de activación de la primera instancia Φ. Se tiene que el plazo límite absoluto queda definifo como:

$$d_{i,j} = \Phi_i + (j-1)T_i + D_i \tag{5.1}$$

```
1 EDF(R(tj)){
2   for instante de planificacion tj
3     r = ordena_deadline(R(tj),dj);
4   return r
5 }
```

Algoritmo 5.1: Algoritmo de planificación EDF.

EDF no realiza ninguna suposición sobre si las tareas son periódicas o aperiódicas debido a que realiza una asignación dinámica.

Al ejecutarse normalmente en escenarios *preemptive*, la tarea que se está ejecutando actualmente realiza una suspensión *preemptive* y se da lugar a cualquier otra que posea el plazo límite más próximo. Por esta flexibilidad es el algoritmo más extendido a la hora de pensar en implementar un planificador.

La primera métrica a considerar para el diseño del *framework* es conocer si un conjunto de tareas es planificable, se utiliza la función de utilidad U, la cual describe el consumo de cómputo C_i de las tareas entre su periodo T_i. Para evitar perder los tiempos límite de las tareas se debe mantener inferior a **1**.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{5.2}$$

Ahora bien, para evitar que una tarea pierda su plazo límite es necesario prevenir que esté bloqueada por más de $D_i - C_i$ unidades de tiempo, asegurando que la tarea concluya antes de que se presente el plazo vencido dm.

Adentrándonos en la medición del desempeño del planificador GPU *preemptive*, una métrica que nos permite evaluar el rendimiento del sistema en cuanto al número de plazos vencidos n_{dm}, cuyo objetivo principal es minimizarlo mediante los módulos del *framework*.

$$n_{dm} = \sum_{i=1}^{n} dm_i \tag{5.3}$$

Durante la descripción del *framework* se ha establecido que una tarea puede sufrir un determinado número de suspensiones *preemptive* generando n_{bk} cambios de contexto.

El cambio de contexto tiene un costo de tiempo computacional alto, debido a la inserción $t_bk[in]$ y extracción $t_bk[ex]$ de este en el *backup*. Dicho tiempo también puede representar una métrica de desempeño donde el objetivo es minimizar el costo de los cambios de contexto t_{bk} que involucran el manejo de la memoria.

$$t_{bk} = t_{bk[in]} + t_{bk[ex]} \tag{5.4}$$

El tiempo de ejecución total con suspensión *preemptive* t_p nos indica el tiempo total que utiliza el programa tomando en cuenta el tiempo empleado en los cambios

de contexto.

Finalmente, una de las métricas empleadas para decidir si un sistema es factible para ser paralelizado es la aceleración relativa (speedup en ingles) que mide las cantidad de veces que es más rápido una aplicación paralela a su contra parte serial.

Para esto, en ocasiones es importante conocer el tiempo total del cálculo t_{np} de cada tarea en el ambiente *preemptive*, para comparar la variación $S(p)$ que pudiera existir con el tiempo original t_{or} en un ambiente sin planificador *preemptive*, donde $S(p) > 1$ significa una mejora en la ejecución de la aplicación.

$$S(p) = \frac{t_{or}}{t_{np}} \tag{5.5}$$

En cuanto a la optimización del reparto de los recursos se puede definir el tiempo fuera de línea o de ociosidad t_{idle} que indica el tiempo en que un *kernel* está fuera de operación y, si se suman todos los tiempos de ociosidad de los *kernels*, se podrían establecer estrategias más complejas de planificación o balanceo de cargas para que se garantice el eventual consumo de recursos, y se eliminen los posibles plazos vencidos.

CAPÍTULO
6

Conclusiones

En este trabajo se ha hablado de los sistemas embebidos, y principalmente de los heterogéneos, de cómo han sido adoptados en la industria para realizar tareas específicas que requieren cada vez más un aumento y aceleración de su procesamiento. Tener las bases del diseño de este *framework* podría permitir planificar la ejecución de tareas *preemptive*, facilitar la disminución de los plazos vencidos de tareas con alta prioridad y mejorar el desempeño general del sistema.

La principal contribución de este trabajo es el generar un *framework* que facilite la planificación de tareas *preemptive* de la GPU en sistemas embebidos heterogéneos contemplando el cómputo general en unidades de procesamiento gráfico y la arquitectura del sistema.

El diseño del *framework* tomó como base el sistema embebido heterogéneo NVIDIA Jetson TX2, aunque puede ser aplicado a otros dispositivos, siempre y cuando cumpla con ciertas características, como la memoria unificada.

El *framework* se basa en una planificación *preemptive* limitada por puntos fijos y posiciona dentro de las siguientes clasificaciones implementación:

- **Por implementación**: *Partición de kernel basada en software.*
- **Por planificación**: *Planificación por prioridad.*
- **Por modificación**: *Modificación de código fuente.*

El *framework* está integrado por cinco bloques que describen el funcionamiento de los componentes necesarios para realizar desde la implementación del modo *preemptive* hasta su planificación y lanzamiento dentro de la GPU. Los módulos que se presentan son:

- **Lanzamiento de kernel**. Presenta las herramientas para manejar los lanzamientos de los *kernels* desde el CPU pidiendo permiso al planificador GPU.
- **Memoria**. Implementa las directivas para utilizar la memoria unificada del sistema embebido y así optimizar el tiempo de programación con respecto a las transferencias de memoria.
- **Puntos Preemptive**. Propuesta para localizar e implementar los puntos clave que ayudarán a realizar suspensiones *preemptive* en el código de las aplicaciones.
- **Planificador GPU**. Módulo para planificar y balancear la carga de los núcleos de procesamiento de la tarjeta gráfica.
- **Asignación de prioridades**. Sección donde se debe emplear el algoritmo en tiempo real para la creación de las listas de prioridades en el tiempo.

Finalmente, se propone un conjunto de métricas para evaluar el rendimiento del sistema una vez implementado, logrando dar la pauta para conocer el desempeño de un conjunto específico de aplicaciones, o en dado caso, mejorar el diseño del *framework*.

- U Función de Utilidad.
- $D_i - C_i$ Tiempo máximo de bloqueo de una tarea.
- n_{dm} Número de plazos vencidos.
- $t_bk[in]$, $t_bk[ex]$ Tiempo de inserción y extracción de contexto.
- $S(p)$ Aceleración relativa.
- t_{idle} Tiempo de ociosidad de los *kernels*.

6.1 Trabajo Futuro

Aunque se realizó un análisis exhaustivo para llegar a la realización del diseño del *framework*, es necesaria su futura implementación para poder corroborar su eficacia en ambientes reales.

Otro apartado que se trabajará en un futuro es la modificación de los módulos para trabajar nativamente con algoritmos de asignación de prioridad específicos para multiprocesadores con subconjuntos de tareas, ya que actualmente sólo se asegura la ejecución de las tareas predefinidas como de mayor prioridad a cada iteración del planificador GPU.

Finalmente, también podría generarse una actualización en algunas herramientas del *framework* para poder implementar programación orientada a objetos, y con ello, tener un mayor campo de acción e impacto con aplicaciones de la industria que utilizan este paradigma.

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] Hwu and Wen-Mei, *NVIDIA CUDA Compute Unified Device Architecture*, version 2. ed., NVIDIA, 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4814979

[3] J. Cheng and M. Grossman, *Professional CUDA C Programming*, 2014.

[4] C. Kubisch, "Life of a triangle - NVIDIA's logical pipeline," 2015. [Online]. Available: https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline

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

[6] D. Franklin, "NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge," 2017. [Online]. Available: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/

[7] L. Jsachs, "Jetson TX2 Developer Kit," pp. 1–24, 2017. [Online]. Available: https://developer.nvidia.com/embedded/jetson-tx2-developer-kit

[8] NVIDIA, "Transformar industrias con la computación de IA," 2018. [Online]. Available: https://www.nvidia.com/es-la/industries/

[9] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," *Office*, vol. 121990, no. 1, p. 1, dec 1990. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=159342

[10] S. Sanchez, *Ingenieria del software: un enfoque desde la guía SWEBOK.* Alfaomega Grupo Editor, S.A. de C.V, 2012.

[11] G. Everett, *Software testing : testing across the entire software development life cycle.* Piscataway, NJ Hoboken, N.J: IEEE Press Wiley-Interscience, 2007.

[12] Software Guru, "Pruebas de Software," 2013. [Online]. Available: https://sg.com.mx/buzz/evento-sg/sg-virtual-4-abril-2013

[13] G. C. Butazzo, *Hard real-time computing systems: Predictable scheduling algorithms and applications.* Springer Science & Business Media, 1998, vol. 36, no. 3.

[14] A. S. Tanenbaum and H. Bos, *Modern Operating Systems.* Pearson Education, Inc., 2014, vol. 2. [Online]. Available: http://www.amazon.com/dp/0136006639

[15] J. Calhoun and H. Jiang, "Preemption of a CUDA kernel function," *Proceedings - 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, SNPD 2012*, pp. 247–252, 2012.

[16] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. A survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013.

[17] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[18] S. Heath, *Embedded systems design.* EDN Series For Design Engineers, 2009, vol. 24.

[19] J. Lehoczky, L. Sha, and Y. Ding, "Rate monotonic scheduling algorithm: Exact characterization and average case behavior," *Proceedings - Real-Time Systems Symposium*, pp. 166–171, 1989.

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

[21] V. Shinde and S. C., "Comparison of Real Time Task Scheduling Algorithms," *International Journal of Computer Applications*, vol. 158, no. 6, pp. 37–41, 2017.

[22] S. Kato and Y. Ishikawa, "Gang edf scheduling of parallel task systems," in *2009 30th IEEE Real-Time Systems Symposium*, Dec 2009, pp. 459–468.

[23] Stancu and A. Codres, *Jetson TX2 and CUDA Programming*, second edi ed., NVIDIA, 2018.

[24] S. Rennich, "Cuda c/c++ streams and concurrency," 2012. [Online]. Available: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

[25] C. Cooper, "GPU Computing with CUDA Lecture 2 - CUDA Memories," Chile, 2011. [Online]. Available: http://www.bu.edu/pasi/files/2011/07/Lecture2.pdf

[26] Y. Lin and V. Grover, "Using CUDA Warp-Level Primitives," 2018. [Online]. Available: https://devblogs.nvidia.com/using-cuda-warp-level-primitives/

[27] W. P. NVIDIA, "NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built." [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[28] 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

[29] C. Hartmann and U. Margull, "GPUart - An application-based limited preemptive GPU real-time scheduler for embedded systems," *Journal of Systems Architecture*, vol. 97, pp. 304–319, 2019.

[30] 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.

[31] H. Zhou, G. Tong, and C. Liu, "GPES: A preemptive execution system for GPGPU computing," *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, vol. 2015-May, pp. 87–97, 2015.

[32] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "RGEM: A responsive GPGPU execution model for runtime engines," *Proceedings - Real-Time Systems Symposium*, pp. 57–66, 2011.

[33] 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.

[34] K. Uchiyama, "NVIDIA Jetson TX2 Enables AI at the Edge | NVIDIA Newsroom," 2017. [Online]. Available: http://nvidianews.nvidia.com/news/nvidia-jetson-tx2-enables-ai-at-the-edge

[35] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. Donelson Smith, "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed," *Proceedings - Real-Time Systems Symposium*, vol. 2018-January, pp. 104–115, 2017.

[36] D. Franklin, "NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge," 2017. [Online]. Available: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/

[37] T. Allen, "Improving Real-Time Performance with CUDA Persistent Threads (CuPer) on the Jetson TX2," 2018. [Online]. Available: https://www.concurrent-rt.com/wp-content/uploads/2016/09/Improving-Real-Time-Performance-With-CUDA-Persistent-Threads.pdf

[38] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," *Proceedings - International Symposium on Computer Architecture*, pp. 193–204, jun 2014.

[39] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing," *Proceedings - International Symposium on High-Performance Computer Architecture*, vol. 2016-April, pp. 358–369, mar 2016.

[40] Z. Lin, L. Nyland, and H. Zhou, "Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching," *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 0, pp. 898–908, nov 2016.

[41] H. Lee and M. A. Al 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.

[42] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "Effisha: A software framework for enabling efficient preemptive scheduling of GPU," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*. Association for Computing Machinery, jan 2017, pp. 3–16.

[43] C. Reano, 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, pp. 1089–1102, 2018.

[44] Adaptive Computing, "TORQUE Resource Manager," 2016. [Online]. Available: http://www.adaptivecomputing.com/products/open-source/torque/

[45] 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, 2017.

[46] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for gpu with preemption support," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 119–130.