**\section{Descripción general del framework}\label{secc:descGen}**

**%Clasficación y tipo de modificación.**

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 sintacticos para la lectura de directivas precompiladas está dentro de las posibilidades de acción del proyecto.

Entonces s necesario modificar el código fuente, con esto tambien no debemos pensar en seleccionar los puntos más cercanos al óptimo para colocar directivas de precompilación.

La solución propuesta se encuentra dentro de las siguientes clasificaciones:

\begin{itemize}

\item Clasificación por implementación: \textbf{Basado en Software. Partición de Kernel.}

\item Clasificación por planificación: \textbf{Planificación por prioridad.}

\item Clasificación por modificación: \textbf{Modificación de código fuente.}

\end{itemize}

Está basado en Software ya que no se requiere de la incorporación o modificación de ningún elemento físico

**%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 implmentación del modo preemptive.

No se permite la memoria dinámica ni compartida entre kernels.

No se permiten apuntadores complejos basados en objetos.

No se permiten los GPU streams.

No se permite el llamado a funciones no rastreables.

**\section{Puntos preemptive}\label{secc:puntosPreemptive}**

**%Descripción general del módulo.**

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 explicitamente 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 seguridad. 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.

**%Precondiciones necesarias.**

**%Por qué existe?**

**%Qué especificación tiene y cómo se maneja?**

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 las primeras lines de la función.

Posteriormente pasamos a la etapa de la inicialización de cada una de estas variables y nos apoyamos apoyandonos de una estructura \textit{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:

\begin{itemize}

\item \textit{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.

\item \textit{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.

\item \textit{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.

\end{itemize}

Una vez inicializadas todas las variables podemos realizar el procesamiento objetivo del kernel. 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:

\begin{itemize}

\item \textit{INICIO}: Como se acaba de lanzar el kernel por primera vez, unicamente se cambia el estado del block a \textit{EJECUCION}, y como ahora se tiene un nuevo valor se puede ingresar al siguiente estado dentro del mismo switch.

\item \textit{EJECUCION}: Al entrar en este caso, en primera instancia se realiza el paso de procesamiento para resolver una parte del kernel origianl, esto se realiza dentro de una estructura \textit{do-while} para que almenos 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 \textit{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.

\item \textit{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.

\end{itemize}



**%Comparación, como manejan los demás trabajos.**

**\section{Memoria}\label{secc:Mem}**

**%Descripción general del módulo.**

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.

**%Clasficación y tipo de modificación.**

**%Precondiciones necesarias.**

**%Por qué existe?**

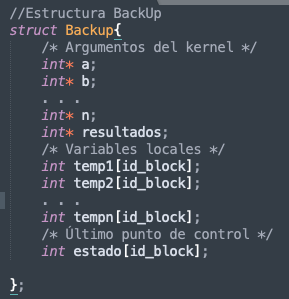
**%Qué especificación tiene y cómo se maneja?**

Todos los parámetros y variables que se encuentren dentro de una función kernel deben almacenrase en una una estructura, por lo que para cada uno de los kernel, se debe crear una estructura \textit{ad hoc}.

Por ejemplo en el código siguiente, debido a que los argumentos desde un inicio son pasado por medio de apuntadores, se deben almacenar de la misma forma, así evitamos combinar tipos de variables y lo mejor es que mantenemos unicamente la copia original para el tiempo de vida del kernel.

Pero para las variables locales esto no se cumple, sumado a esto, aunque cada uno de los blocks tiene declarada la misma variable, esta contiene un valor diferente, por lo que en la estructura debemos crear un arreglo que almacene cada uno de los valores, pudiendo despues recuperar su información por el id del bloque que lo almacenó.

Finalmente debemos incluir un arreglo más que no ayude a guardar el estado en que se quedó un block al ser dentenido por el planificador.



**%Comparación, como manejan los demás trabajos.**

**\section{Planificación}\label{secc:plan}**

**%Descripción general del módulo.**

Debido a que la planificación se realiza sobre un sistema embebido, como se muestra en la sección \ref{sec:seh}

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. Si termina su tiempo de ejecución significa que el kernel no está completo por lo que dependiendo del algoritmo que se esté utilizando, se decidirá cual es la tarea que próxima a ejecutar.

**%Clasficación y tipo de modificación.**

**%Precondiciones necesarias.**

**%Por qué existe?**

**%Qué especificación tiene y cómo se maneja?**

**%Comparación, como manejan los demás trabajos.**

La mayoría de los trabajos relacionados realizan una planificación por medio de colas masivas en paralelo, aunque esto ayuda a que no existan ejecuciones fuera de orden , todas las tareas tienen la misma prioridad y eso en una implementación real no es viable.