



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

“Diseño de un framework para la planificación de tareas preemptive
en sistemas embebidos heterogéneos”

TESIS

QUE PARA OPTAR POR EL GRADO DE:

MAESTRO EN CIENCIA E INGENIERÍA
DE LA COMPUTACIÓN

PRESENTA:

José Antonio Ayala Barbosa

DIRECTOR DE TESIS:

Dr. Paul Erick Méndez Monroy

Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas

Ciudad Universitaria, CDMX a Septiembre 2020

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN,
UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

JOSÉ ANTONIO AYALA BARBOSA

Tesis que para optar por el grado de
Maestro en Ciencia e Ingeniería de la Computación
Primera Edición, 3 de junio de 2020

Agradecimientos

A mis padres

A mis amigos

A la UNAM

A CONACYT

Índice general

1. Introducción	1
1.1. Estructura de la tesis	2
2. Antecedentes	3
2.1. Ingeniería de Software	3
2.1.1. Framework	4
2.2. Sistemas en tiempo real	4
2.2.1. Tipos de tarea	5
2.2.2. Esquemas de planificación	5
2.2.2.1. Planificación cooperativa	5
2.2.2.2. Planificación preemptive	6
2.2.2.3. Clasificación de soporte preemptive	8
2.2.3. Algoritmos de planificación	10
2.2.3.1. Shortest Job First	10
2.2.3.2. Earliest Deadline First	11
2.2.3.3. Rate Monotonic	11
2.2.3.4. Deadline Monotonic	12
2.2.3.5. Least Laxity First	12
2.2.3.6. Gang Earliest Deadline First	12
2.3. CPU	12
2.3.1. Arquitectura del CPU	13
2.3.2. Manycore y Multicore	13
2.4. GPU	14
2.4.1. Arquitectura del GPU	14
2.4.2. Arquitectura CUDA	15
2.4.2.1. Perspectiva del programador	16
2.4.2.2. Perspectiva del hardware	17

2.4.3.	Arquitectura Pascal	19
2.4.4.	GPGPU	20
2.5.	Sistemas embebidos	20
2.5.1.	Sistemas embebidos heterogéneos	21
2.6.	Material de trabajo	21
2.6.1.	Jetson TX2	21
3.	Trabajo Relacionado	25
3.1.	Resumen	32
4.	Diseño	35
4.1.	Descripción general del framework	35
4.1.1.	Precondiciones necesarias	37
4.2.	Puntos preemptive	38
4.2.1.	Condición de carrera	43
4.3.	Memoria	43
4.3.1.	Almacenamiento del contexto	43
4.3.2.	Variables compartidas	45
4.4.	Lanzamiento del kernel	45
4.4.1.	Planificador	47
4.4.1.1.	Balanceador de carga	49
4.5.	Asignación de prioridades	58
5.	Rendimiento	59
5.1.	Métricas de rendimiento por kernel	59
5.2.	Métricas de rendimiento multikernel	61
6.	Conclusiones	63
6.1.	Contribuciones	63
6.2.	Trabajo futuro	63

Índice de figuras

2.1. Planificación cooperativa.[1]	6
2.2. Planificación cooperativa con plazos vencidos.[1]	7
2.3. Planificación preemptive.[1]	8
2.4. Representación de un CPU y un GPU[2].	14
2.5. Escritura en DRAM[2].	15
2.6. Lectura en DRAM[2].	16
2.7. Representación de los componentes de un grid[3].	17
2.8. Orden de threads y blocks dentro de un grid[3].	18
2.9. Comparación de directivas para manejo de memoria.	19
2.10. Aceleración de programas en GPUs[4].	20
2.11. Diagrama de la arquitectura del sistema Jetson TX2[5].	22
3.1. Diagrama de asignación de un kernel a los SM.	27
3.2. Diagrama de asignación de varios kernels a los SM.	28
4.1. Diagrama del framework para la planificación de tareas preemptive en sistemas embebidos heterogéneos.	36
4.2. Diagrama del flujo del framework.	37
4.3. Aplicaciones en ejecución concurrente.	45
4.4. Diagrama de flujo del planificador.	48
4.5. Diagrama de balanceo de carga de kernels en los SM.	49
4.6. Diagrama de flujo del balanceador de carga.	51
4.7. Caso I Kernels mantienen su prioridad en la siguiente iteración.	52
4.8. Caso II Kernels de baja prioridad deben relanzarse.	52
4.9. Caso IIIa Kernels que completaron su ejecución.	53
4.10. Caso IIIb Kernels que completaron su ejecución.	54
4.11. Caso IV Kernels muy grandes.	55

4.12. Caso Va Kernels de mayor prioridad que no estaban en ejecución. . .	56
4.13. Caso Vb Kernels de mayor prioridad que no estaban en ejecución. . .	56
4.14. Caso Vc Kernels de mayor prioridad que no estaban en ejecución. . .	57
4.15. Caso Vd Kernels de mayor prioridad que no estaban en ejecución. . .	57
4.16. Caso VI.	58

Índice de tablas

2.1.	Matriz de comparación de algoritmos de planificación.	11
2.2.	Componentes de CUDA para el programador.	17
2.3.	Jerarquía de almacenamiento en device.	18
2.4.	Componentes de CUDA para el hardware.	18
2.5.	Especificaciones del sistema Jetson TX2[6].	23
3.1.	Asignación de un solo kernel a los SM.	26
3.2.	Asignación de varios kernels a los SM.	27
3.3.	Matriz de clasificación de trabajos relacionados.	33

Índice de Algoritmos

2.1.	Transformación para obtener id del thread y del block.	16
4.1.	Fase de declaración de variables.	39
4.2.	Fase de inicialización.	40
4.3.	Fase de procesamiento.	42
4.4.	Estructura Backup para almacenar el contexto.	44
4.5.	Algoritmo para lanzamiento del kernel en el lado del host.	46
4.6.	Función kernel completo.	47
4.7.	Estructura task.	48

Introducción

Actualmente las organizaciones requieren de la utilización de aplicaciones con tareas específicas, por lo que se puede valer de los sistemas embebidos en vez de gastar en computadoras de propósito general. Es común encontrar que estas aplicaciones requieren trabajar con tiempos de respuesta relativamente cortos, comúnmente llamados sistemas en tiempo real, que deben reaccionar dentro de límites de tiempo precisos para garantizar una corrección funcional, satisfacer los criterios de calidad o evitar daños críticos.

Las tecnologías emergentes están requiriendo de soluciones cada vez más intensivo, por lo que existe una creciente migración de paradigma en las empresas para acelerar sus aplicaciones embebidas mediante la utilización de cómputo general en unidades de cómputo gráfico (GPGPU) con el fin de solventar estas demandas de recursos.

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 de que las empresas que manufacturan las tarjetas gráficas están haciendo un gran esfuerzo por generalizar el uso de sus plataformas, no es suficiente debido a que no se conoce completamente la arquitectura de operación, por lo que se requiere buscar nuevas formas de administrar los sistemas que correrán sobre ellas.

1. INTRODUCCIÓN

El objetivo principal de esta tesis es la definición del diseño de un framework que facilite la planificación de tareas preemptive, específicamente, en sistemas embebidos heterogéneos, aquellos que integran tanto una Unidad Central de Procesamiento (CPU) y una Unidad de Procesamiento Gráfico (GPU). La solución que se presenta estará orientada a aplicaciones cuyos requisitos de ejecución cumplen con las características de procesamiento embebido y puedan ser manejadas por un planificador de sistemas en tiempo real.

Tener las bases del diseño de un framework que permita planificar la ejecución de tareas preemptive, permite disminuir los plazos vencidos de tareas con alta prioridad y mejorar el desempeño general del sistema.

El principal problema al que nos enfrentamos a la hora de implementar aplicaciones embebidas heterogéneas es la poca investigación 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 desarrollos de aplicaciones en tiempo real que utilice tareas con suspensión preemptive.

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. En seguida, en el capítulo **Trabajo Relacionado**, se da un breve resumen sobre los textos que contienen información pertinente del estado del arte del tema. Posteriormente, se encuentra el capítulo **Diseño del framework** en donde se describe puntualmente la propuesta de solución. Una vez conocida la solución, en el capítulo **Rendimiento** se anexan algunas métricas de rendimiento que se le podrán aplicar al framework cuando en un futuro se encuentre implementado. Finalmente, en el capítulo **Conclusiones y Trabajo Futuro** se recapitulan los alcances del trabajo y se mencionan los puntos que se dejaron para un trabajo futuro,

Antecedentes

2.1 Ingeniería de Software

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

"La Ingeniería de Software[7] es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software; es decir, la aplicación de la ingeniería al software."

La Ingeniería de Software aplica diferentes técnicas, normas y métodos que permiten obtener mejores resultados al desarrollar y usar piezas software, al tratar con muchas de las áreas de Ciencias de la Computación es posible llegar a cumplir de manera satisfactoria con los objetivos fundamentales de la Ingeniería de Software. Entre los objetivos de la Ingeniería de Software están[8]:

- Mejorar el diseño de aplicaciones o software de tal modo que se adapten de mejor manera a las necesidades de las organizaciones o finalidades para las cuales fueron creadas.
- Promover mayor calidad al desarrollar aplicaciones complejas.
- Brindar mayor exactitud en los costos de proyectos y tiempo de desarrollo de los mismos.

2. ANTECEDENTES

- Aumentar la eficiencia de los sistemas al introducir procesos que permitan medir mediante normas específicas la calidad del software desarrollado, buscando siempre la mejor calidad posible según las necesidades y resultados que se quieren generar.
- Una mejor organización de equipos de trabajo, en el área de desarrollo y mantenimiento de software.
- Detectar a través de pruebas, posibles mejoras para un mejor funcionamiento del software desarrollado.

2.1.1 Framework

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

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

2.2 Sistemas en tiempo real

Los sistemas en tiempo real son sistemas de cómputo cuyas tareas deben actuar dentro de limitaciones de tiempo precisas ante eventos en su entorno. Por lo que el comportamiento del sistema depende, no solo del resultado del cálculo, sino también del momento (tiempo) en qué se produce [9].

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

- Soft deadline: En este tipo se pueden superar algunos tiempos límites y el sistema puede aún funcionar correctamente.

- Firm deadline: Aquí los resultados obtenidos en los plazos vencidos no son útiles, pero los plazos son tolerados frecuentemente.
- Hard deadline: Si una tarea no se cumple en el tiempo límite, se producirán resultados catastróficos. Este tipo de límites se utilizan comúnmente en tareas que realizan operaciones críticas.

2.2.1 Tipos de tarea

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

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

La mayoría del tiempo los intervalos de tiempo se dan por el plazo límite de una tarea.

2.2.2 Esquemas de planificación

2.2.2.1 Planificación cooperativa

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

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

2. ANTECEDENTES

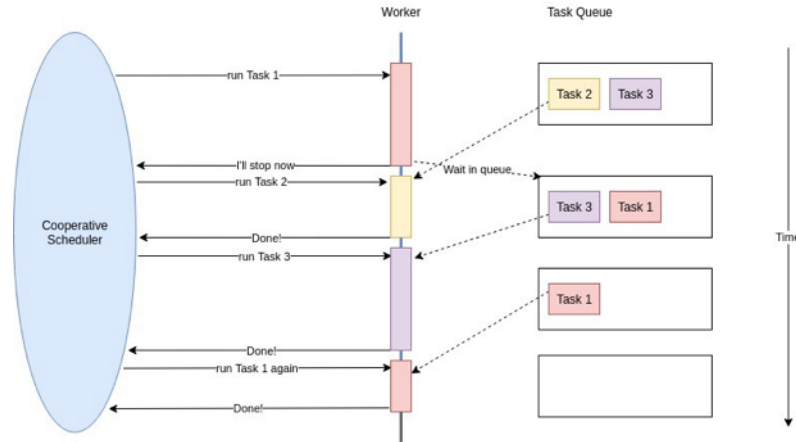


Figura 2.1: Planificación cooperativa.[1]

2.2.2.2 Planificación preemptive

En este esquema el planificador asigna las tareas a los recursos disponibles, y les define un tiempo de ejecución máximo, comúnmente llamado quantum[10]. Superado este punto, el planificador interrumpe la tarea para que otra sea ejecutada en su lugar, y la tarea interrumpida debe esperar hasta que le toque su turno nuevamente. Un ejemplo de este esquema, lo tenemos en la figura 2.3.

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

Debido a que muchas veces se interrumpen tareas a la mitad de un proceso, es necesario almacenar y restaurar el contexto que se tenía 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 actualmente es un requisito para la implementación de los sistemas en tiempo real. Por lo que ha surgido una clasificación dependiendo de su implementación[11].

- **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[11]:

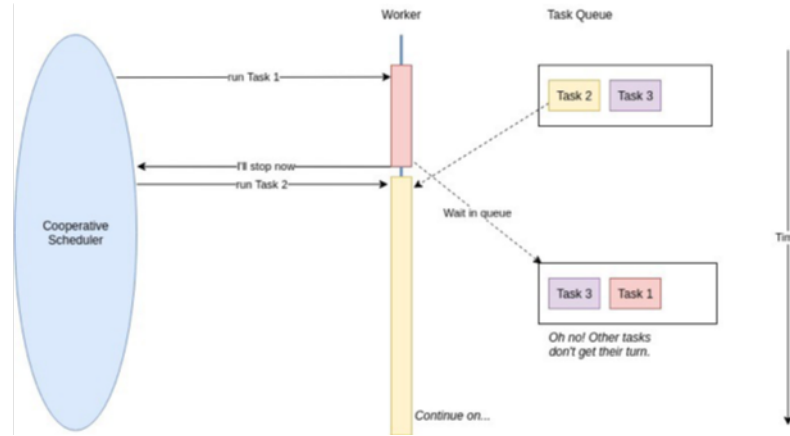


Figura 2.2: Planificación cooperativa con plazos vencidos.[1]

- **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 threshold 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 threshold de la tarea en ejecución.
- **Planificación de suspensiones preemptive diferidas.** Cada tarea puede ser ejecutado en un periodo non-preemptive. 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 non-preemptive son definidas por el programador insertando primitivas específicas en el código que habilitan y deshabilitan el modo preemptive. Dado que el tiempo inicio de ejecución en cada región no se especifica en el modelo, se considera que los puntos están flotando en el código, aunque cumpliendo con una duración que no excede a su quantum.
 - **Modelo de activación por triggers:** Las regiones non-preemptive son activadas 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

2. ANTECEDENTES

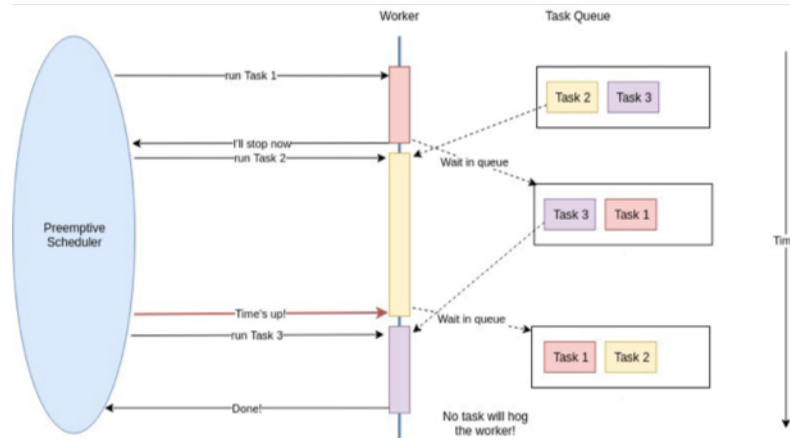


Figura 2.3: Planificación preemptiva.[1]

quantum, con lo que se decide si se suspende o continua en ejecución.

- **Puntos preemptive fijos.** Una tarea se ejecuta implícitamente en modo non-preemptive 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 non-preemptive (también llamados subjobs). 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.

2.2.2.3 Clasificación de soporte preemptive

Existen diversas clasificaciones en que se pueden agrupar las soluciones para dar soporte a la planificación de tareas preemptive, 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 subkernels dividiendo sus grids en fragmentos más pequeños. Es decir, en vez de lanzar todos los Threads por Bloque (TB) de un kernel a la vez, sólo se van lanzando fracciones de éste a la vez. Este enfoque es útil para núcleos con muchos TB, donde cada TB tienen un tiempo de ejecución

corto. 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 la GPU y CPU comparten una variable que se utiliza para señalar las solicitudes de cambio de contexto entre los threads activos y los detenidos.
- ***Entorno de Scripts.*** Esta técnica permite manejar automáticamente los kernels dependiendo de ciertos parámetros o puntos de control, liberando al programador de realizarlo manualmente. Esta aproximación funciona especialmente para entornos con kernels pequeños, ya que para aquellos que tienen una ejecución larga es necesario cuidar el nivel de granularidad o podría llegarse a plazos vencidos.

Otra clasificación la podemos formar dependiendo de la manera en que son planificadas 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 cuál de las que actualmente 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.

Ya que, aunque no esta originalmente contemplado, es necesario discriminar entre la administración de la memoria y la administración de los núcleos que una GPU nos

2. ANTECEDENTES

va a prestar para realizar el procesamiento.

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

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

2.2.3 Algoritmos de planificación

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

La planificación en tiempo real se puede dividir en:

- Estática o Fixed Task Priority (FTP): Todas las prioridades se asignan en el momento del diseño del sistema y esas prioridades se mantienen constantes durante el tiempo de vida de una tarea.
- Dinámica o Dynamic Task Priority (DTP): Se las asignan prioridades en tiempo de ejecución, en función de los parámetros de las tareas. Su objetivo es adaptarse al progreso del sistema para buscar la configuración óptima de planificación.

2.2.3.1 Shortest Job First

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

2.2 Sistemas en tiempo real

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

Tabla 2.1: Matriz de comparación de algoritmos de planificación.

2.2.3.2 Earliest Deadline First

Earliest deadline First (EDF) es un algoritmo con prioridad dinámica, en el que la tarea con el plazo fijo más próximo tiene la máxima prioridad. Este algoritmo es óptimo para implementación sobre un único procesador, y cuando el sistema se encuentra en bajos y moderados niveles de contención de recursos y datos[13]. Ya que cuando se sobrecarga el sistema, la mayoría de las tareas obtienen una alta prioridad, lo que termina en un rendimiento disminuido.

Es un algoritmo muy extendido en sistemas en tiempo real debido a su optimalidad teórica en el campo no-preemptive, pero al momento de implementarlo en un planificador preemptive, el resultado puede acarrear un exceso de ejecución si se toma el peor caso [14]. Por ello es necesario buscar alternativas de algoritmos que tengan un mejor desempeño en tareas específicas.

2.2.3.3 Rate Monotonic

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

2. ANTECEDENTES

2.2.3.4 Deadline Monotonic

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

2.2.3.5 Least Laxity First

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

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

2.2.3.6 Gang Earliest Deadline First

Gang Earliest Deadline First (GEDF) está pensado para mejorar el desempeño de EDF durante condiciones de sobrecarga[18]. La idea principal de su funcionamiento es agrupar las tareas con plazo límite similares y dentro de cada grupo planificar las tareas con SJF [17]. El parámetro rango de grupo (Gr) determina qué tarea ingresa a qué grupo el cuál es un porcentaje de la tarea al comienzo del plazo absoluto de cada cola.

2.3 CPU

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

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

2.3.1 Arquitectura del CPU

Un CPU está compuesto principalmente por:

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

2.3.2 Manycore y Multicore

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

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

Con aplicaciones computacionales intensivas, las secciones del programa a menudo muestran una gran cantidad de paralelismo de datos. Las GPU se usan para acelerar la ejecución de esta porción código. Cuando un componente de hardware que está físicamente separado de la CPU y se utiliza para acelerar secciones computacionalmente intensivas de una aplicación, se le denomina acelerador de hardware. Se puede decir que las GPU son el ejemplo más común de un acelerador de hardware.

2. ANTECEDENTES



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

2.4 GPU

La unidad de procesamiento gráfico o GPU es un procesador especializado para tareas que requieren de un alto grado de paralelismo. Su uso más extendido es el del procesamiento de instrucciones aplicadas a campo de imágenes 2D y 3D, realizando cálculos con píxeles y texeles[19].

La tarjeta gráfica en su interior puede contener una cantidad de núcleos de un orden de cientos hasta miles de unidades que son más pequeñas y que por ende, individualmente realizan un menor número de operaciones. Esto hace que la GPU esté optimizada para procesar cantidades enormes de datos pero con programas más específicos[4]. Lo más común al utilizar la aceleración por GPU es ejecutar una misma instrucción a múltiples datos para aprovechar su arquitectura.

2.4.1 Arquitectura del GPU

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

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

- Memoria: cuentan con diferentes tipos de memoria y 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 streams: La arquitectura de una GPU está diseñada con base en la programación de streams, el cual involucra a múltiples cálculos en paralelo para un stream de datos[20].
 - Stream: Conjunto de elementos que tendrán un tratamiento similar.
 - Kernel: Tratamiento aplicado a cada elemento del stream.
 - Thread: Tratamiento ejecutado por procesador aplicado a un elemento del stream.
- Gather y Scatter: Cuando se aplica un kernel a un stream, este aplica todas sus instrucciones a cada elemento, por lo que cada elemento se almacena en una posición bien definida dentro de la memoria utilizando índices que auxilian a localizarlo, a esta acción se le conoce como Scatter.

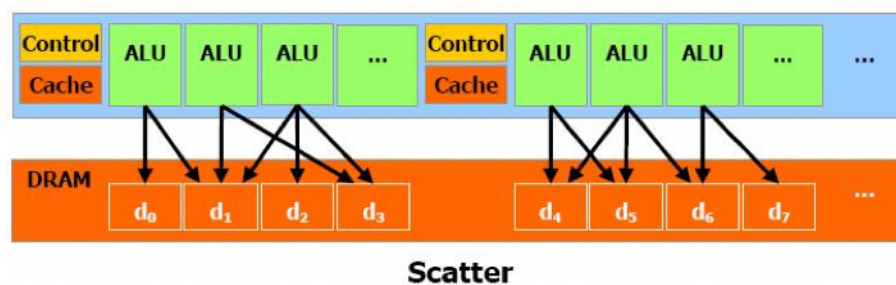


Figura 2.5: Escritura en DRAM[2].

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

2.4.2 Arquitectura CUDA

CUDA es el acrónimo en inglés de Compute Unified Device Architecture, el cual es una arquitectura de hardware y de software que permite ejecutar programas en

2. ANTECEDENTES

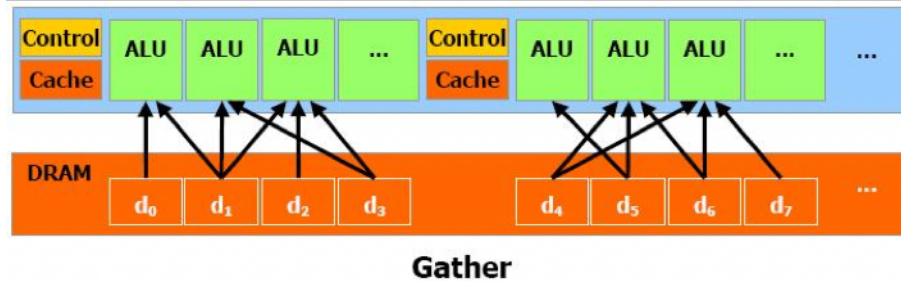


Figura 2.6: Lectura en DRAM[2].

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 host code, que tiene que ver con lo realizado por el CPU, y el device code, que ejecutará el 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 device 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 sus componentes principales:

La figura 2.7 muestra esquemáticamente el como se conforma el grid de un kernel.

Muchas veces es necesario el conocer el ID tanto del thread como del block con el que se está trabajando. En la figura 2.7 se observa el como la información para saber la posición exacta está en un formato secuencial.

Para saber ambos fácilmente se aplica el algoritmo 2.1:

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

kernel	Funciones paralelas escritas en el programa que indican que operaciones se realizaran en el GPU.
thread	Unidad mínima que ejecuta una instancia de un kernel. Tiene su propio id dentro un block, 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.

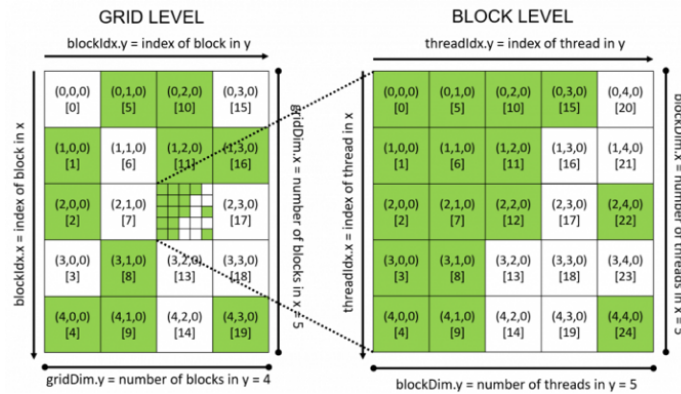


Figura 2.7: Representación de los componentes de un grid[3].

```

2
3   id_thread = threadIdx.y * blockDim.x + threadIdx.x
4

```

Algoritmo 2.1: Transformación para obtener id del thread y del block.

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

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. 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

2. ANTECEDENTES

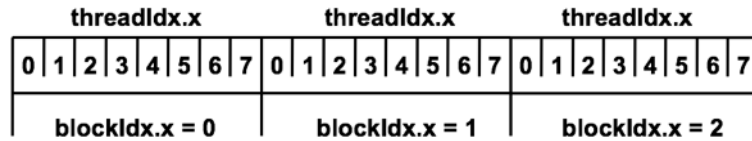


Figura 2.8: Orden de threads y blocks dentro de un grid[3].

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 device.

warp	Agrupación de threads que ejecutan la misma instrucción.
Thread Block (TB)	Varios warps constituyen un TB.
Streaming Multiprocessor (SM)	Las unidades que ejecutan los arreglos de TB.
GPU	Varias SM forman una GPU.

Tabla 2.4: Componentes de CUDA para el hardware.

la oportunidad a un nuevo kernel de la cola de ejecución, es fundamental que todos los hilos de ese TB concluyan completamente su ejecución [21].

Un es la entidad mínima ejecutable dentro del GPU y es un conjunto de 32 threads dentro de un TB, por lo que todos los hilos en su interior ejecutan la misma instrucción, y son seleccionados serialmente por el SM que los ejecutará. [22]

Una vez que se lanza un TB en un SM, todos los warps permanecen en ejecución 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 registros libres para que todos los warps de un TB puedan iniciar su ejecución.

2.4.3 Arquitectura Pascal

La principal ventaja de la arquitectura Pascal está en su construcción ya que está implementada con transistores FinFET[23], los cuales a ser de un tamaño de 16 nanómetros, permiten tener un tamaño reducido, proporcionar un rendimiento alto y obtener una gran eficiencia energética. Dicha combinación la hacen ideal para ser implementada en dispositivos embebidos que requieran ejecutar tareas híbridas.

En esta arquitectura se implementaron dos cambios significativos que ayudan a mejorar el rendimiento de los sistemas.

La primera la encontramos en la memoria unificada, la cual proporciona un único espacio de direcciones virtuales para la memoria de la CPU y GPU, permitiendo la migración transparente de datos entre los espacios de direcciones virtuales completos tanto de la tarjeta gráfica como del procesador. Esto simplifica la programación en GPUs y su portabilidad ya que, como programador, no es necesario el preocuparse por administrar el intercambio de datos entre dos sistemas de memoria virtual diferentes[24].

En la figura 2.9 podemos observar tres tipos de código, el central es el código original y que se ejecuta normalmente en un CPU, de lado izquierdo tenemos su versión en CUDA, y de lado derecho se puede observar la versión en CUDA con memoria unificada. Con lo que se puede observar que se ahorra mucho espacio de código para el manejo de la memoria entre CPU y GPU.

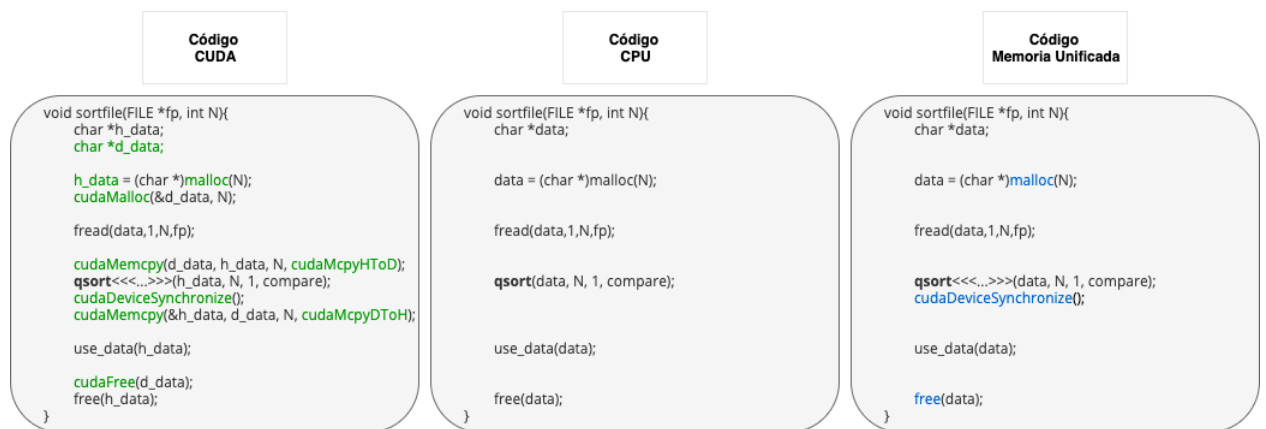


Figura 2.9: Comparación de directivas para manejo de memoria.

La segunda incorporación son la implementación de operaciones atómicas de memoria. Estas son frecuentemente usadas en el cómputo de alto rendimiento ya que

2. ANTECEDENTES

permiten que los hilos concurrentemente lean, escriban y modifiquen variables compartidas sobre la memoria unificada.

2.4.4 GPGPU

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

El cómputo de propósito general en unidades de procesamiento de gráficos o GPGPU es utilizado para acelerar el procesamiento realizado tradicionalmente por la CPU únicamente, donde la GPU actúa como un coprocesador que puede aumentar la velocidad del trabajo [25].

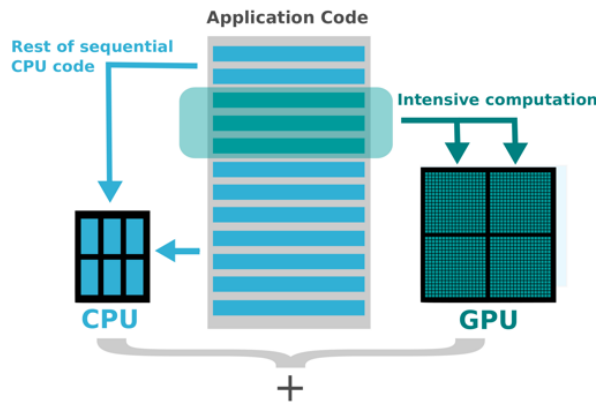


Figura 2.10: Aceleración de programas en GPUs[4].

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

2.5 Sistemas embebidos

Un sistema embebido es un sistema de cómputo diseñado para realizar tareas dedicadas, donde el mayor reto es realizar tareas específicas donde la mayoría de ellas tengan requerimientos de tiempo real [26].

2.5.1 Sistemas embebidos heterogéneos

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

Actualmente la empresa NVIDIA tiene en su catálogo sistemas embebidos heterogéneos con un gran soporte y bibliotecas para el cómputo de alto rendimiento. Dichos sistemas cuentan con la arquitectura Pascal de última generación [27], la cual permite compartir memoria entre CPU y GPU.

Debido a que la mayoría de las GPU en sistemas embebidos no son de naturaleza preemptive, es importante programar los recursos de GPU de manera eficiente en múltiples tareas [28] ya sea de planificación o memoria, lo que permite pensar en un framework que ayude a la administración de sus características.

2.6 Material de trabajo

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

En la figura 2.11 se muestra diagrama de bloques de la arquitectura del sistema Jetson TX2.

2.6.1 Jetson TX2

Las especificaciones del sistema están descritas en la tabla 2.5.

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 computadora remota.

2. ANTECEDENTES

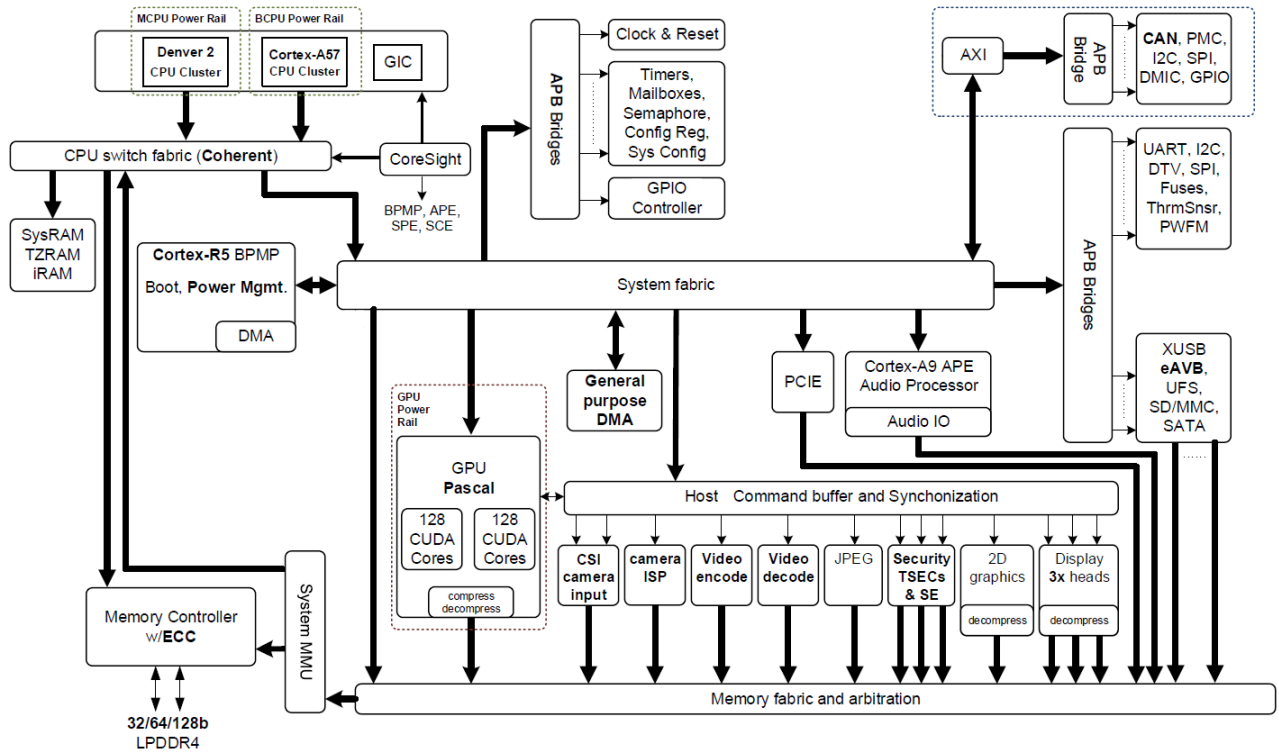


Figura 2.11: Diagrama de la arquitectura del sistema Jetson TX2[5].

- Investigación e implementación de ejercicios de GPGPU.
- Realización y modificación de ejercicios para la familiarización con la arquitectura Pascal, estructura de la tarjeta y su memoria.

2.6 Material de trabajo

Elemento	Componentes	Descripción
Arquitectura	NVIDIA Pascal GPU	256 núcleos Optimizados para un mejor rendimiento en sistemas embebidos.
CPU	Dual-Core Denver 2 64-bit CPUs + Quad-Core A57 Complex	Contiene dos clústers de procesamiento, el Denver 2 de 64 bits que se utiliza para tareas pesadas o de un sólo thread; y el ARMv8 Cortex-A57 Complex que actúa en tareas multi-thread y en cargas ligeras.
Memoria	8 GB L128 bit DDR4 Memory	DRAM de 128 bits que da soporte con un gran ancho de banda para una interfaz LPDDR4.
Almacenamiento	32 GB eMMC 5.1 Flash Storage	Integrada en el módulo.
Conectividad	802.11ac Wi-Fi and Bluetooth-Enabled Devices	
Ethernet	10/100/1000 BASE-T Ethernet	
Procesador de señales	1.4Gpix/s Advanced image signal processing Audio Processing Engine	Acelerador por hardware para captura de video y de imágenes. Subsistema que permite el completo soporte de audio multicanal por las diversas interfaces.
Video	Codificador avanzado de video HD Decodificador avanzado de video HD	Permite la grabación de video ultra-high-definition a 60 fps, soporta los estándares H.265 and H.264 BP/MP/HP/MVC, VP9 y VP8. 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[6].

2. ANTECEDENTES

Trabajo Relacionado

Al momento de realizar la revisión de la literatura se encontró que existe muy poca bibliografía sobre el tema, el problema es atribuible a la poca información que las empresas manufactureras de las tarjetas gráficas liberan al publico en general.

El trabajo [29] realizó un estudio con base en la documentación y pruebas de caja negra sobre la arquitectura arquitectura Pascal. Como caso de estudio utilizaron el sistema NVIDIA Jetson TX2, que es precisamente el se tubo disponible al realizar esta tesis.

El artículo tiene como objetivo dar las pautas para realizar normas oficiales de seguridad a los sistemas embebidos heterogéneos y así pueda certificarse su uso en ambientes críticos, pero debido a que sus diseños no están documentados o se describen a un muy alto nivel que no revelan los detalles técnicos cruciales.

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 el hardware no revelan mucho sobre las peculiaridades de como se realiza el reparto de recursos para ejecutar las peticiones, únicamente sabemos que el 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 y si este tiene suficientes recursos para recibirlo, si no, esperará hasta que pueda ser lanzado, el como realiza esta espera tampoco es

3. TRABAJO RELACIONADO

revelado por las compañías.

Para nuestro caso de estudio con el sistema Jeston TX2, el cual cuenta con 2 SM, y cada uno cuenta con 128 cores que juntos pueden procesar concurrentemente 2048 threads[30]. Ahora bien, tomando en cuenta esto, cada SM puede manejar un máximo de 64 warps.

Por ejemplo, en la tabla 3.1 se muestran diversos escenarios en los que se desea agregar kernels para procesar a la tarjeta gráfica.

Escenario	Kernel	Blocks	Threads por block	Threads Totales	Blocks SM0	Warps SM0	Blocks SM1	Warps SM1	Threads utilizados
a)	K2	4	512	2048	4	64	0	0	2048
b)	K1	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) tenemos un kernel con 4 block y cada uno de 512 threads. La primer condición para poder agregarlo recae en que la totalidad del kernel debe poder ser ejecutado en concurrente en la tarjeta gráfica en un instante de tiempo. En este caso recordemos que un SM tiene un máximo de 64 warps disponible, 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 de 16 blocks y cada uno consta de 256 threads por block. En este caso, al lanzarlo se utilizará la totalidad de los recursos del 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, y así sucesivamente hasta terminar de lanzar todos los blocks del kernel. En 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 se desperdician. En el SM1, deberían ocuparse 43.75 warps, pero se usan 44, dejando 8

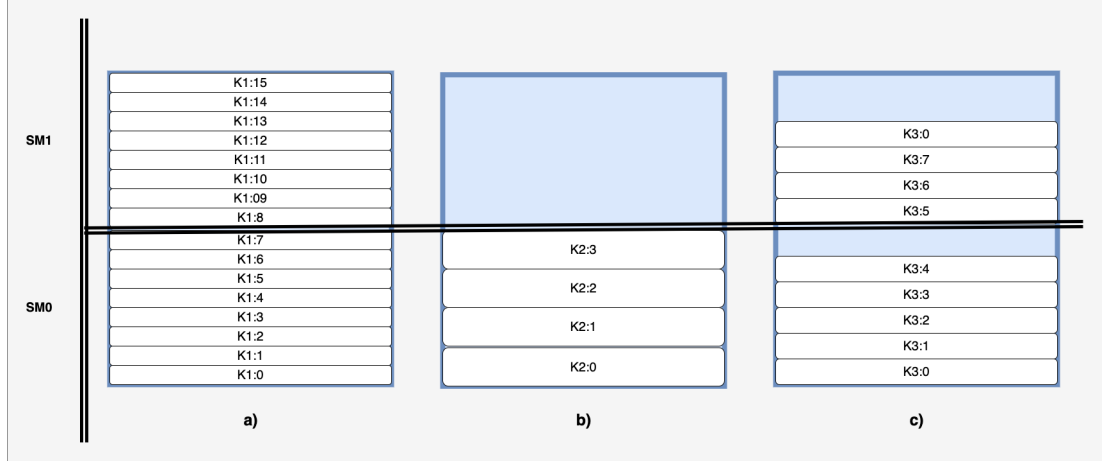


Figura 3.1: Diagrama de asignación de un kernel a los SM.

threads sin poderse utilizar.

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

Escenario	Kernel	Blocks	Threads por block	Threads por kernel	Threads en concurrente	Warps	Warps aportados a SM0	Warps SM0	Warps aportados 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. Recordemos que la restricción es que un kernel sólo puede ser asignado si existen suficientes recursos para warps en secuencia. La primera tarea utiliza 48 warps, y la segunda 80, por lo que en el primer SM podemos asignar los 48 warps del primer kernel más 16 del segundo, y en el SM colocamos los 64 restantes, ocupando

3. TRABAJO RELACIONADO

así la totalidad de localidades de la GPU.

Continuando con el caso e), ahora tenemos 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 podemos asignar los 32 warps del **K1** al SM0, justo después los 24 del **K2** y aún sobran 8 warps que muy bien sirven 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 kernels que en su conjunto representan 3072 threads, lo que nos hace pensar que muy bien pueden ser ejecutados en concurrente dentro del 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 muy bien se podrían aprovechar, la documentación actual no permite realizarlo, por ello se recurrió a una solución en la sección 4.4.1.1.

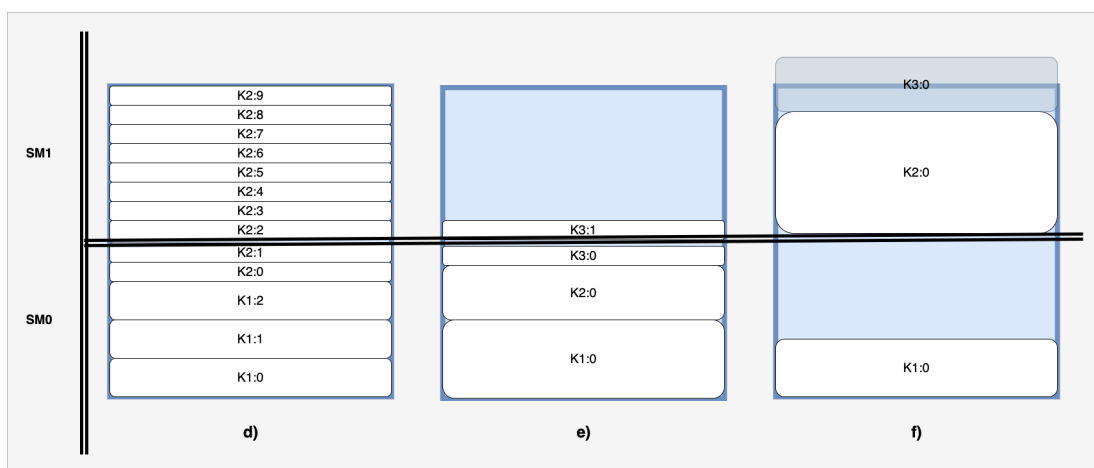


Figura 3.2: Diagrama de asignación de varios kernels a los SM.

Ahora bien, teniendo una referencia por medio de pruebas de software, se puede conocer un poco sobre funcionamiento de la asignación de kernel a la tarjeta gráfica,

se puede afirmar que es posible gestionar las peticiones de recursos a la GPU.

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

La clasificación planificación de tareas preemptive esta compuesta de diversos tipos dependiendo de sus técnicas de implementación, cómo se describe en la sección 2.2.2.3. Las soluciones basadas en hardware son costosas, ya que debemos desarrollar y construir un dispositivo que auxilie con el cambio de contexto, por ejemplo, en el artículo [31] se utilizan extensiones de hardware a modo de registros que almacenan el contexto y en general las direcciones de memoria que contienen la información necesaria para la restauración de la ejecución de un kernel.

En [32] se propone la utilización de extensiones de hardware mediante el intercambio equitativo de recursos entre los núcleos de procesamiento, esto realizando un cambio de contexto aplicando el modo preemptive en el espacio de procesamiento. En lugar de intercambiar el contexto de todo el grid, se pretende intercambiar suficientes TB de un kernel en ejecución para que haya suficientes recursos disponibles para despachar la nueva tarea.

Otra solución la observamos en [33], en donde se desarrolló un compilador y que emplea una extensión de hardware para reducir la latencia al implementar el modo preemptive, el compilador inserta puntos preemptive utilizando un análisis del ciclo de vida de los registros. Se utiliza una lógica de compresión descompresión para disminuir el tamaño del contexto de una tarea. Es decir, cuando el valor almacenado en un determinado registro es siempre igual a lo largo de la ejecución de los TB de un kernel, sólo se guardará un valor durante el cambio de contexto.

El artículo [34] implementa una serie de funciones para realizar particiones de kernel y de datos, esto realizando subkernels y dividiendo las transacciones de datos en fragmentos. Específicamente, se presenta una técnica de reescritura binaria para reconfigurar de manera transparente el código de los kernel. Mientras que para los kernel un poco más complejos, se desarrolló una técnica de transformación fuente a fuente que compila el código del kernel transformado en binarios CUDA. La prioridad de las tareas esta dada por colas de ejecución. GPES modifica el API de CUDA utilizando las bibliotecas de *openCUDA* para reconfigurar el código binario de los kernels, esto lo realiza obteniendo un máximo de blocks que se pueden ejecutar por

3. TRABAJO RELACIONADO

quantum. Para realizar esto, se ayuda de dos implementaciones, por una parte realiza una transformación fuente a fuente para así apoyarse de en realizar la implementación de la partición de la transferencia de datos.

En el artículo [35] se propone un framework de planificación que parte los kernels del GPU y genera secuencias de lanzamiento en subkernels dinámicamente para entrar el modo preemptive con la implementación de un divisor de carga de trabajo y de un planificador de tareas. Utiliza un divisor de carga de trabajo que divide el kernel GPU en múltiples subkernels durante el tiempo de ejecución para implementar el modo preemptive. Dependiendo del estado actual de sistema y de la prioridad, el divisor de carga de trabajo decide el número y el tamaño de cada subkernel.

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

El trabajo [36] 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 del 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 CPU y un proxy de este en el lado del GPU. Dicho daemon planifica el momento en que los kernel deben comenzar, reanudarse o detenerse en la GPU, y dependiendo de la acción se notifica el proceso del CPU que lanzó el kernel.

Como muestran los resultados del trabajo, esta solución funciona bien para kernels con ejecución pequeña, por que al tener en el sistema aquellos que salen de la media, la granularidad del TB limita el retraso mínimo preemptive que se puede lograr, resultando muy seguramente en plazos vencidos.

Es importante mencionar que el artículo [37] es el primer trabajo que genera un framework para utilizar tareas en tiempo real en tarjetas gráficas. Este trabajo entra dentro de la categoría de colas masivas en paralelo, ya que se basa en la partición en fragmentos de memoria a procesar, cada fragmento es agregado a una cola de

procesamiento para ser ejecutado. Su solución es dividir las transacciones de copiado de memoria en varios fragmentos para insertar puntos preemptive. Esto también garantiza que sólo las tareas de mayor prioridad se ejecuten en el GPU en cualquier momento, y así evitar interferencias de rendimiento causadas por lanzamientos concurrentes.

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

El artículo [10] se basa en preguntar continuamente si ha terminado el quantum de una tarea, en caso afirmativo la saca de 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 del 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.

El artículo [38] propone la creación del framework schedGPU, el cual utiliza el administrador de trabajo Slurm para planificar las tareas. Este framework administra las múltiples solicitudes para acceder a las GPU de forma segura al garantizar que no se produzcan sobrecargas de memoria durante su ejecución. Este acceso es controlado mediante bloqueos de archivos, señales del sistema y exclusión mutua.

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

La tarea es modificada únicamente al llamar explícitamente las funciones de la

3. TRABAJO RELACIONADO

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

El artículo [39] presenta una técnica para la ejecución en GPUs llamada "*Planificación de recursos compartidos con reserva de presupuesto*" o por sus siglas en inglés *BR-SRS*, la cual limita el número de núcleos de procesamiento de una GPU para una tarea basándose en su prioridad, esto lo realiza modificando las bibliotecas de *OpenGL-ES*. Así se previene que una tarea que se encuentra en segundo plano retrase a otra que se encuentra en ejecución, también se minimiza la sobrecarga de planificación al invocarse solamente dos veces, en el inicio de la tarea y en su finalización.

El único trabajo que utiliza algoritmos para la planificación de tareas en tiempo real, hasta el momento de la revisión del estado del arte es GPUart [27]. Permite la implementación preemptive dentro de los TB y cada uno de estos subkernels se pueden planificar bajo las políticas de Earliest Deadline First (EDF) y de aquellos algoritmos que mantengan la prioridad de las tareas fijas.

GPUart se centra en las GPU integradas, 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 DMA.

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

3.1 Resumen

Este capítulo presenta los trabajos relacionados con el tema de esta tesis, se analizan

- Planificación de EDF preemptive limitado de sistemas con tareas esporádicas (*Limited Preemption EDF Scheduling of Sporadic Task Systems*);
- Planificación de recursos espaciales compartidos con prioridad para unidades de gráficos embebidos (*Priority-driven spatial resource sharing scheduling for*

3.1 Resumen

Ref.	Artículo	Clasificación por implementación	Clasificación por planificación	Clasificación por Modificación
[31]	Enabling preemptive multiprogramming on GPUs	Basado en Hardware: Añade registros para almacenar contexto	Colas masivas en paralelo	Modificación del API
[32]	Simultaneous Multikernel GPU: Multitasking throughput processors via fine-grained sharing	Basado en Hardware: Selector de núcleos de procesamiento	Administración dinámica de los núcleos de procesamiento	Modificación del API
[33]	Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching	Basado en Hardware: Analizador del ciclo de vida de registros	Administración dinámica de memoria	Modificación del API
[34]	GPES: A preemptive execution system for gpgpu computing	Basado en Software: Partición de kernel	Administración dinámica de memoria	Modificación del API
[35]	Run-Time Scheduling Framework for Event-Driven Applications on a GPU-Based Embedded System*	Basado en Software: Partición en tareas	Administración dinámica de la memoria	Modificación del código fuente
[36]	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
[37]	RGEM: A Responsive GPGPU Execution Model for Runtime Engines	Basado en Software: Partición de kernel	Colas masivas en paralelo	Modificación de código fuente
[10]	Preemption of a CUDA Kernel Function	Basado en Software: Partición de kernel	Colas masivas en paralelo	Modificación del API
[38]	Intra-Node Memory Safe GPU Co-Scheduling	Basado en Software: Partición de kernel	Administración dinámica de la memoria	Modificación del API
[39]	Priority-driven spatial resource sharing scheduling for embedded graphics processing units	Basado en Software: Partición en tareas	Administración dinámica de los núcleos de procesamiento	Modificación del API
[27]	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.

embedded graphics processing units);

- Framework para planificación en tiempo de ejecución de aplicaciones con manejo de eventos en sistemas embebidos basados en GPU (*Run-Time Scheduling Framework for Event-Driven Applications on a GPU-Based Embedded System*);
- Sobre planificación dinámica para el GPU, sus aplicaciones en computación gráfica y más (*On Dynamic Scheduling for the GPU and its Applications in Computer Graphics and Beyond*); y
- REGM: Un modelo de ejecución GPGPU responsivo para soluciones en tiempo

3. TRABAJO RELACIONADO

de ejecución (*REGM: A Responsive GPGPU Execution Model for Runtime Engines*);

- Planificación conjunta con GPU y aseguramiento de la memoria intra-nodo (*Intra-Node Memory Safe GPU Co-Scheduling*);

Diseño

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

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

4.1 Descripción general del framework

La solución propuesta se encuentra dentro de las siguientes clasificaciones:

- **Clasificación por implementación:** *Basado en Software. Partición de Kernel.*
- **Clasificación por planificación:** *Planificación por prioridad.*
- **Clasificación 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 bloques agrupa las bases necesarias para el funcionamiento del framework.

El framework está dividido en dos zonas de implementación, la primera tiene que ver con aquellas actividades que son propias con del Host, como lo es el protocolo de

4. DISEÑO

lanzamiento de los kernel (ver 4.4. En el caso del manejo de la memoria, debido a que la tarjeta Jetson TX2 utiliza una arquitectura Pascal (ver 2.4.3), se cuenta con una memoria unificada con lo que se simplifican el manejo de las copias de memoria entre el en Host y el Device, resultando en que el módulo **Memoria** (ver 4.3) pertenezca a ambas zonas, también se presenta el almacenamiento de los contextos de cada una de las tareas.

En la zona de implementación Device encontramos con el módulo **Puntos Preemptive** (ver 4.2), 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** (ver 4.4.1), ya que es en donde se dan las pautas para realizar la planificación de las tareas que se ejecutarán en un determinado momento en el Device. Pero para poder realizar dicha prioridad, se plantea el módulo **Asignación de prioridades** (ver 4.5) el cual se encargará de seleccionar dentro de un conjunto de tareas aquella que tiene la mayor prioridad en un momento específico de tiempo.



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

Como se detallará más adelante, esta solución no es transparente al programador, es necesaria la modificación del código fuente, aunque en un inicio parecería que el rendimiento sería inferior al realizar comprobaciones continuas del estado del quantum, el modificar las bibliotecas del API de CUDA o el compilador del dispositivo ni tampoco la implementación de analizadores sintácticos para la lectura de directivas precompiladas está dentro de las posibilidades de acción del proyecto. Entonces es necesario modificar el código fuente, con esto también no debemos pensar en seleccionar los puntos más cercanos al óptimo para colocar directivas de precompilación.

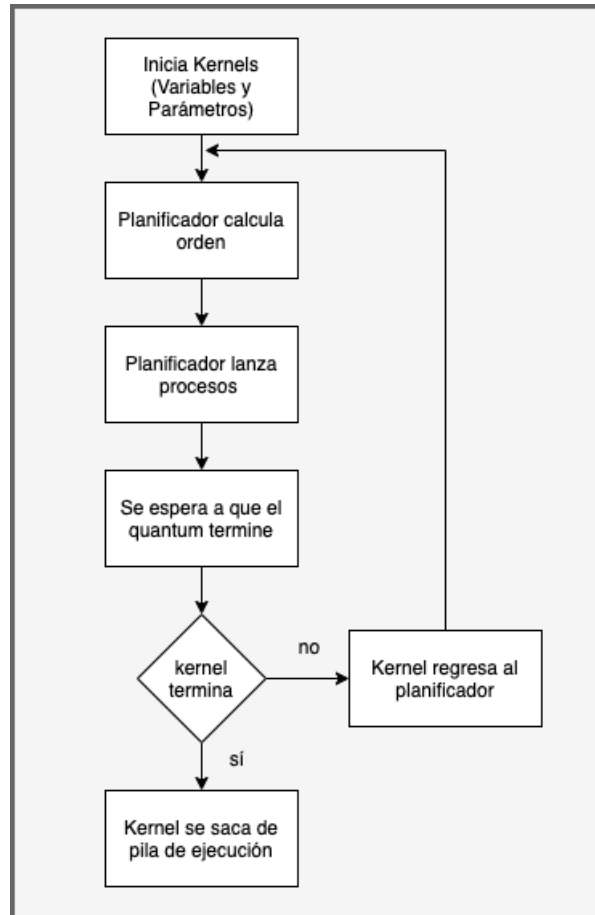


Figura 4.2: Diagrama del flujo del framework.

4.1.1 Precondiciones necesarias

La precondición más importante es que el framework debe ser implementado en un programa 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 permiten apuntadores complejos basados en objetos. No se permite el llamado a funciones no rastreables. El quantum de las tareas debe ser similar para que aquellas que estén en ejecución terminen en

4. DISEÑO

tiempos similares.

El número de threads por block debe ser menor o igual a la cantidad de threads disponibles en cada SM.

Los contextos de cada kernel deben poder coexistir en la memoria al mismo tiempo para que se puedan ejecutar y suspender en cada punto preemptive.

4.2 Puntos preemptive

En una aplicación acelerada por el cómputo gráfico muchas veces se implementan más de una función kernel, y en el momento en que ejecutamos varias aplicaciones en el GPU habrá alguna que mantenga en sobretiempo los recursos causando así un retraso en la ejecución en general de todo el sistema.

Este módulo permite gestionar la actividad de un kernel a nivel de aplicación, aquí se marca la pauta el punto exacto donde se podrá realizar la administración del contexto de una tarea en ejecución, contará con tres casos principales, si se está iniciando el proceso, si está a la mitad de una ejecución o si ya ha terminado, con esto se podrá liberar las unidades de procesamiento para dar lugar a otras tareas de consumir recursos.

Se propone una serie de puntos de control que se incluirán explícitamente dentro del código que se desea implementar en modo preemptive, esto durante básicamente tres etapas iterativas del ciclo de vida de un kernel a)inicio, b)en ejecución y c)finalización.

Este elemento tiene como objetivo que cada que se alcance alguno de los puntos de control dentro de un kernel y sea necesario detener su ejecución, se guarde una copia de su contexto actual en una estructura de datos para que cuando sea nuevamente su oportunidad de ejecución se reanude como si nunca se hubiera detenido.

Una vez que una tarea, independientemente de en que momento de su ciclo de vida se encuentre seguirá ejecutándose en la GPU hasta que complete su cálculo o termine su quantum.

Al momento de lanzar la tarea siguiente en ejecución se inicializaran todas las variables necesarias en el nuevo contexto por medio de estructura copia de seguri-

4.2 Puntos preemptive

dad. Cuando se está en la etapa de inicio de un kernel, se inicializan tanto los datos necesarios para el funcionamiento de este en su cuerpo y en la estructura de datos.

Al inicio del algoritmo 4.1 la función kernel está ligeramente modificada en sus parámetros, ya que es necesario que reciba la estructura *Backup* en donde se almacenará su contexto cuando se presente una suspensión preemptive, y se recibirá el apuntador al estado del quantum, dicho valor arrojará *true* cuando se haya concluido el tiempo del quantum.

Como se mencionó anteriormente, esta solución es completamente basada en software, por lo que se debe modificar la función kernel, para mantener una convención que ayude a mitigar posibles problemas, todas las declaraciones de variables se deberán realizar en la primer fase, la cual se encuentra en las primeras líneas del kernel.

Las únicas declaraciones con inicialización permitidas en esta fase son aquellas que designan la posición tanto de los thread como de los blocks dentro de un grid, esto porque su información es necesaria en cada una de las siguientes fases. La única variable que es necesaria para todos kernels es *id_block*, el que servirá para en las siguientes fases para extraer la información de la estructura *Backup*.

```
1 __global__ void kernelFunction(int* a,..., int* resultados,...,
                                struct Backup* backup, bool* quantum_expirado){
2     /* Fase de declaración de variables */
3
4     /* Variable posición de thread y block */
5     int id_block = blockIdx.y * gridDim.x + blockIdx.x;
6     int id_thread = threadIdx.y * blockDim.x + threadIdx.x;
7
8     /* Variables locales */
9     int temp1;
10    int temp2;
11    . . .
12    int tempn;
13    . . .
14
15    /*Contadores*/
16    int i;
17    int j;
18    . . .
19    int k;
20    . . .
```

4. DISEÑO

Algoritmo 4.1: Fase de declaración de variables.

Enseguida pasamos a la fase de la inicialización (ver algoritmo 4.1) de cada una de estas variables y como se muestra en el algoritmo 4.2 nos apoyamos apoyándonos de una estructura *switch-case* con tres casos dependiendo del estado de cada block. Para seleccionar cada uno de los casos debemos leer el valor que se encuentra en la estructura de copia de seguridad, esto por que hay que recordar que el kernel por si solo no sabe si es la primera vez que se ejecuta o es el producto de un cambio de contexto dentro del sistema.

Cada uno de estos tres diferentes estados es:

- **INICIO**: Es el primer estado, y 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**: Este estado es el que se presenta una vez que ya se han inicializado las variables en el estado anterior, o cuando el planificador le da otra vez la oportunidad de ejecutarse para terminar el procesamiento. Aquí se copia la información de la estructura de copia de información a las variables locales, para trabajar con la información como si nunca se hubiera suspendido el kernel.
- **TERMINADO**: Debido a que muchas veces dentro de un kerel hay blocks que finalizan su procesamiento antes que otros, es necesario indicar que esa sección ya terminó y no requiere hacer ningún cálculo.

```
1      . . .
2
3      /* Fase de inicialización */
4      switch(backup.estado[id_block]){
5          case INICIO:
6              //Inicialización de variables locales
7              temp1 = 0;
8              temp2 = 0;
9              . . .
10             tempn = 0;
11
12             //Inicialización de contadores
13             i = 0;
14             j = 0;
```

```

15     . . .
16     i = 0;
17     break;
18     case EJECUCION:
19         //Inicialización de variables con respecto al backup
20         temp1 = backup->temp1[id_block * blockDim.x + id_thread];
21         temp2 = backup->temp2[id_block * blockDim.x + id_thread];
22         . . .
23         temp3 = backup->tempn[id_block * blockDim.x + id_thread];
24
25         //Inicialización de contadores con respecto al backup
26         i = backup->i[id_block];
27         j = backup->j[id_block];
28         . . .
29         k = backup->k[id_block];
30         break;
31     case TERMINADO:
32         break;
33 }
34 . . .

```

Algoritmo 4.2: Fase de inicialización.

Una vez inicializadas todas las variables podemos realizar el procesamiento objetivo del kernel (ver algoritmo 4.2). Para ello nuevamente preguntamos a la estructura de copia de seguridad el estado individual de cada block, dependiendo de lo que responda a cada uno, se realiza:

- **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 resolver una parte del kernel original, esto se realiza dentro de una estructura *do-while* para que al menos se realice una vez antes de que, o el quantum haya expirado, o se haya completado el procesamiento. Si algo de esto se cumple se rompe el ciclo y se pregunta si ya se completo el procesamiento, si es así, el estado del block en el backup se modifica a *TERMINADO* y finaliza ese block sin realizar copia de seguridad para ahorrar tiempo de procesamiento. En caso que no haya sido completado, significa que el quantum expiró, por lo que se deben guardar todas las variables locales en su correspondiente espacio designado dentro del backup, terminado esto, se finaliza el block.

4. DISEÑO

- **TERMINADO:** En dado supuesto que se llegue a este caso, significa que se lanzó nuevamente el kernel por que existen blocks que aún no terminan su trabajo, con lo que este simplemente termina su ejecución.

```
1  . . .
2
3  /* Fase de procesamiento*/
4  switch(backup.estado[id_block]){
5      case INICIO:
6          if(id_thread == 0)
7              backup->estado[id_block] = EJECUCION;
8          _synctreads(); // Todos esperan a que se modifique el estado
9          // No hay break para que continúe al siguiente caso
10     case EJECUCION:
11         //Ejecución del kernel
12         do {
13             //Procesa
14             resultados = #paso_de_procesamiento;
15         } while(!quantum_expirado && !block_completo);
16
17         /* Si se realizó la ejecución completamente */
18         if(block_completo){
19             if(id_thread == 0)
20                 backup->estado[id_block] = TERMINADO;
21             break;
22         }
23         _synctreads(); //TB sincronizan para llegar al mismo valor de contador
24
25         /* Si ya expiró el quantum almacena contexto en backup
26     */
27     //Variables locales
28     backup->temp1[id_block * blockDim.x + id_thread] = temp1;
29     backup->temp2[id_block * blockDim.x + id_thread] = temp2;
30     . . .
31     backup->tempn[id_block * blockDim.x + id_thread] = tempn;
32     //Contador
33     if(id_thread == 0){
34         backup->i[id_block]=i;
35         backup->j[id_block]=j;
36         backup->k[id_block]=k;
37     }
38     break;
39 case TERMINADO:
40     break;
41 }
```

41

42 . . .

Algoritmo 4.3: Fase de procesamiento.

4.2.1 Condición de carrera

La fase de procesamiento (ver algoritmo 4.3) es un procedimiento en el que hay que poner especial atención ya que es donde se concentra el núcleo de las operaciones del kernel, aparte 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 ello 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 thread 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 el arreglo *estado* en la estructura de copia de estado.

Finalmente, si el procesamiento se realiza con ayuda de contadores, al momento de que se expire el quantum, todos los threads deberán suspenderse cuando lleguen al mismo valor, por lo 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*.

4.3 Memoria

4.3.1 Almacenamiento del contexto

Es necesario la creación de 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 una estructura, por lo que para cada uno de los kernel, se debe crear una estructura *ad hoc*.

4. DISEÑO

La estructura *backup* (ver algoritmo ??) almacena tres tipos de valores, 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 de cada bloque. 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 el de tipo contador. Dependiendo del cálculo que se esté realizando, muchas veces se deberán paralelizar *estructuras for* sin dependencia de datos, por esta razón, puede 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 block. Como todos llegaron a ese punto, se puede simplemente guardar un valor del contador. En caso de que se esté utilizando contadores que son propiamente controlados por un punto de verificación de quantum, se deberá utilizar el formato de variable local.

Finalmente, debemos incluir un arreglo más que nos ayude a guardar el estado en que se quedó un block al ser detenido por el planificador.

```
1 //Estructura BackUp
2 struct Backup{
3
4     /* Variables locales */
5     int temp1[blockDim*gridDim];
6     int temp2[blockDim*gridDim];
7     . . .
8     int tempn[blockDim*gridDim];
9
10    /*Contadores*/
11    int i[gridDim];
12    int j[gridDim];
13    . . .
14    int k[gridDim];
15
16    /* Último punto de control */
17    int estado[gridDim];
18
19 };
```

Algoritmo 4.4: Estructura Backup para almacenar el contexto.

4.3.2 Variables compartidas

Al momento de realizar una solución de GPGPU, se debe tener en cuenta que existirán variables que se deben mantener visibles tanto para el host como para el device. En el algoritmo 4.5 de la sección 4.4 tenemos ciertas variables que deben ser compartidas entre ambos lados.

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 estamos en el dominio de la memoria unificada, ambos parámetros existirán en la memoria global para que estén disponibles para ambos dispositivos.

4.4 Lanzamiento del kernel

Dos precodiciones que plantea el framework (ver 4.1.1) estrechamente relacionadas son 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 sistema.

Cada una de las aplicaciones GPGPU que se ejecutarán en el sistema embebido deberán estar agrupadas en cabeceras de C. Al inicio de la ejecución del framework, se ejecutarán las aplicaciones de forma concurrente (Figura 4.3) para que todas alcancen el punto en que requieren realizar cálculos en la GPU.

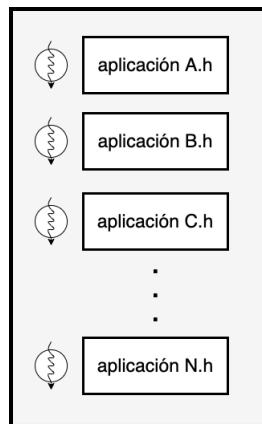


Figura 4.3: Aplicaciones en ejecución concurrente.

4. DISEÑO

Código de cada una de las aplicaciones se le debe añadir una serie de variables y parámetros para que el planificador pueda programar su ejecución.

Se debe encapsular la llamada a la función kernel para que el planificador permita dar el orden de lanzamientos. Dentro de cada enclave se debe implementar una serie de variables y banderas para que el contexto pueda ser almacenado al alcanzar a cada punto preemptive.

Es necesaria la definición de la 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 debemos inicializar *quantum_expirado* en **false** para indicar que se tiene tiempo de ejecución. La bandera *continuar_eje* será modificada por el planificador para permitir la ejecución del kernel. Una vez que sea planificado para su ejecución, se lanzará el kernel y se esperará el tiempo definido para el quantum. Terminado este tiempo, se cambiará el estado de *quantum_expirado* a **true** y se sincronizarán todos los threads del kernel con *cudaDeviceSynchronize()* para cerciorarnos de que se terminó la ejecución del grid.

Ahora debemos cambiar el estado de *continuar_eje* a **false** para que la tarea permanezca suspendida hasta que el planificador permita una nueva ejecución. Finalmente, se pregunta si todos los blocks completaron su trabajo.

```
1 //kernelN.h
2
3 void kernelN(bool* continuar_eje, bool* kernel_completado){
4
5     //Declaración del backup
6     Backup backup;
7     //Duración máxima del quantum en microsegundos
8     int quantum_time = #Duración del quantum
9     //
10    bool quantum_expirado;
11
12
13    /* Ejecución del kernel */
14    while(!kernel_completado){
```

```

15     quantum_expirado = false;
16
17     if(continuar_eje == true){
18
19         kernelFunction<<<blockDim,gridDim >>>(a, ... , resultados,
20                                             backup,quantum_expirado);
21         /*Espera el tiempo del quantum */
22         usleep(quantum_time);
23         quantum_expirado = true;
24         cudaDeviceSynchronize();
25         continuar_eje = false; // automáticamente detiene
26                                 //su ejecución terminado el quantum
27
28         kernel_completado=kc(backup.estado);
29     }
30 }
31 }

```

Algoritmo 4.5: Algoritmo para lanzamiento del kernel en el lado del host.

Para poder determinar si un kernel ha terminado completamente su procesamiento, nos auxiliamos de la función *kc* (Algoritmo 4.6). Simplemente se pasa como parámetro el arreglo *estado* de la estructura *backup* y se pregunta si el estado de todos los blocks es **TERMINADO**, regresando *true*.

```

1 bool kc(int* estado){
2     for (int i = 0; i < gridDim; i++)
3         if (estado[i] != TERMINADO)
4             return false;
5     return true;
6 }

```

Algoritmo 4.6: Función kernel completo.

4.4.1 Planificador

El módulo principal del framework es el de el **Planificador**, porque es aquí donde se realiza toda la calendarización de las tareas a ser ejecutadas en la GPU, trabaja muy estrechamente con el módulo de la asignación de prioridades, el cual le da la pauta para poder elegir aquella tarea que tenga más relevancia en un momento determinado en el tiempo.

4. DISEÑO

Para poder planificar un conjunto de tareas 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 que conozcamos el número máximo de threads que pueden estar en ejecución concurrentemente.

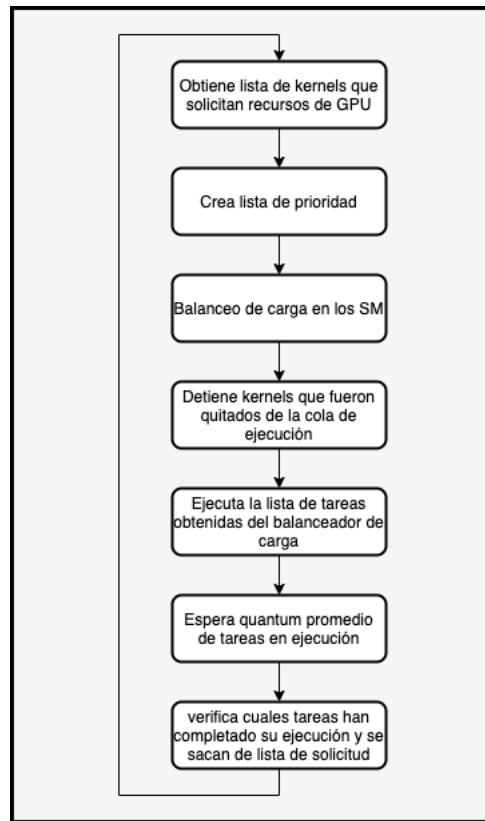


Figura 4.4: Diagrama de flujo del planificador.

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

```
1 //Estructura Task
2 struct Task{
3     int id;
4     int costo;
5     bool* continuar_eje;
```

```
6  bool* kernel_completado;
7  }
```

Algoritmo 4.7: Estructura task.

4.4.1.1 Balanceador de carga

Como se vió en la 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 lanzo **K1**, seguido de **K2** y al final **K3**, derivando en que se desperdicien recursos y no sea posible ejecutar un kernel. Pero si se modifica este orden, se puede optimizar la asignación de tareas en las localidades, como lo vemos en el ejemplo b).

Por esta razón, se ideó un balanceador de carga que permitirá la maximización de la planificación de tareas ejecutables en un quantum.

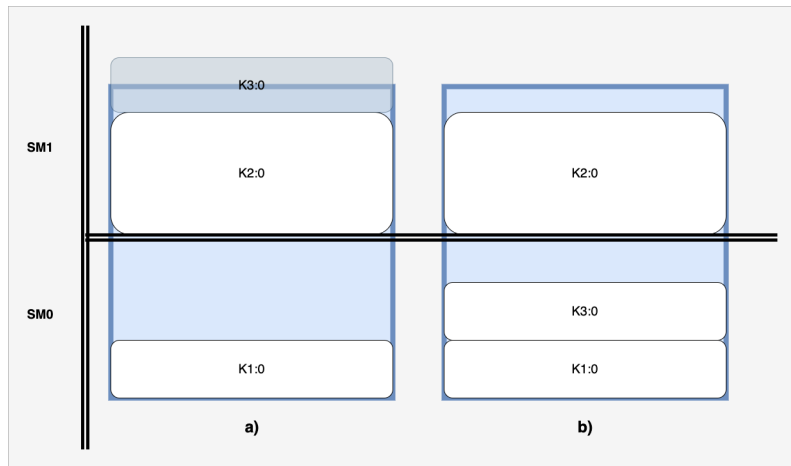


Figura 4.5: Diagrama de balanceo de carga de kernels en los SM.

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.

Debido a que las tareas que se ejecutarán sobre el sistema se presuponen embebidas, se considera que probablemente se podrán ejecutar al menos dos kernels en

4. DISEÑO

concurrente en la tarjeta, por lo que es imprescindible que ellas siempre tengan recursos suficientes para su ejecución, en caso de que la tarea de mayor prioridad absorba todos los recursos, no será necesario ejecutar la segunda tarea en ese quantum.

La lógica de asignación de tareas a los SM se basa en colocar la mayor cantidad posible de kernels en concurrente para que se puedan realizar más cálculos en menor tiempo.

A continuación se mostraran algunos de los diversos escenarios en forma de casos de estudio que pudieran aparecer al recurrir al balanceador de carga. Cada uno está representado en un diagrama, dicho diagrama poseerá en la parte superior la lista ordenada por prioridad (mayor a menor), cada tarea contiene el nombre del kernel así como el número de blocks que se requieren lanzar seguido del número de threads que los comprende. Se sombrea aquellas tareas que lograron ejecutarse en cada iteración, también se tiene una división en la lista, de lado izquierdo se comprenden las tareas que pudieran correr en concurrente en un punto en el tiempo de ejecución, de lado derecho aquellas que salen de dicha cuenta, justo en la última tarea de la primera mitad se tiene la suma de los threads que hipotéticamente pudieran ejecutarse en concurrente.

El Caso I (ver figura 4.7), representa un escenario en donde la totalidad de las tareas que estaban ejecutándose en la iteración anterior a) vuelven a quedar en la posición de mayor prioridad en la lista en b), puesto que ninguna tarea tuvo que ser sacada de ejecución, se les dará oportunidad de un siguiente quantum sin necesidad de detenerse guardar su contexto y relanzarse.

Una vez terminado el quantum de la iteración y recalculando la lista de prioridades observamos que se presenta el Caso II (ver figura 4.8) donde algunos de las tareas ahora cambiaron el orden en la lista. Procedemos a calcular el rango de las tareas que pudieran correr en concurrente, y encontramos que la mayoría puede continuar en ejecución por que se encuentran del lado derecho, aquella que quedaron de lado derecho se deben sacar de la lista de ejecución y ser insertadas en la lista de paro de actividades para dar oportunidad de ejecución a aquellas que tienen mayor prioridad, para este caso serían los kernel **KE**, **KF** y **KI**. Y procedemos a agregar las del lado derecho por orden de prioridad, **Kk** y **KL**. Una vez agregadas las tareas de mayor prioridad, se observa que aún queda espacio y se pregunta a las tareas de menor prioridad si sus características cumplen con los requisitos para que puedan ser ejecutadas en esa iteración, con lo que encontramos que la tarea **KE** puede ingresar a ejecutarse.

4.4 Lanzamiento del kernel

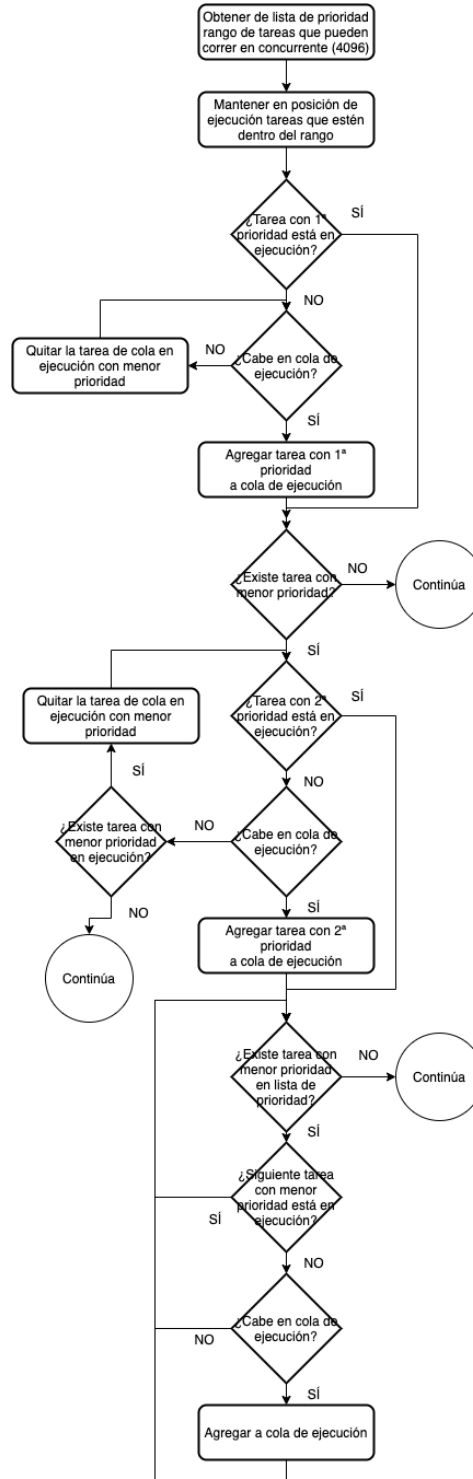


Figura 4.6: Diagrama de flujo del balanceador de carga.

4. DISEÑO

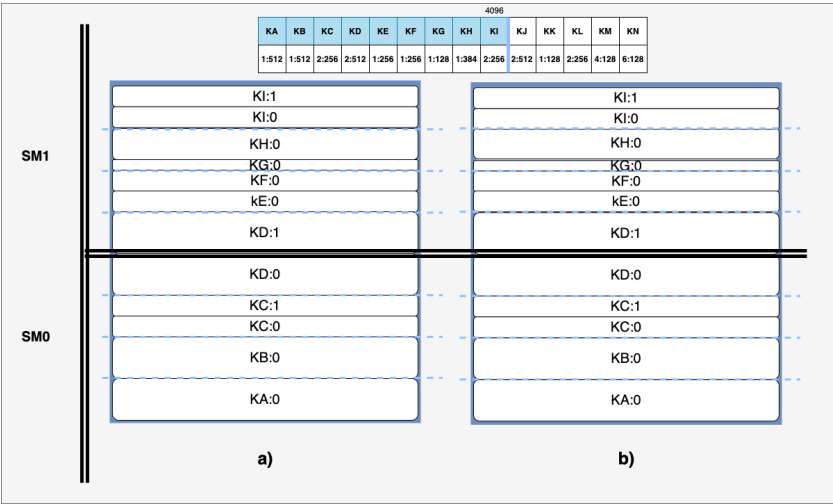


Figura 4.7: Caso I Kernels mantienen su prioridad en la siguiente iteración.

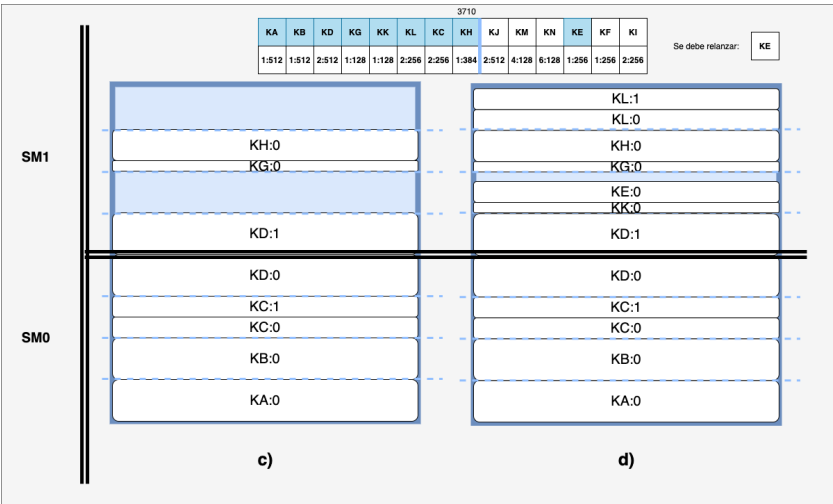


Figura 4.8: Caso II Kernels de baja prioridad deben relanzarse.

4.4 Lanzamiento del kernel

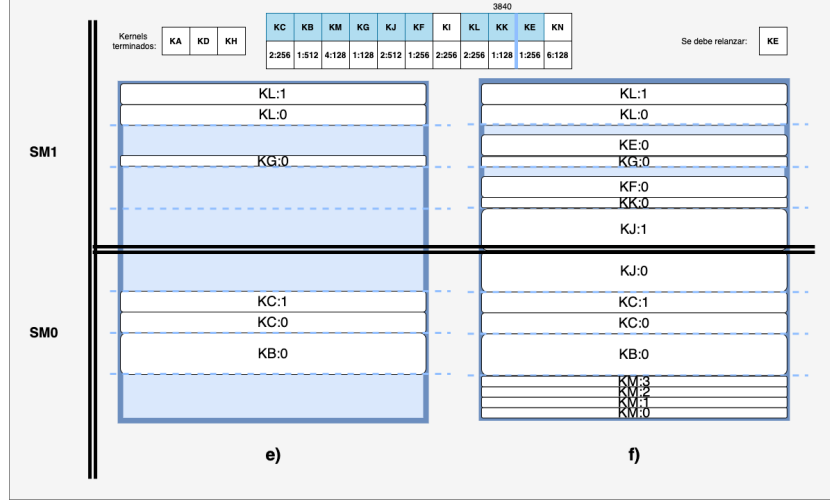


Figura 4.9: Caso IIIa Kernels que completaron su ejecución.

Después de terminada la iteración d), se ha encontrado con que las tareas **KA**, **KD** y **KH** han completado su ejecución, por lo que son eliminados de la lista de peticiones, por ello la lista de prioridades para esta ejecución se modificó tanto en orden, como en longitud. Una vez calculado el rango de las posibles tareas en concurrente, se encuentra que varias de ellas ya no existen o bien deben ser quitadas de ejecución para dar lugar a otras con mayor prioridad. En esta ocasión nos quedamos únicamente con **KB**, **KC**, **KG** y **KL**. Y procedemos a rellenar el espacio existente con las tareas según su orden de prioridad.

Pero en el Caso III (ver figura 4.9), debido a la forma en que están dispuestas las tareas, teóricamente la tarea **KI** debería poder ejecutarse por que hay suficientes threads para su ejecución, pero el hecho de que no estén contiguos imposibilita esta acción. Por ello se salta y se decide agregar tareas de menor prioridad, incluso **KE** que está del lado derecho de la lista de prioridad. Debido a que nuevamente tenía una baja prioridad y tuvo que ser quitada de ejecución, y se le dio oportunidad de ejecutarse, sin importar si fuera en su misma ubicación, debe detener su ejecución, guardar su contexto y volver a ejecutarse.

Siguiendo con el flujo del planificador, (ver figura 4.10) se vuelve a obtener una nueva lista de ejecución, en donde el lado derecho nos indica que se podrían lanzar sus tareas y se podrían utilizar los 4096 threads de los SM, pero al momento de

4. DISEÑO

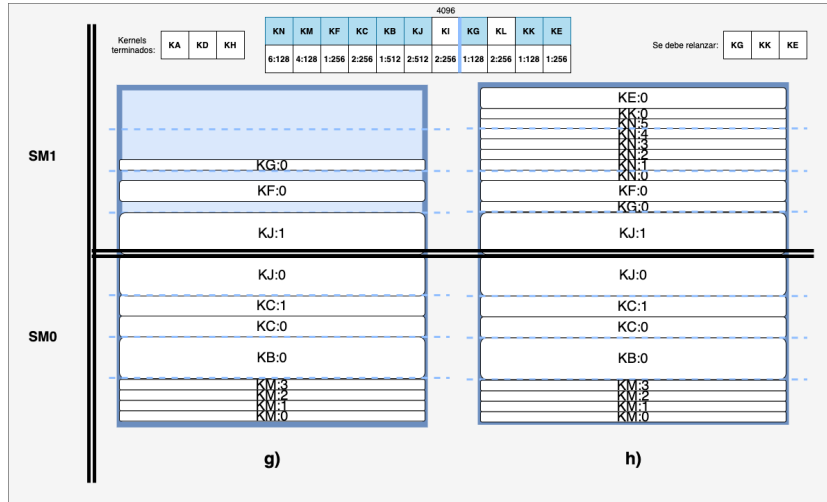


Figura 4.10: Caso IIIb Kernels que completaron su ejecución.

llenar la cola de ejecución por prioridad, nuevamente la tarea **KI** se queda fuera de ejecución, pero ganamos recursos para 3 tareas de menor prioridad **KG**, **KK** y **KE**, y como dichas tareas se encontraban en ejecución en la iteración pasada, pero ahora son de baja prioridad, se deben detener para ser lanzadas nuevamente.

En el Caso IV (ver figura 4.11) se tiene que la lista ordenada de prioridad de lado izquierdo contiene tareas que no ocupan por completo los recursos aportados por los SM, pero la primer tarea del lado derecho por sus características no puede ser ejecutada, por ello no fue seleccionada en primer lugar, por ello es necesario empezar a llenar el espacio sobrante con las tareas de baja prioridad. En este caso maximizamos la totalidad de los recursos de los SM y ejecutamos toda la lista de prioridad a excepción de la tarea **KJ**. Y nuevamente como inicialmente no estaban planeadas para su ejecución en esta iteración, se debe detener y relanzar las tareas **KG**, **KK** y **KE**.

Después de diversas iteraciones del ciclo de vida del planificador, nos encontramos con el escenario k) (ver figura 4.12), donde tenemos que se encuentran en ejecución un cierto número de tareas esparcidas por los los SM que utilizan solamente una fracción de sus recursos. Justo después que se acaba el quantum de su iteración, se calcula nuevamente la lista de prioridad en l) y ahora han aparecido 2 tareas en la cabeza de la cola que solicitan recursos, debido a que la de mayor prioridad no puede recibir recursos contiguos para su ejecución, se procede a sacar la tarea de menor prioridad actualmente en ejecución **KP**, y pregunta si ahora puede entrar en

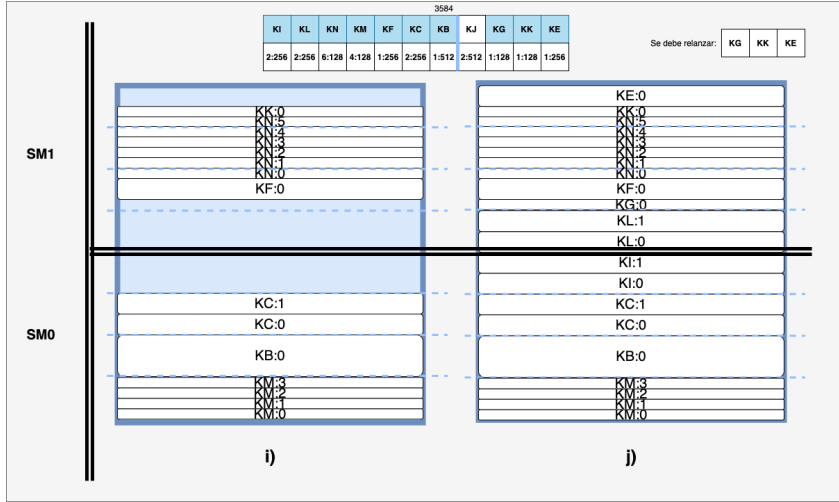


Figura 4.11: Caso IV Kernels muy grandes.

ejecución.

Como no es posible que ingrese, se vuelve a quitar de ejecución la tarea con la menor prioridad (ver figura 4.13), y así sucesivamente hasta que se reúnan las características que se requiere. Una vez liberado el espacio en \tilde{n}), se añade la tarea con la mayor prioridad en o), y se pregunta ahora por el espacio requerido por la tarea con la segunda mayor prioridad, como nuevamente no se puede agregar a ejecución, se eliminan las tareas de menor prioridad hasta que sea posible colocarla.

Una vez liberado el espacio contiguo necesario para la tarea **KT**(ver figura 4.14), y así sucesivamente hasta que se reúnan las características que se requiere. Una vez liberado el espacio en \tilde{n}), se añade la tarea con la mayor prioridad en o), y se pregunta ahora por el espacio requerido por la tarea con la segunda mayor prioridad, como nuevamente no se puede agregar a ejecución, se eliminan las tareas de menor prioridad hasta que sea posible colocarla.

Liberado el espacio necesario (ver figura 4.15), se procede a colocar la tarea **KU** y posteriormente se rellena el espacio con las tareas de la lista ordenada. Como se sacaron de ejecución las tareas **KW**, **KS**, **KY**, **KR** y **KP**, y posteriormente entraron nuevamente a la lista, es necesario relanzarlas.

En el Caso VI (ver figura 4.16), nos encontramos en q) que la tarea **KZ** ya se

4. DISEÑO

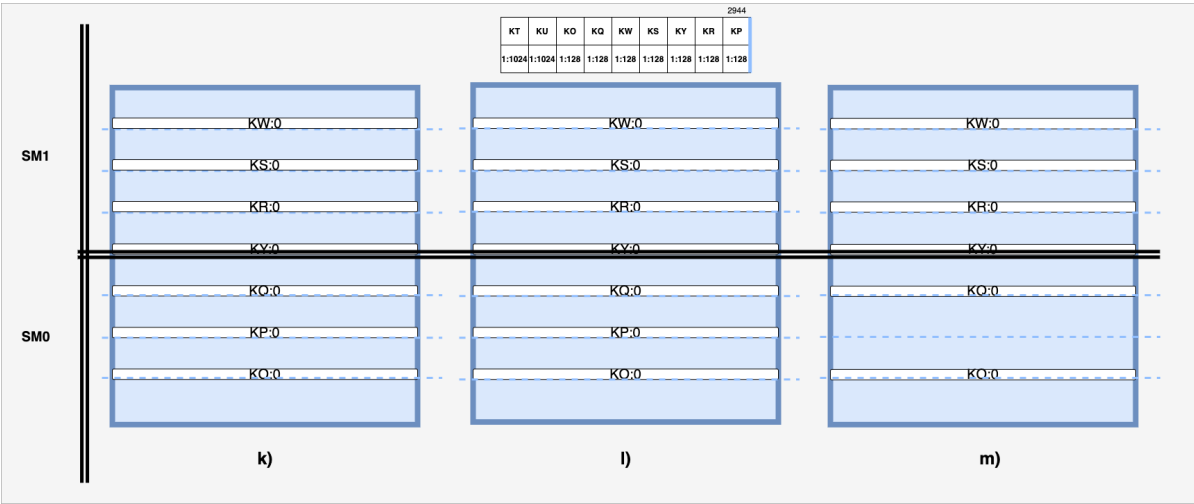


Figura 4.12: Caso Va Kernels de mayor prioridad que no estaban en ejecución.

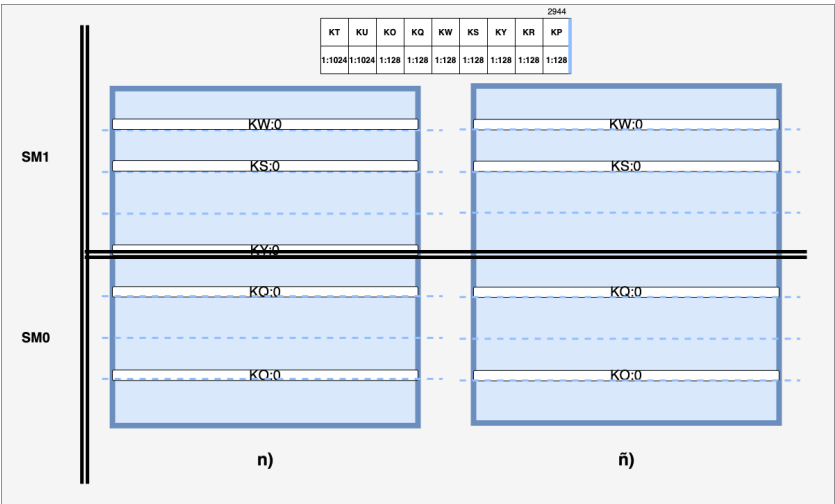


Figura 4.13: Caso Vb Kernels de mayor prioridad que no estaban en ejecución.

4.4 Lanzamiento del kernel

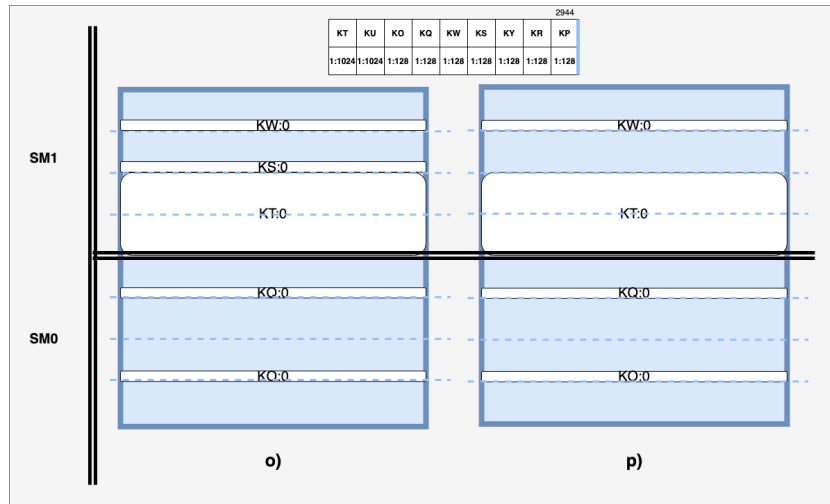


Figura 4.14: Caso Vc Kernels de mayor prioridad que no estaban en ejecución.

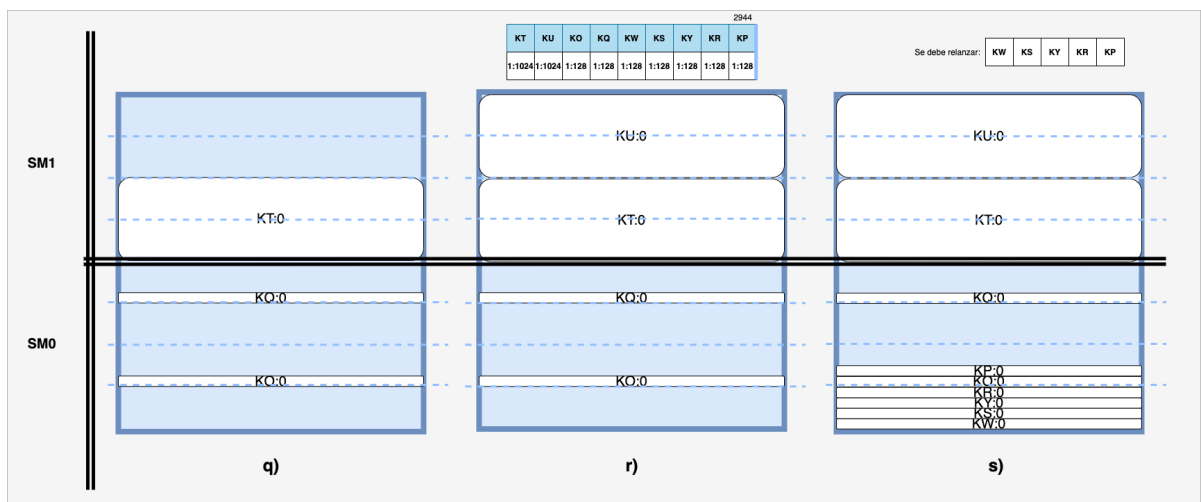


Figura 4.15: Caso Vd Kernels de mayor prioridad que no estaban en ejecución.



Figura 4.16: Caso VI.

encontraba en ejecución, pero en la siguiente ejecución, se presenta la tarea **KW** con una prioridad menor, aunque teóricamente ambas tareas podrían ejecutarse concurrentemente, una tarea con menor prioridad no puede mover a una de mayor, por lo que en esta iteración únicamente se quedará en ejecución la tarea con la mayor prioridad.

4.5 Asignación de prioridades

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

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 prioridad ordenando las tareas de mayor a menor, según sea el caso.

Rendimiento

Este capítulo propone posibles métricas para evaluar el rendimiento del framework.

5.1 Métricas de rendimiento por kernel

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 permita valorar la tendencia de los resultados.

- **Número total de subkernels:** Denota el número de veces que se suspendió el kernel.

$$n_{sk} \tag{5.1}$$

- **Tiempo de ejecución de subkernel:** Duración promedio de la ejecución de un subkernel una vez que es restablecido de una suspensión preemptive.

$$t_{sk} = \sum_{i=1}^{n_{sk}} \frac{t_{sk}(i)}{n_{sk}} \tag{5.2}$$

- **Número total de cambios de contexto:** Denota el número de veces que se guardó y extrajo el contexto.

$$n_{bk} = n_{sk} - 1 \tag{5.3}$$

5. RENDIMIENTO

- **Tiempo de inserción del backup:** Tiempo promedio que se utiliza para almacenar el contexto de un subkernel en la estructura de backup.

$$t_{bk[in]} = \sum_{i=1}^{n_{bk}} \frac{t_{bk[in]}(i)}{n_{sk}} \quad (5.4)$$

- **Tiempo de extracción del backup:** Tiempo promedio que se utiliza para copiar el contexto almacenado en la estructura de backup al subkernel.

$$t_{bk[ex]} = \sum_{i=1}^{n_{bk}} \frac{t_{bk[ex]}(i)}{n_{sk}} \quad (5.5)$$

- **Tiempo de cambio de contexto:** Tiempo total que duran las inserciones y extracciones de un kernel.

$$t_{tbk} = t_{bk[in]} + t_{bk[ex]} \quad (5.6)$$

- **Tiempo de ejecución total con suspensión preemptive:** Nos indica el tiempo total desde que el programa inicia la primera vez, hasta que finaliza completamente, este valor trae consigo el tiempo que estuvo esperando el kernel al ser nuevamente lanzado después de una suspensión preemptive, la mayoría de las veces será mayor a 1 ya que se deben realizar operaciones añadidas al kernel original.

$$t_p = t_f - t_i \quad (5.7)$$

- **Tiempo de ejecución total sin suspensión preemptive:** Indica el tiempo total de ejecución del kernel, únicamente el tiempo que está activo dentro del GPU.

$$t_{np} = \sum_{i=1}^{n_{sk}} t_{sk}(i) \quad (5.8)$$

- **Tiempo de ejecución real total:** Indica el tiempo real de procesamiento sin contar con el tiempo de cambio de contexto.

$$t_r = t_{np} - t_{tbk} \quad (5.9)$$

5.2 Métricas de rendimiento multikernel

- **Tiempo de ociosidad:** Indica el tiempo que un kernel está fuera de operación.

$$t_{idle} = t_p - t_{np} \quad (5.10)$$

- **Tasa relativa de ejecución:** Muestra la relación que existe entre el tiempo de ejecución total con suspensión preemptive y el tiempo de ejecución original del kernel.

$$t_{tr} = \frac{t_{np}}{t_{or}} \quad (5.11)$$

5.2 Métricas de rendimiento multikernel

5. RENDIMIENTO

Conclusiones

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

Partiendo de la hipótesis presentada en el Capítulo 1:

6.1 Contribuciones

Las contribuciones del presente trabajo son:

-

6.2 Trabajo futuro

A continuación se muestran algunos temas de trabajos futuros:

-

6. CONCLUSIONES

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] W.-M. Hwu, *NVIDIA CUDA Compute Unified Device Architecture*, version 2. ed., NVIDIA, 2009. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4814979>
- [3] John Cheng, Max Grossman, and Ty McKercher, *Professional CUDA C Programming*, 2014.
- [4] NVIDIA, “COMPUTACIÓN ACELERADA: Supera los desafíos más importantes del mundo.” [Online]. Available: <https://la.nvidia.com/object/what-is-gpu-computing-la.html>
- [5] 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/>
- [6] plawrence Jsachs, “Jetson TX2 Developer Kit,” pp. 1–24, 2017. [Online]. Available: <https://developer.nvidia.com/embedded/jetson-tx2-developer-kit>
- [7] 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
- [8] D. R. Salvador Sánchez, Miguel Ángel Sicilia, *Ingeniería del software: un enfoque desde la guía SWEBOK*. Alfaomega Grupo Editor, S.A. de C.V, 2012.

BIBLIOGRAFÍA

- [9] G. C. Butazzo, *Hard real-time computing systems: Predictable scheduling algorithms and applications*. Springer Science & Business Media, 1998, vol. 36, no. 3.
- [10] 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.
- [11] 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.
- [12] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*. Pearson Education, Inc., 2014, vol. 2. [Online]. Available: <http://www.amazon.com/dp/0136006639>
- [13] 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.
- [14] S. Heath, *Embedded systems design*. EDN Series For Design Engineers, 2009, vol. 24.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] A. STANCU, E. CODRES, and M. M. Guerrero, *Jetson TX2 and CUDA Programming*, second edi ed., NVIDIA, 2018.

- [20] S. Rennich, “Cuda c/c++ streams and concurrency,” 2012. [Online]. Available: <http://scholar.google.com/scholar?hl=en{\&}btnG=Search{\&}q=intitle: CUDA+C++Streams+and+Concurrency{\#}0{\%}5Cnhttp://scholar.google.com/scholar?hl=en{\&}btnG=Search{\&}q=intitle: Cuda+c/c++streams+and+concurrency{\#}0>
- [21] C. Cooper, “GPU Computing with CUDA Lecture 2 - CUDA Memories,” Chile, 2011. [Online]. Available: <http://www.bu.edu/pasi/files/2011/07/Lecture2.pdf>
- [22] Y. Lin and V. Grover, “Using CUDA Warp-Level Primitives,” 2018. [Online]. Available: <https://devblogs.nvidia.com/using-cuda-warp-level-primitives/>
- [23] NVIDIA, “ARQUITECTURA PASCAL DE NVIDIA Computación infinita para oportunidades infinitas.” [Online]. Available: <https://www.nvidia.com/es-la/data-center/pascal-gpu-architecture/>
- [24] 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>
- [25] 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>
- [26] 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.
- [27] 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.
- [28] Kristin 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>
- [29] 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.

BIBLIOGRAFÍA

- [30] 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/>
- [31] 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.
- [32] 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.
- [33] 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.
- [34] 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.
- [35] 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.
- [36] 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.
- [37] 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.
- [38] 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.

BIBLIOGRAFÍA

- [39] 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.