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

**%Clasficación y tipo de modificación.**

Como se detallará más a delante, 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 por que no se requiere de la incorporación o modificación de ningún elemento físico,**

**\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?**

**%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.**

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.

**%Por qué existe?**

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

**%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.