Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Implementación de Control Predictivo Basado en CUDA

Autor: Andrés Álvarez Cía

Tutor: José María Maestre Torreblanca

Dep. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Implementación de Control Predictivo Basado en CUDA

Autor:

Andrés Álvarez Cía

Tutor:

José María Maestre Torreblanca

Profesor Contratado Doctor

Dep. Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2017

Proyecto Fin de Grado: Implementación de Control Predictivo Basado en CUDA

Autor: Andrés Álvarez Cía
Tutor: José María Maestre Torreblanca

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

Agradecimientos

Aprovecho estas líneas para agradecer a todos los profesores que he conocido en mi formación; de cada uno he aprendido algo que, en suma, se dispone en este trabajo para simbolizar todo el esfuerzo y estudio de la etapa que ahora acabo.

Agradezco también a mi tutor la oportunidad de realizar este trabajo con él y el entusiasmo mostrado en que desarrollara este tema.

Andrés Álvarez Cía
Sevilla, 2017

Resumen

Este trabajo pretende reunir una metodología para abordar el problema de control predictivo basado en modelo razonablemente más rápido que de la forma tradicional. Para ello se propone que el procesamiento de la aplicación sea conjunto entre la CPU y la GPU, cuya principal característica es el tratamiento de grandes volúmenes de datos en paralelo.

Con este propósito, se presenta en primer lugar el modelo de programación CUDA®, desarrollado por Nvidia® para programar sobre unidades de procesamiento gráfico. Se tratan las principales características de esta tecnología y se explica lo esencial para poder iniciarse en el uso de este lenguaje, con idea de utilizarlo en la aplicación de control.

A continuación se da la oportunidad de hacer la implementación en MATLAB®, ya que facilita la elaboración de código para ser ejecutado en la GPU a cambio de una gestión de sus recursos menos detallada. Se llegará a una solución de compromiso de manera que se aproveche lo mejor de cada uno.

Por último, se abordará el problema de control predictivo tomando como partida los avances y trabajos realizados por Gade-Nielsen, Jorgensen y Dammann (2012), Brand et al. (2011) y Brand y Chen (2011), llegando a realizar una comparativa de cuánto tiempo tarda en simularse el mismo problema con cada procedimiento aquí explicado.

Abstract

This paper aims to gather a methodology to approach the problem of model predictive control reasonably faster than in the traditional way. To this end, it is proposed to use both the CPU and the GPU for computations, so that large amounts of data can be treated in parallel.

For this purpose, the CUDA® programming model developed by Nvidia® for programming on graphic processing units is presented first. The main features of this technology are discussed and the essentials are explained to use this language in control applications.

Next implementation in MATLAB® is covered, since it facilitates the programming in exchange for a less detailed management of its resources. A compromise solution is reached so that the best of both worlds can be enjoyed.

Finally, the problem of predictive control is tackled taking as a starting point the advances and works done by Gade-Nielsen, Jorgensen and Dammann (2012), Brand et al. (2011), and Brand and Chen (2011). A comparison of how long it takes to simulate the same problem with each procedure explained here is also provided.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

A^*	Conjugado
c.t.p.	En casi todos los puntos
c.q.d.	Como queríamos demostrar
■	Como queríamos demostrar
e.o.c.	En cualquier otro caso
e	número e
IRe	Parte real
IIm	Parte imaginaria
sen	Función seno
tg	Función tangente
arctg	Función arco tangente
sen	Función seno
$\sin^x y$	Función seno de x elevado a y
$\cos^x y$	Función coseno de x elevado a y
Sa	Función sampling
sgn	Función signo
rect	Función rectángulo
Sinc	Función sinc
∂y ∂x	Derivada parcial de y respecto
$x_\circ$	Notación de grado, x grados.
Pr(A)	Probabilidad del suceso A
SNR	Signal-to-noise ratio
MSE	Minimum square error
:	Tal que
<	Menor o igual
>	Mayor o igual
\	Backslash
⇔	Si y sólo si

1 INTRODUCCIÓN A CUDA

1.1 Introducción

1.1.1 Qué es un GPU

Una unidad de procesamiento gráfico o GPU (Graphics Processor Unit) es un coprocesador que se encuentra en las tarjetas gráficas y que coopera en ciertas tareas de procesamiento más intensivo como son las operaciones de gráficos y los cálculos en coma flotante, de manera que alivia de esta carga de trabajo a la unidad de procesamiento central o CPU y las aplicaciones se ejecutan con mayor velocidad. Se tiene por lo tanto una unidad principal que ejecuta la mayoría del código y otra unidad que, al estar optimizadas para cálculos intensivos, agiliza aquellas tareas que requieren de su potencia, como pueden ser la codificación de video o el renderizado 3D.

Básicamente GPU dispone de una gran cantidad de núcleos organizados en multiprocesadores. Permite de esta manera un alto grado de procesamiento en paralelo gracias a la ejecución concurrente de hilos, resultando así un dispositivo específicamente dedicado al tratamiento masivo de datos.

La CPU por su parte no cuenta con más de uno o un par de núcleos optimizados para el procesamiento en serie secuencial y se dedica más al control del flujo y al almacenamiento en caché de datos.

Se puede observar la arquitectura simplificada de cada una en esta figura tomada de Nvidia®[1]

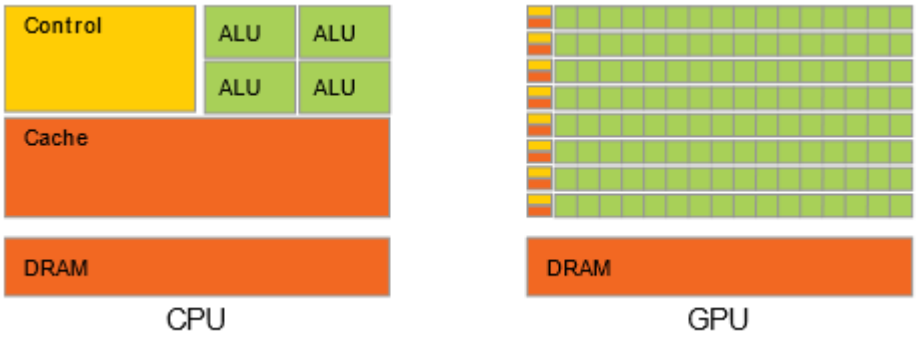

Figura 1-1. CPU vs GPU (Nvidia, 2016).

En sus orígenes, su funcionalidad solo se aplicaba al procesamiento de imágenes y gráficos pero en su exitosa evolución se están acelerando algoritmos tan diversos como los relacionados con el procesamiento general de señal, simulaciones físicas, computación financiera o biología computacional: es común hablar ya por tanto de GPUs de propósito general.

[1] Nvidia® 2016. En: CUDA C Programming guide. [Consulta 20 febrero 2017]. Disponible en: http://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf

1.1.2 Qué es CUDA

CUDA (Compute Unified Device Architecture) se presenta en noviembre de 2006 por Nvidia® como "una plataforma de computación paralela de propósito general y un modelo de programación que aprovecha el motor de cálculo paralelo de las GPUs Nvidia para resolver problemas computacionales complejos de una forma más eficiente que con una CPU." [2]

El entorno software es lenguaje C, por lo tanto se está programando en alto nivel, con unas mínimas extensiones que permiten explotar las capacidades de la tarjeta gráfica. Aparte de C, los desarrolladores también pueden programar sus aplicaciones en lenguajes como FORTRAN, DirectCompute u OpenACC, así como usar librerías de MATLAB.

A la hora de trabajar con CUDA se presentan una serie de ideas que cambian la forma tradicional de programar una aplicación en C; la principal es la implementación de un código que será ejecutado en paralelo, a diferencia de la programación en serie que se viene desarrollando en los excasos núcleos de la CPU.

Tal y como se ha visto, se tiene más capacidad de cálculo al combinar código secuencial ejecutado por la CPU con código ejecutado en paralelo por la GPU. El concepto de computación heterogénea precisa entonces determinar qué partes del código se dispondrán en cada uno, con el fin de sacar el máximo partido y optimizar la aplicación, como se ilustra en la figura 1-2.

El programador deberá dar trabajo a los núcleos multiprocesadores de las tarjetas y aprovechar este paralelismo para descomponer el problema en sub-problemas de los que se encarguen dichos núcleos.

Tales sub-problemas son resueltos por bloques de hilos, compartiendo memoria y con unas ciertas barreras de sincronización, funciones que son expuestas utilizando las extensiones del lenguaje a las que se hacía alusión al principio.

La realidad es que en el mercado existen multitud de arquitecturas GPUs: desde productos profesionales donde priman los altos rendimientos, con gran cantidad de núcleos por multiprocesador, hasta una gama más asequible al usuario que busca acelerar sus aplicaciones. Este amplio rango de características que se pueden encontrar en una GPU comercial evidencia que el paralelismo de las aplicaciones se escala según el número de núcleos procesadores disponibles y que gracias a la descomposición de las tareas en grupos de hilos, no hace falta reescribir el código sino que se ejecuta automáticamente (ver figura 1-3). Por consiguiente, en función de la disponibilidad de elementos de procesamiento, el programa ejecutará independientemente cada grupo de hilos en cualquier orden, ya sea secuencial o concurrentemente. Asimismo, a mayor número de núcleos mayor velocidad de ejecución.

[2] Nvidia® 2016. En: CUDA C Programming guide, cap. 1, Introduction. [Consulta 20 febrero 2017]. Disponible en: http://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf

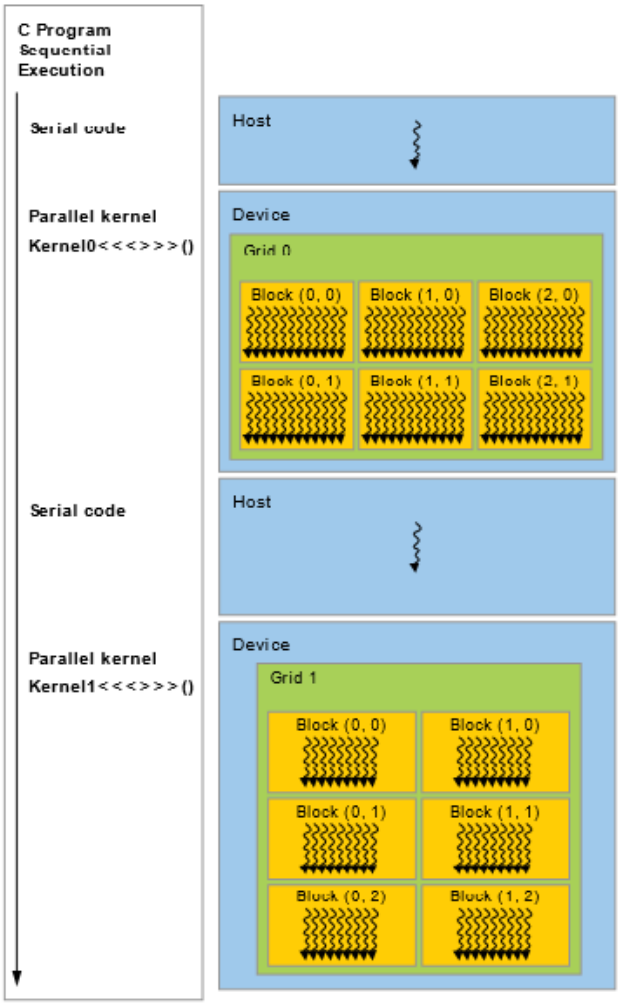

Figura 1-2. Programación heterogénea (Nvidia, 2016).

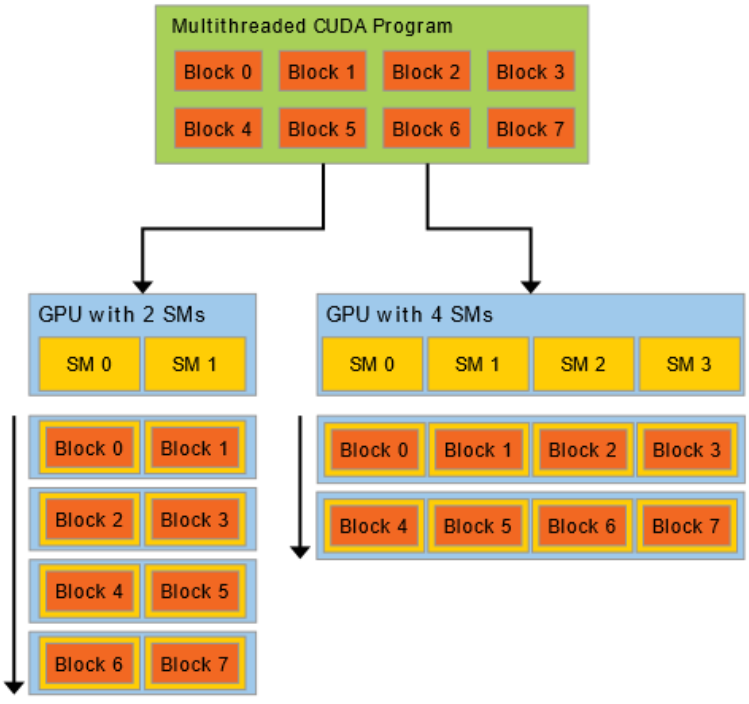

Figura 1-3. Escalado automático en función de la disponibilidad de recursos (Nvidia, 2016).

1.2 Arquitectura

1.2.1 Introducción

Antes de describir cómo es la arquitectura general, es preciso presentar términos asociados a la programación multihilo (*multithreading*) y adelantar qué supone lanzar un programa en CUDA para entender mejor detalles de la misma.

Cuando un programa CUDA lanza un *kernel* (función a ejecutar sobre la GPU) desde el host (donde se encuentra la CPU, el anfitrión), este se descompone en tantos bloques de hilos como el programador haya establecido. Los hilos se agrupan en bloques de hilos y los bloques en mallas. Al invocar una malla de *kernel*, los bloques de hilos que contiene se enumeran y distribuyen en los multiprocesadores disponibles. En cada multiprocesador se ejecutan bloques de hilos concurrentemente y del mismo modo los hilos de cada bloque, de forma que al terminar un bloque de hilos, un nuevo bloque ocupará su lugar y así hasta ejecutar la malla al completo.

1.2.2 Unidades de procesamiento

La arquitectura Nvida GPU consta de un conjunto de multiprocesadores o procesadores *streaming* (en inglés, Streaming Multiprocessors (SMs)) con capacidad para ejecutar cientos de hilos concurrentemente. Cada uno de los multiprocesadores tiene su banco de registros, una memoria compartida, una caché de texturas y acceso a la memoria global del dispositivo, como se aprecia en la figura 1.2.2[3].

Para hacer frente a esta gestión, se emplea una arquitectura llamada SIMT (Single-Instruction, Multiple-Thread) y se pretende que las instrucciones aprovechen el paralelismo de los hilos y así obtener un hardware multihilado, a diferencia de los núcleos de la CPU que operan de manera secuencial.

El multiprocesador no crea, gestiona, programa y ejecuta hilos de cualquier forma, sino en grupos de 32 hilos llamados *warps*. (Término que podría traducirse por urdimbre y que tiene su origen en las tejedurías, considerada la primera tecnología de hilos paralelos).

Cada hilo de un *warp* empieza en la misma dirección de programa pero al tener su propio contador de direcciones de instrucción y registro de estados, está libre de ejecutarse de forma independiente a los demás.

La forma en la que el multiprocesador gestiona los hilos siempre es la misma: en primer lugar recibe los bloques de hilos y los divide en *warps*, cada *warps* tiene entonces 32 hilos que son numerados del 0 al 31, asignándole así un identificador a cada hilo. A continuación, cada *warp* ejecuta una instrucción a la vez, obteniéndose la máxima eficiencia cuando todos los hilos del *warp* siguen el mismo camino de ejecución y disminuyendo la eficiencia cuando los hilos divergen por ramas condicionales. Esta divergencia de rama se da solo entre hilos de un mismo *warps*, ya que *warps* diferentes se ejecutan de forma independiente.

Se dice que los hilos de un *warp* están activos cuando están en el camino de ejecución actual del *warp* e inactivos cuando se han deshabilitado por haber tomado una rama diferente de la que se está ejecutando en el *warp* o porque son los últimos hilos de un bloque cuyo número de hilos no es múltiplo del tamaño del *warp*.

Como último detalle de los *warps*, y dado que se mantienen en el chip del procesador durante su tiempo de vida, el cambio de contexto no tiene coste.

Debido a las limitaciones en registros y memoria compartida disponible en el multiprocesador, el programador

[3] Figura tomada de Ujaldón, Manuel 2014. Programando la GPU con CUDA. Curso en el Dpto de Matemáticas e Informática.Junio 23-25, 2014

debe tener en cuenta el máximo número de bloques y de *warps* que pueden residir en un multiprocesador ya que estos recursos se repartirán entre ellos, siendo estos límites función de la generación del dispositivo. En el caso de que no hubiera recursos suficientes para procesar al menos un bloque, el *kernel* fallará al lanzarse.

Con SIMT el programador puede escribir código paralelo a nivel de hilo independientemente, escalar los hilos y trabajar con ellos de forma coordinada. Puede incluso obviar el comportamiento SIMT pero obtendría mejores rendimientos siendo consciente de él.

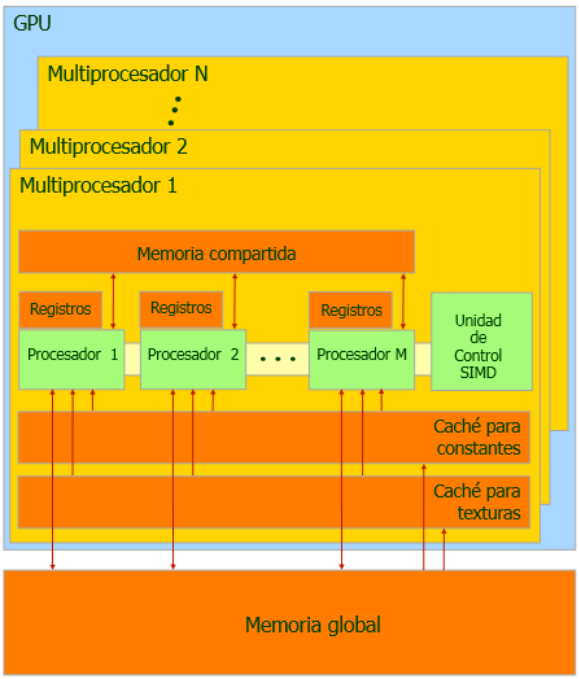

Figura 1-4. Modelo hardware de CUDA (Manuel Ujaldón, 2014).

1.2.3 Unidades de memoria

En cuanto a la memoria que se puede encontrar en un multiprocesador, se presenta con esta jerarquía:

- Memoria global: reside en la memoria del dispositivo y es la que recibe desde la memoria del host todos los datos que estén involucrados en el *kernel*. Al ser la memoria de mayor tamaño que encontramos en la GPU también es la de mayor latencia, por lo que hay que minimizar su uso desde los *kernels*. A ella pueden acceder todos los multiprocesadores.

- Memoria local: se trata de un almacenamiento privado para un hilo en ejecución y no es visible fuera de este hilo, cada hilo posee su propia memoria local. Se gestiona automáticamente cuando un hilo que se ejecuta dentro de un SM ha ocupado el número máximo de registros disponibles en ese SM.

- Memoria compartida: se encuentra en el multiprocesador y solo puede ser accedida por los núcleos del multiprocesador. Al estar en el chip tiene mucho mayor ancho de banda y menos latencia que la memoria global. La memoria compartida es usada por los bloques para compartir datos entre sus hilos y es gestionada explícitamente por el programador.

- Caché de constantes: se encuentra en cada multiprocesador y se usa para albergar datos que no cambian durante la ejecución del *kernel*. No puede utilizarse para dejar resultados, por lo que es de solo lectura.

- Caché de texturas: se encuentra en cada multiprocesador y es de solo lectura. Tiene un uso muy específico y marginal.

- Registros: es la memoria más cercana a los procesadores y con menor latencia que encontramos en una GPU. Cada SM tiene miles de registros que se reparten entre todos los hilos de ejecución del *kernel*.

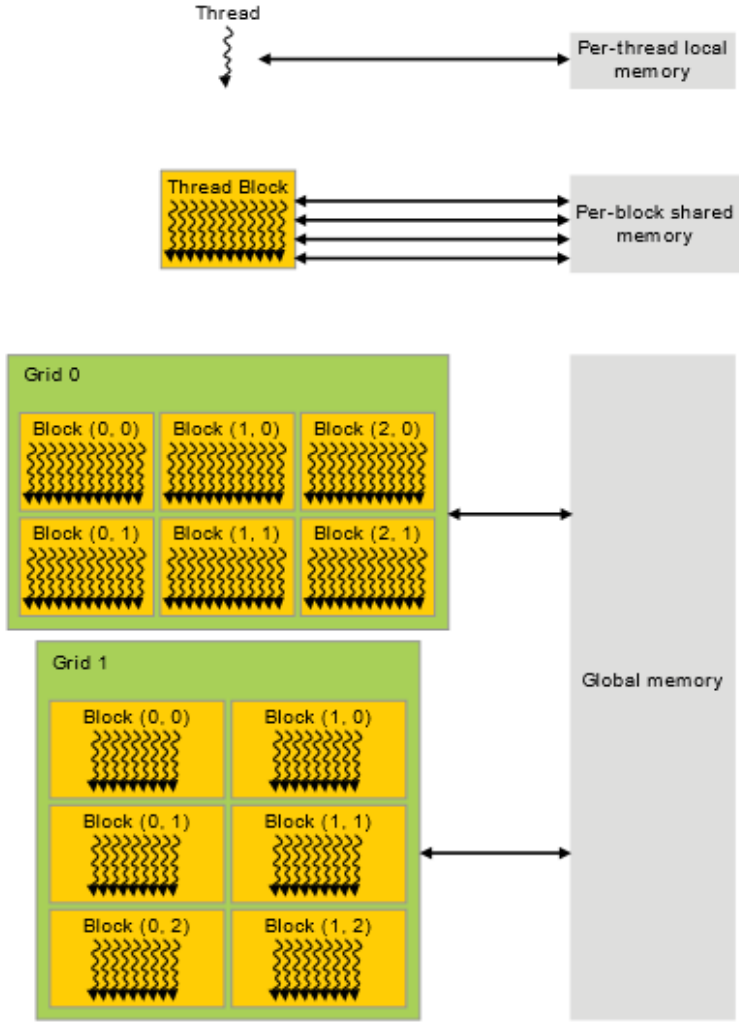

Figura 1-5. Jerarquía de memoria (Nvidia, 2016).

Según lo visto, cuando se lanza un kernel, los datos han debido ser previamente transferidos a la memoria global del dispositivo a través del puerto PCI-Express. Una vez allí los bloques que se ejecutan en distintos multiprocesadores pueden acceder a ellos y volcar el resultado a la memoria global para que el código ejecutado en la CPU prosiga. Dado que la memoria compartida tiene mayor ancho de banda que la memoria global, los hilos de cada bloque pueden pasar los datos que vayan a manejar a esta memoria y así contribuir a una mejora del rendimiento.

Se tiene entonces que los hilos de un mismo bloque utilizan los registros y pueden hacer uso de la memoria compartida para compartir datos entre sí y que para la comunicación entre bloques, ya sea en el mismo o en distintos multiprocesadores, se usa la memoria global, visible por todos los hilos.

Como se ve, en esta programación la gestión de la memoria es baste explícita y el programador puede tener un mayor control de estos recursos, principalmente el uso como caché de la memoria compartida.

1.2.4 Generaciones hardware de CUDA y versiones

Una vez que se ha descrito de forma general cómo los multiprocesadores gestionan los hilos y cómo se accede a los distintos recursos de memoria, se recoge en la siguiente tabla de forma resumida las distintas tecnologías de arquitecturas y la evolución que han vivido.

Tabla 1-1. Generaciones hardware de CUDA

Generación y año	Número de procesadores	Núcleos/procesador	Memoria compartida	Memoria global
Primera generación: Tesla (2008)	16	8	16 KB	1.5 GB
Segunda generación: Fermi (2010)	16	32	48 KB	6 GB
Tercera generación: Kepler (2012)	8	192	112 KB	4 GB
Cuarta generación: Maxwell (2014)	16	128	96 KB	8 GB
Quinta generación: Pascal (2016)	20	128	96 KB	12 GB

Origen: Nvidia, 2017. Disponible en: http://www.nvidia.es

Según se ve en la tabla, el crecimiento de las arquitecturas viene dado en tres niveles:

- Aumentar el número de multiprocesadores.
- Aumentar el número de núcleos por procesador.
- Aumentar el tamaño de la memoria compartida.

Para identificar las características soportadas por el hardware GPU, se usa el número de versión y en él se representa la capacidad de cálculo del dispositivo. Se denota por X.Y, donde X es el número de revisión mayor e indica la arquitectura de núcleos e Y es el número de revisión menor, que se corresponde con una mejora incremental de la arquitectura del núcleo.

1.3 Programación

1.3.1 Sobre los kernels

Como se adelantaba al principio de la sección anterior, un *kernel* es una función propia de CUDA definida por el programador que hace las veces de una función C convencional, pero a diferencia de ésta que se ejecutaría solo una vez, ahora al ser invocada se ejecuta N veces por N hilos CUDA diferentes.

Para provocar su lanzamiento es preciso seguir la sintaxis específica, declarando *__global__* seguido del nombre del *kernel* y de <<<M, N>>>, donde se establecerá el número de hilos por bloques (N) y el número de bloques que formarán la malla (M). Cada hilo que ejecuta un *kernel* recibe un *thread ID*, es decir, un identificador del hilo, que es accesible dentro del *kernel* a través de la variable *threadIDx* incorporada.

En el último apartado de este capítulo se expone un ejemplo comentado para señalar las diferencias principales con el lenguaje C tradicional y así aclarar aquellas ideas de programación que se introduzcan en esta sección de programación.

1.3.2 Jerarquía de hilos

La variable *threadIDx* es en general un vector de tres componentes que identifica al hilo usando un índice unidimensional, bidimensional o tridimensional ya que el hilo puede encontrarse en un bloque de hasta tres dimensiones y será conveniente poder invocar al hilo oportuno según la necesidad de nuestra aplicación, y así movernos en el contexto de vectores, matrices y volúmenes con facilidad.

A su vez, los bloques de una malla también pueden adoptar la misma configuración que los hilos dentro de un bloque, de manera que existe la variable *blockIDx* para situar el bloque dentro de la malla.

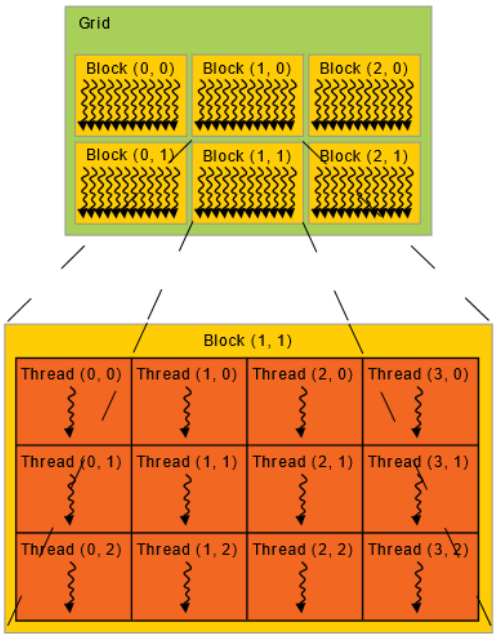

Figura 1-6. Organización de hilos y bloques (Nvidia, 2016).

Normalmente encontramos GPUs donde cada bloque puede contener más de 1024 hilos, existiendo un límite en el número de hilos por bloque. Esto se debe a que todos los hilos de un bloque esperan residir en el mismo núcleo procesador y deben compartir los limitados recursos de memoria del núcleo.

Para solucionar estas limitaciones, un *kernel* puede ser ejecutado por múltiples bloques de hilos de igual tamaño, resultando el número total de hilos igual al número de hilos por bloque multiplicado por el número de bloques. Así también se garantiza que haya el máximo número de multiprocesadores trabajando y que se aprovechen las posibilidades de cálculo y recursos que ofrezca una GPU dada.

Otra particularidad de la programación pero que se convierte en su principal ventaja a la hora de la portabilidad de un mismo código a otros dispositivos con diferentes arquitecturas, es que los bloques de hilos requieren ejecutarse de forma independiente y en cualquier orden, ya sea en serie o en paralelo. Este requerimiento de independencia es el que permite al bloque de hilos ser programado en cualquier dispositivo, cualquiera que sea su arquitectura, y dejar al programador un código escalable según el número de núcleos disponibles, como se veía en la figura 1-3.

Además de esta independencia a nivel de bloques, se encuentra un comportamiento opuesto dentro de ellos, y es que los hilos pueden ser programados para cooperar entre ellos compartiendo datos a través de la memoria compartida y sincronizar su ejecución para coordinar el acceso a ella. Esto se consigue especificando puntos de sincronización en el *kernel*. Llamando a la función *__syncthreads ()*, se establece una barrera en la que todos los hilos esperan antes de proseguir y así evitar errores como los relacionados con la lectura y escritura de datos en memoria.

1.4 Uso de memoria

Como ya se mencionó, el modelo de programación heterogénea de CUDA asume un sistema compuesto por un host y un dispositivo, cada uno con memorias separadas. Dado que los *kernels* operan fuera de la memoria del dispositivo, es en el tiempo de ejecución cuando se proporcionan funciones para reservar, liberar y copiar la memoria de dispositivo, así como transferir datos entre la memoria del host y del dispositivo.

La memoria del dispositivo puede ser reservada como memoria lineal usando *cudaMalloc (),* liberada usando *cudaFree ()* y las transferencias de datos entre las dos memorias se hacen típicamente usando *cudaMemcpy ().*

Puesto que los punteros son solo direcciones, no se puede conocer a través del valor de un puntero si la dirección pertenece al espacio de la CPU o de la GPU, por lo que habrá que evitar errores causados por acceder desde la CPU a los datos de la GPU y viceversa. Si bien es cierto que a partir de la versión 6 de CUDA se pueden unificar virtualmente dichos espacios de memoria para simplificar el proceso de intercambio de datos, siempre será más efectivo gestionar la memoria manualmente.

Los pasos a seguir para gestionar la memoria de forma adecuada serían:

- Reservar memoria en la CPU.
- Reservar memoria en la GPU.
- Almacenar datos en la memoria reservada en la CPU.
- Copiar los datos desde la CPU a la GPU.
- Copiar los resultados desde la GPU a la CPU.
- Liberar la memoria reservada en la GPU.

1.5 Ejecución concurrente asíncrona

Otra de las características de CUDA es disponer que las tareas independientes puedan operar concurrentemente. Entre estas tareas encontramos la computación en el host, la computación en el dispositivo, la transferencia de memoria entre host y dispositivo y la transferencia de memoria dentro de la memoria del dispositivo.

Este nivel de concurrencia dependerá como es obvio de la capacidad de cómputo del dispositivo.

Como se habrá deducido, la tarea más importante de las citadas es la ejecución concurrente entre host y dispositivo, lo que se consigue a través de funciones de librería asíncronas. Estas devuelven el control al hilo del host antes de que el dispositivo complete la tarea, por lo que el host queda libre para ejecutar otras.

Las siguientes operaciones del dispositivo son asíncronas con respecto al host:

- Lanzamiento de *kernels*.
- Copiar memoria dentro de una sola memoria de dispositivo.
- Copiar memoria del host al dispositivo de un bloque de memoria de 64 KB o menos.
- Copiar memoria usando funciones que son precedidas con *Async.*
- Conjunto de llamadas a funciones de memoria.

Conviene añadir que algunos dispositivos de capacidad de cálculo 2.x y los de capacidad de cálculo superior pueden ejecutar múltiples *kernels* concurrentemente, siendo el máximo número de *kernels* lanzados concurrentemente por un dispositivo dependiente de su capacidad de cálculo.

Asimismo también se pueden usar funciones asíncronas donde el control no se devuelve al hilo del host hasta que el dispositivo no haya completado la tarea.

1.6 Estrategias para optimizar el rendimiento

Con el fin de crear aplicaciones que tengan un rendimiento óptimo se deberían tener en cuenta las siguientes estrategias:

- Maximizar la ejecución en paralelo para lograr la máxima utilización.
- Optimizar el uso de memoria para lograr el máximo rendimiento de memoria.
- Optimizar el uso de instrucciones para lograr el máximo rendimiento de instrucciones.

1.6.1 Maximizar la utilización

Para maximizar la utilización, la aplicación debe estar estructurada de manera que exponga tanto paralelismo como sea posible y mapee eficientemente este paralelismo a varios componentes del sistema para mantenerlos ocupados la mayor parte del tiempo. En este sentido se propone que a alto nivel, la aplicación maximice la ejecución paralela entre el host y los dispositivos, usando para ello funciones asíncronas. Este reparto de tareas se asigna sabiendo el tipo de trabajo que mejor hace cada procesador: cargas de trabajo en serie para el host y en paralelo para los dispositivos.

En un nivel más bajo, de dispositivo, la aplicación debe maximizar la ejecución paralela entre los multiprocesadores de un dispositivo y así evitar que haya multiprocesadores sin utilizar, este caso se daría si se declara un número insuficiente de bloques.

Por último, a nivel de multiprocesador se debe maximizar la ejecución paralela entre varias unidades funcionales. Para ello se busca que siempre haya un número suficiente de *warps* listos para ejecutar las instrucciones que se emiten a cada ciclo de reloj y así ocultar los periodos de latencia -se define la latencia como el número de ciclos de reloj que tarda un *warp* en estar listo para ejecutar su siguiente instrucción-.

La razón más común de que un *warp* no esté listo para ejecutar la siguiente instrucción es que los datos de entrada de la instrucción no estén disponibles todavía o que algunos de estos datos sean escritos por instrucciones cuya ejecución no ha sido completada aún, estando entonces la latencia causada por la dependencia de los registros.

Otra razón para que el *warp* no esté listo para ejecutar su siguiente instrucción es que esté esperando en algún área de memoria o algún punto de sincronización. Un punto de sincronización puede forzar al multiprocesador a parar ya que más y más *warps* esperan a que otros warps del mismo bloque completen su ejecución de instrucciones anterior al punto de sincronización. Teniendo múltiples bloques residiendo por multiprocesador se puede ayudar a reducir la parada en este caso, ya que *warps* de diferentes bloques no necesitan esperar a los otros en los puntos de sincronización.

El número de bloques y *warps* que residen en cada multiprocesador para una llamada *kernel* dada depende de la configuración de ejecución de la llamada, los recursos de memoria del multiprocesador y de los requerimientos de recursos del *kernel*. A su vez, el número de hilos por bloque debe ser elegido como un múltiplo del tamaño del *warp* para evitar desperdiciar recursos de computación con *warps* infrautilizados tanto como sea posible.

Los efectos de la configuración de la ejecución en el rendimiento de una llamada *kernel* dada dependen generalmente del código del *kernel*, por lo que se recomienda la experimentación.

1.6.2 Maximizar el rendimiento de la memoria

El primer paso para maximizar en general el rendimiento de la memoria es minimizar la transferencia de datos con bajo ancho de banda. Esto ocurre principalmente en la trasferencia de datos entre el host y el dispositivo, mitigándose si se hace un uso eficiente del bus que los conecta.

De igual forma, el rendimiento disminuirá si se accede siempre a la memoria global y no se aprovecha la ventaja de que al pasar los datos a la memoria compartida se gana ancho de banda.

Como consecuencia a esto último, un patrón de programación típico es almacenar datos que vienen de la memoria del dispositivo a la memoria compartida; en otras palabras, tener cada hilo de un bloque:

- Cargando datos desde la memoria del dispositivo a la memoria compartida
- Sincronizándose con todos los otros hilos del bloque para que cada hilo pueda leer posiciones de memoria de forma segura.
- Procesando los datos en la memoria compartida.
- Sincronizando de nuevo si es necesario para asegurar que la memoria compartida ha sido actualizada con los resultados
- Escribiendo el resultado de vuelta a la memoria de dispositivo.

1.6.3 Maximizar el rendimiento de instrucciones

Para maximizar el rendimiento de instrucciones la aplicación debe:

- Minimizar el uso de instrucciones aritméticas con bajo rendimiento; esto incluye incrementar la velocidad cuando no se afecte al resultado final, por ejemplo usando funciones nativas propias de CUDA o precisión simple en lugar de doble precisión.
- Minimizar los *warps* divergentes causados por instrucciones condicionales.
- Reducir el número de instrucciones, por ejemplo, optimizando los puntos de sincronización siempre que sea posible.

1.7 Ejemplo de programación

Una vez el lector conoce las características principales de CUDA presentadas en los apartados anteriores, conviene ya concretar en algún ejemplo no muy complejo a fin señalar tanto las diferencias con la programación C tradicional como exponer algunas particularidades de CUDA.

Ejemplo 1–1

En este ejemplo se compara mediante la suma de dos vectores cómo se escribiría el código para ejecutarlo en la CPU (Ejemplo 1-1) y cómo se escribiría si queremos ayudarnos de la GPU para acelerar los cálculos (Ejemplo 1-2).

Tal y como se haría en código C, en el *main* se declararía tres vectores que corresponderían a los sumandos y al resultado y se reserva memoria para contener SIZE datos de tipo entero.

A continuación se recorrería con un bucle las componentes de los vectores para inicializarlos y se llamaría a la función que realiza la correspondiente suma realizando la operación con un bucle que en cada iteración calcula una componente del vector resultado.

Por último se imprimirían las primeras componentes del resultado por comprobar la validez del código y se liberaría la memoria utilizada.

En CUDA, se habría que realizar las siguientes modificaciones:

En primer lugar, en *main* se empieza declarando las variables por pares y es que tal y como se expuso, hay que mantener variables en la memoria del host y variables en la memoria del dispositivo, no pudiendo accederse a espacios de memoria que no le corresponde. Típicamente a las variables del dispositivo se las nombra como a las variables del host pero antecediéndolas con 'd_' de dispositivo.

A la hora de reservar memoria, se procede con las variables del host como en el código anterior y es a la hora de reservar memoria para las variables del dispositivo cuando se hace uso de la función nativa *cudaMalloc ()*, cuyos parámetros son la variable a alojar y su tamaño.

Antes de pasar los datos desde la memoria del host a la memoria del dispositivo para que pueda trabajar con ellos accediendo al espacio de memoria ya reservado, es preciso inicializar las variables de la misma forma que en el código.

Una vez hecho esto, se puede hacer tal copia utilizando la función *cudaMemcpy ()*, que recibirá como argumentos el objetivo donde se copia, el origen de la misma, el tamaño y se especifica el sentido de la copia, bien *cudaMemcpyHostToDevice* para pasar los datos del host al dispositivo o bien *cudaMemcpyDeviceToHost* si se pretende lo contrario.

En estas condiciones, con los datos de entrada listos en el dispositivo, ya se está en disposición de hacer la llamada al *kernel*, para lo cual se escribe el nombre del *kernel*, se usan los elementos <<<n, m>>>, necesarios en la sintaxis ya que el primer valor determina el número de bloques y el segundo el número de hilos por bloque y por último se pasan los parámetros necesarios para realizar la suma. Por simplicidad, el programador decide que la suma se lleve a cabo en un único bloque con tantos hilos como componentes tiene los vectores involucrados, de manera que cada hilo calcule una componente y se logre así un paralelismo de grano fino (cada hilo un dato).

Como se veía en el subsección de la ejecución asíncrona, tras la llamada al *kernel* el control se devuelve al host para que pueda seguir realizando tareas mientras la GPU está ocupada, si bien es cierto que en este ejemplo sólo le queda esperar a que el vector resultado tenga el contenido de la suma para copiar el vector al espacio de memoria del host y así seguir operando con él.

El código termina de la misma forma que el anterior, imprimiendo por pantalla para verificar la validez del resultado y liberando los espacios de memoria asociados, labor que también se lleva a cabo en el dispositivo utilizando la función *cudaFree ()*.

Queda solo por comentar el fragmento de código relacionado con el *kernel* propiamente, es decir, con la función que se ejecuta en los multiprocesadores. Para declarar un *kernel*, se utiliza *__global__* antes de la función para indicar que se ejecuta en el dispositivo y que se llama desde el código del host. Asimismo se nombra la función y los parámetros que espera recibir.

Los cambios más notables en la forma de realizar el cometido de la función vienen dados por el paralelismo que se pretende, por lo que ahora cada hilo del bloque ejecutará el código de la función. La forma de relacionar cada hilo con la posición que ocupa cada componente del vector es identificando al hilo utilizando la variable *threadIDx.x*, la cual en este contexto de bloque unidireccional de tamaño SIZE, otorga a cada hilo su índice dentro del bloque y así indexa al vector para que al final cada componente del vector resultado haya sido calculado por la suma de cada componente con el mismo índice.

Por ilustrar este concepto, esta tabla pretende aclarar con un ejemplo más breve:

Suma <<<1, 3>>>

Tabla 1-2. Ejemplo

Un único bloque con tres hilos:	**Hilo 0**	**Hilo 1**	**Hilo 2**
Leerán las componentes:	a [0] b [0]	a [1] b [1]	a [2] b [2]
Para dar:	c [0]	c [1]	c [2]

De forma que cada hilo solo ha tenido que mirar su *threadIDx.x* para que el vector resultado tuviera su componente calculada.

Para terminar este ejemplo, queda señalar el detalle de la sentencia condicional. Su existencia se justifica con que no siempre el número de componentes del vector o en general del objeto de nuestros cálculos es múltiplo del número de hilos, por lo que los hilos que tengan un identificador que esté fuera de la dimensión quedarán desactivados.

Este es un ejemplo didáctico, en la práctica las aplicaciones se realizan declarando cientos de bloques e hilos.

Ejemplo 1-1. *Código escrito en C*

```
#include <stdio.h>

#define SIZE 1024

void VectorAdd(int *a, int *b, int *c, int n)
{
  int i;

  for (i = 0; i < n; ++i)
    c[i] = a[i] + b[i];
}

int main()
{
  int *a, *b, *c;

  a = (int *)malloc(SIZE * sizeof(int));
  b = (int *)malloc(SIZE * sizeof(int));
  c = (int *)malloc(SIZE * sizeof(int));

  for (int i = 0; i < SIZE; ++i)
  {
    a[i] = i;
    b[i] = i;
    c[i] = 0;
  }
```

```
  VectorAdd(a, b, c, SIZE);

  for (int i = 0; i < 10; ++i)
  printf("c[%d] = %d\n", i,c[i]);

  free(a);
  free(b);
  free(c);

  return 0;

}
```

Ejemplo 1-2. *Código escrito en CUDA*

```
#include <stdio.h>

#define SIZE 1024

__global__ void VectorAdd(int *a, int *b, int *c, int n)
{
  int i = threadIDx.x;

  if (i < n)
  c[i] = a[i] + b[i];
}

int main()
{
  int *a, *b, *c;
  int *d_a, *d_b, *d_c;

  a = (int *)malloc(SIZE * sizeof(int));
  b = (int *)malloc(SIZE * sizeof(int));
  c = (int *)malloc(SIZE * sizeof(int));

  cudaMalloc(&d_a, SIZE*sizeof(int));
  cudaMalloc(&d_b, SIZE*sizeof(int));
  cudaMalloc(&d_c, SIZE*sizeof(int));

  for (int i = 0; i < SIZE; ++i)
  {
    a[i] = i;
    b[i] = i;
    c[i] = 0;
  }

  cudaMemcpy(d_a, a, SIZE*sizeof(int),cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, SIZE*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_c, c, SIZE*sizeof(int), cudaMemcpyHostToDevice);
  VectorAdd<<< 1, SIZE >>>(d_a, d_b, d_c, SIZE);
```

```
  cudaMemcpy(c, d_c, SIZE*sizeof(int), cudaMemcpyDeviceToHost);


  for (int i = 0; i < 10; ++i)
    printf("c[%d] = %d\n", i, c[i]);

  free(a);
  free(b);
  free(c);

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);

  return 0;
}
```

2 PROGRAMAR GPU USANDO MATLAB

2.1 Introducción

Según se ha presentado en el capítulo anterior, es posible desarrollar aplicaciones en un dispositivo cuya arquitectura tiene múltiples núcleos procesadores como es la GPU utilizando CUDA. De este modo se puede poner en marcha un modelo de programación heterogénea en la que la CPU realiza una serie de tareas y otras más relacionadas con el procesamiento masivo de datos tienen lugar en la GPU.

Se ha visto que para un buen desarrollo de esta tecnología y obtener los beneficios que ofrece, son necesarios unos conocimientos acerca de CUDA. Efectivamente, programar en código C para una GPU requiere cambiar el modelo mental y desarrollar unas habilidades que pueden ser difíciles y/o llevar tiempo en adquirir. Esto provoca que posibles usuarios motivados en acelerar sus aplicaciones de ingeniería o de análisis financiero por ejemplo, no puedan acceder a las ventajas aportadas por las GPUs.

Asimismo, los programadores que llevan más tiempo utilizando CUDA saben que aunque los algoritmos escritos para la GPU son mucho más rápidos, el proceso de desarrollar y comprobar estos algoritmos es costoso en cuanto a tiempo se refiere. Esto es así porque para verificar el diseño de un *kernel* es preciso escribir bastante código (ver ejemplo del capítulo 1) que tiene que ver con transferir los datos a la memoria de la GPU, gestionar esa memoria, inicializar y lanzar los *kernels*… En definitiva, se vuelve engorroso y es difícil de modificar el código a la hora de evaluar los resultados de un *kernel* en el que se están llevando a cabo diferentes pruebas para decidir sobre su rendimiento o eficacia.

Por todo ello, desde la versión R2007a, MATLAB ofrece el Parallel Computing Toolbox, que permite ejecutar código escrito en MATLAB en una GPU realizando solo algunos cambios en el código.

Dado que MATLAB es un lenguaje de alto nivel, ya no hace falta conocer cómo funciona una GPU con el mismo detalle que se requería usando CUDA, pudiendo por tanto explotar sus ventajas con mayor facilidad. A cambio, se pierde control sobre el dispositivo y el manejo de sus recursos, con lo que la eficiencia de la aplicación no será máxima.

Un punto a tener siempre en cuenta programando con CUDA, y por extensión ahora que se pretenden desarrollar algoritmos con MATLAB, es preguntarse si realmente la GPU va a acelerar una aplicación dada. Es decir, para saber si una GPU podrá cumplir con aquello que se espera de ella y así justificar su utilización, se deberá cumplir los dos criterios siguientes:

- La aplicación debe ser computacionalmente intensiva, lo que quiere decir que el tiempo empleado en su ejecución exceda de forma significativa el tiempo que se necesita en trasferir datos entre CPU y GPU.
- La aplicación debe ser susceptible de llevarse a cabo en paralelo y poder descomponer el problema en cientos de unidades de trabajos independientes, los hilos.

En el caso de que la aplicación no satisfaga estos criterios se daría el caso de que la CPU ejecute la aplicación más rápido que la GPU.

2.2 Programar una GPU usando MATLAB

El paquete de computación paralela incorporado por MATLAB facilita las tareas de transferir los datos a la memoria de la GPU, recuperar los resultados pasándolos al espacio de trabajo de MATLAB, es decir en memoria principal de la CPU, y la gestión de la memoria del dispositivo entre otras.

A diferencia del código necesario para tales tareas en CUDA, ahora se ejecutan comandos más simples y las funciones utilizadas en MATLAB existentes en otros paquetes como los de optimización o procesamiento de imágenes pueden ejecutarse directamente en la GPU; tan solo hay que trabajar con variables del tipo GPUArray proporcionada por el Parallel Computing Toolbox para alcanzar esta compatibilidad.

Tal y como se procedía en CUDA, el primer paso para trabajar con las variables en el dispositivo es escribir dichas variables desde el espacio de trabajo de MATLAB a la memoria del dispositivo. Esto se consigue con el comando *gpuArray ().*

Mientras los datos sean del tipo *gpuArray*, se pueden realizar los cálculos directamente en la GPU sin más que llamar a las funciones o realizar operaciones como se hace normalmente con MATLAB y así de sacar provecho del multiprocesador.

Los resultados de dichas operaciones seguirán siendo variables del tipo *gpuArray* y del mismo modo que en CUDA, una vez que se quiera recuperar ese resultado para seguir trabajando con él desde el espacio de trabajo de MATLAB se llamará a la función *gather (),* que registrará la variable en doble precisión.

Vista la facilidad a la hora de hacer uso de la GPU para acelerar los cálculos que los precisen, se debe tener en cuenta que abusar de la transferencia de datos entre CPU y GPU provoca que el rendimiento general de la aplicación se degrade, por lo que es más eficiente realizar todas las operaciones posibles en los datos aprovechando que están en la GPU y solo traer de vuelta los datos a la CPU cuando sea necesario. En este sentido, es una buena práctica declarar directamente las variables siempre que se pueda en la GPU, evitando así tener que hacer la transferencia y ahorrando el tiempo dedicado a ello.

Otro punto a tener presente es que ambos dispositivos tienen espacios de memoria finitos, pero la GPU a diferencia de la CPU no puede llevar datos de la memoria principal al disco duro y viceversa, por lo que se deberá comprobar que los datos que se pasan a la GPU no excedan sus límites.

Ejemplo 2–1.

```
x = 1:1000;       % Se inicializa un vector
d_x = gpuArray(x); % Se transfiere a la GPU
                   % donde se almacenará como variable
                   % de tipo gpuArray
d_y = d_x.^2;     % Este cálculo tiene lugar en la GPU
                  % sin indicaciones extra al ser la variable
                  % de entrada de tipo gpuArray
plot(d_x,d_y)     % Función ejecutada por la GPU, aprovechando
                  % que se tienen los datos aquí
y = gather(d_y); % Se recupera el resultado convirtiéndolo
                  % a tipo doble por defecto.
```

Como ha quedado reflejado en este ejemplo, se puede hacer uso de la GPU con un esfuerzo mínimo y sin apenas conocimiento de GPUs, por lo que a diferencia del aprendizaje que suponía programar con CUDA, MATLAB facilita esta tarea a cambio de no disponer de un control tan profundo sobre la GPU. Este control suponía gestionar la memoria del dispositivo a voluntad, hacer uso de la memoria compartida para la comunicación de hilos, hacer cambios en el código para mejorar el rendimiento... Como contrapartida, se gana simplicidad y disminuye la probabilidad de que existan los errores asociados a estas acciones.

Además de todo lo anterior, para los programadores que estén más familiarizados con la programación de GPUs, MATLAB ofrece integrar los *kernels* programados en CUDA en las aplicaciones hechas con MATLAB sin tener que escribir código en C adicional.

Las aplicaciones programadas de este modo son mejores al tenerse ahora un mayor control sobre las partes del código que requieren una intensidad de cálculos superior, ya que no solo se ejecutarán en la GPU sino que se tiene un control del paralelismo y del cómo se están procesando esos datos gracias a la programación de *kernels*.

Como consecuencia, se consigue un equilibrio entre la eficiencia perdida al delegar toda la ejecución en MATLAB y la facilidad de su lenguaje.

Para los programadores que lo deseen, MATLAB habilita la interfaz CUDAKernel del Parallel Computing Toolbox, la cual crea un objeto de tipo *kernel* que habrá que configurar antes de su utilización para establecer el número de hilos y los bloques necesarios.

La manera en la que MATLAB relaciona el *kernel* escrito en CUDA y el objeto creado es mediante la compilación del *kernel* en un código PTX.- un conjunto de instrucciones para la ejecución paralela de hilos a bajo nivel.-

En general, una aplicación escrita en MATLAB que implemente porciones de código en CUDA se llevaría a cabo siguiendo los pasos siguientes:

- En primer lugar, y usando la opción Run and Time de MATLAB, se localizan las partes de código que tardan más en ejecutarse y que por tanto son susceptibles de ser implementadas en un *kernel*. Los mejores candidatos serán aquellos que claramente pueden ser paralelizados masivamente y puedan ser acelerados usando la GPU.

- A continuación se implementa en CUDA el código seleccionado, escribiendo solo la parte concerniente al kernel: empezando por *__global__,* seguido del nombre del *kernel*, sus argumentos y el cuerpo del código.

- Una vez guardado el archivo, es preciso compilarlo a PTX utilizando el compilador de NVIDIA, donde se escribe el comando: nvcc –ptx nombre_del_archivo.cu

- Ahora que se tiene el *kernel* compilado, está en disposición de ser usado para sustituir el código homólogo en MATLAB y para poder trabajar con él, habrá primero que crear el objeto *kernel* del siguiente modo:
  kernel = parallel.gpu.CUDAKernel (‘nombre_del_archivo.ptx’, ’nombre_del_archivo.cu’).

- Una vez que el *kernel* se ha cargado, se debe configurar el tamaño, número de bloques e inicializar la variable que recogerá el resultado antes de poder utilizarlo como si fuera una función cualquiera de MATLAB. Esto último gracias al comando *feval*, que evalúa los datos de entrada según el *kernel*.

Se tiene entonces un código de MATLAB más eficiente dado que ejecutará las tareas correspondientes con la CPU y para las tareas más costosas intervendrán los *kernels* programados en CUDA, garantizando así que la colaboración CPU-GPU lleve a nuestra aplicación al rendimiento deseado.

Para practicar los detalles de esta implementación se propone el siguiente ejemplo.

Ejemplo 2-3. *Este sería el código en CUDA para su utilización como kernel desde MATLAB*

```
__global__ void VectorAdd(int *a, int *b, int *c, int n)
{
    int i = threadIdx.x;

    if (i < n)
    c[i] = a[i] + b[i];

    return;
}
```

Ejemplo 2-4. *Este sería el código en MATLAB incluyendo el kernel en CUDA*

```
Size = 1024;

% Inicializamos los vectores
a = zeros(Size,1);
b = zeros(Size,1);

for i=1:Size
    a(i) = i;
    b(i) = i;
end

% Cargamos el kernel
kernel = parallel.gpu.CUDAKernel('prueba.ptx','prueba.cu');

% Configuramos los hilos
[nRows,nCols,~] = size(a);
blockSize = nRows;
kernel.ThreadBlockSize = [blockSize,1,1];
kernel.GridSize = [ceil(nRows/blockSize),ncols];

% Copiamos los datos a la GPU
d_a = gpuArray(a);
d_b = gpuArray(b);

% Reservamos e inicializamos el vector resultado, directamente en la GPU
d_c = gpuArray(zeros(size(a)));

% Aplicamos el kernel
d_c = feval(kernel,d_a,d_b,d_c,Size);

% Copiamos el resultado desde la GPU a la memoria principal
c = gather(d_c);

for i=0:10
     c(i)
end
```

Ejemplo 2-5. *Este sería todo el código en CUDA equivalente a la implementación anterior en MATLAB*

```
#include <stdio.h>

#define SIZE 1024
__global__ void VectorAdd(int *a, int *b, int *c, int n)
{
   int i = threadIDx.x;

   if (i < n)
   c[i] = a[i] + b[i];

   return;
}

int main()
{
   int *a, *b, *c;
   int *d_a, *d_b, *d_c;

   a = (int *)malloc(SIZE * sizeof(int));
   b = (int *)malloc(SIZE * sizeof(int));
   c = (int *)malloc(SIZE * sizeof(int));

   cudaMalloc(&d_a, SIZE*sizeof(int));
   cudaMalloc(&d_b, SIZE*sizeof(int));
   cudaMalloc(&d_c, SIZE*sizeof(int));

   for (int i = 0; i < SIZE; ++i)
   {

     a[i] = i;
     b[i] = i;
     c[i] = 0;
   }

cudaMemcpy(d_a, a, SIZE*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, SIZE*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_c, c, SIZE*sizeof(int), cudaMemcpyHostToDevice);

VectorAdd<<< 1, SIZE >>>(d_a, d_b, d_c, SIZE);

cudaMemcpy(c, d_c, SIZE*sizeof(int), cudaMemcpyDeviceToHost);

for (int i = 0; i < 10; ++i)
printf("c[%d] = %d\n", i, c[i]);

free(a);
free(b);
free(c);

cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

return 0;}
```

Se comprueba entonces que es relativamente sencillo llamar a los *kernels* CUDA desde MATLAB, además de realizar todas las tareas asociadas (inicializar datos de retorno, transferir datos a la GPU, lanzar el *kernel*…) con tal solo una línea de código MATLAB, a diferencia de todo el código que se requiere para desarrollar la misma aplicación solo desde CUDA.

Otra ventaja de utilizar la combinación de *kernels* y MATLAB es que se consigue así que el código esté más depurado, pudiendo localizar más fácilmente los errores y que se gana en flexibilidad, ya que se puede evaluar el *kernel* sin tener que retocarlo, tan solo cambiando los parámetros de entrada desde MATAB. Al mismo tiempo, el código resultante es más propenso a que otros desarrolladores lo comprendan o incluso adapten según sus necesidades. (La misma aplicación en CUDA requiere más código, es más difícil de entender y por tanto invita menos a su modificación).

3 LA METODOLOGÍA MPC

3.1 Introducción

El control predictivo basado en modelo (MPC por sus siglas en inglés) es una estrategia de control surgida en los años 70 y que se enmarca dentro de la teoría de control moderna. En sus comienzos se asentó en la industria petroquímica, siendo posteriormente implementada en otras disciplinas como el control de brazos robóticos, plantas solares, columnas de destilación… Entre sus puntos fuertes cabe citar la robustez del método, la facilidad para trabajar con sistemas multivariables y su capacidad para incorporar restricciones y retrasos. Como contrapartida, el coste computacional asociado a resolver un problema de optimización a cada instante lastró su desarrollo, aplicándose fundamentalmente a procesos con dinámicas lentas e intervalos de muestreo amplios, hasta que los avances tecnológicos de las últimas décadas posibilitaron la aceleración de dichos cálculos gracias a las mejoras en el hardware y de los algoritmos de optimización.

En esencia, el control predictivo se basa en disponer de un modelo de la planta o sistema a controlar, de manera que se puedan realizar predicciones de los estados futuros de la misma y así establecer la acción de control óptima en cada instante tras minimizar una función de coste. A partir de este concepto, se han desarrollado distintas metodologías, estando la mayoría de ellas dedicadas a modelos de predicción lineales, aunque en los últimos años se están desarrollando modelos no lineales como los de A. Grancharova y T.A. Johansen (2012) y L. Grüne y J. Panneck (2011) para resolver y abarcar un mayor número de problemas de control.

3.2 Estrategia de control

Como en cualquier controlador, el objetivo es calcular la acción de control adecuada para llevar el sistema a la referencia deseada. En MPC este propósito se concreta en la obtención de una trayectoria futura de la variable manipulada u.

Para ello se dispone de un horizonte de predicción N, en el que se harán las predicciones de las variables en cada instante k_i. Para confiar en la bondad de estas predicciones y tal como sugiere el nombre de este método, el disponer de un modelo que recoja adecuadamente las dinámicas y el comportamiento del sistema en cuestión es fundamental. Para realizar las predicciones se cuenta con la información de las variables en el instante k_i, como pueden ser el estado del sistema y la secuencia de salidas hasta el instante k_i (ver figura 3-1). Con estos datos se establece una función de coste que valore los datos disponibles con unos pesos y se minimiza la misma para encontrar las señales de control futuras. Típicamente la función de coste suele ser cuadrática y penaliza las desviaciones entre la salida predicha y la referencia, así como incluir el esfuerzo de la acción de control.

En ausencia de restricciones y dado un sistema lineal, se puede encontrar una solución analítica y, en caso contrario, será necesario resolver un algoritmo numérico para su optimización.

Dado que en la mayoría de los casos se eligen funciones de coste cuadráticas y se tienen restricciones lineales, dicha solución pasará por un algoritmo de programación cuadrática (Quadratic Programming, QP por sus siglas en inglés).

Sea como fuere, una vez se tiene el vector con las entradas futuras de la planta, se toma la primera componente, desechando el resto, y una vez llevado el sistema al siguiente tiempo de muestreo k+1, se vuelve a repetir el proceso, ya que en el siguiente instante se dispondrá de más información para hacer nuevas predicciones que diferirán en general de las anteriores.

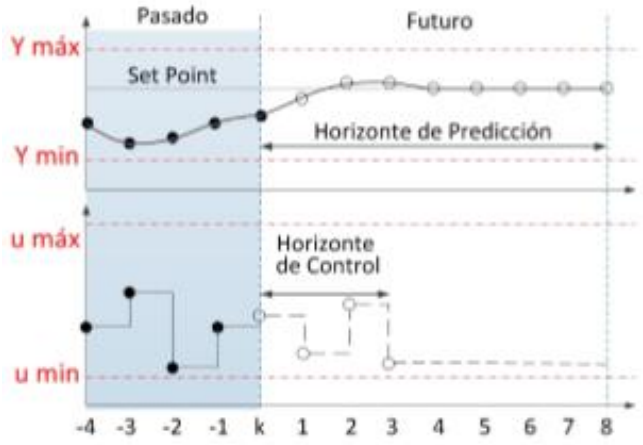

Figura 3-1. Esquema de control predictivo (Zambrano J. y González A., 2013)[4].

3.3 Formulación del problema en el espacio de estados

Una vez presentado el modo de operar, se deberá contar con un modelo del sistema en el espacio de estados para hacer las predicciones. Como los estados de la planta son indispensables, en ocasiones serán necesarios observadores de estados para disponer de los mismos. El modelo aquí utilizado será de la forma siguiente, aunque en la literatura se pueden encontrar otras variantes de esta formulación:

$$x(k+1) = Ax(k) + Bu(k) + Dw(k) \tag{3-1}$$

Donde: $x(k)$ es el estado en el instante k

$u(k)$ es la entrada en el instante k

$w(k)$ es la perturbación en el instante k

$A, B\ y\ D$ son las matrices del modelo

A fin de concretar una notación con la que trabajar, se establece que:

$$X = \begin{bmatrix} x(1) \\ x(2) \\ \vdots \\ x(N) \end{bmatrix} \in \mathbb{R}^{N \cdot n_x \times 1},\ U = \begin{bmatrix} u(0) \\ u(1) \\ \vdots \\ u(N-1) \end{bmatrix} \in \mathbb{R}^{N \cdot n_u \times 1},\ W = \begin{bmatrix} w(0) \\ w(1) \\ \vdots \\ w(N-1) \end{bmatrix} \in \mathbb{R}^{N \cdot n_w \times 1}$$

Escribiendo el estado siguiente a partir de los datos en el estado actual, se puede construir la siguiente ecuación matricial que recogerá los estados hasta el horizonte N elegido:

$$X = \begin{bmatrix} x(1) \\ x(2) \\ x(3) \\ \vdots \\ x(N) \end{bmatrix} = \begin{bmatrix} A \\ A^2 \\ A^3 \\ \vdots \\ A^N \end{bmatrix} x(0) + \begin{bmatrix} B & 0 & 0 & 0 & 0 \\ AB & B & 0 & 0 & 0 \\ A^2B & AB & B & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ A^{N-1}B & A^{N-2}B & \cdots & AB & B \end{bmatrix} \begin{bmatrix} u(0) \\ u(1) \\ u(2) \\ \vdots \\ u(N-1) \end{bmatrix} + \begin{bmatrix} D & 0 & 0 & 0 & 0 \\ AD & D & 0 & 0 & 0 \\ A^2D & AD & D & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ A^{N-1}D & A^{N-2}D & \cdots & AD & D \end{bmatrix} \begin{bmatrix} w(0) \\ w(1) \\ w(2) \\ \vdots \\ w(N-1) \end{bmatrix}$$

[4] Imagen tomada del trabajo "Implementación de un algoritmo de control predictivo en espacio de estados sobre una plataforma de simulación desarrollada en MATLAB" Zambrano J. y González A., 2013

Por lo que se puede escribir que el estado viene dado por

$$X = G_x x(0) + G_u U + G_w W \tag{3-2}$$

Con $G_x \in \mathbb{R}^{N \cdot n_x \times nx}$, $G_u \in \mathbb{R}^{N \cdot n_x \times N \cdot n_u}$, $G_w \in \mathbb{R}^{N \cdot n_x \times N \cdot n_w}$

Con el modelo bien definido, es preciso fijar un controlador que será el que proporcione la acción de control al minimizar una función de coste que penalizará las desviaciones del estado futuro del sistema, matriz Q, y el esfuerzo de la acción de control, matriz R.

$$V = \sum_{k=0}^{N-1} (x(k+1)^T Q\ x(k+1) + u(k)^T R\ u(k)) \tag{3-3}$$

Hay que tener en cuenta que la función de coste anterior intentará llevar el estado a 0, por lo que un cambio de variables aquí servirá para establecer otra referencia.

Tras desarrollar la función de coste, se tiene

$$V = X^T \begin{bmatrix} Q & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & Q & 0 \\ 0 & 0 & 0 & Q \end{bmatrix} X + U^T \begin{bmatrix} R & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & R & 0 \\ 0 & 0 & 0 & R \end{bmatrix} U \tag{3-4}$$

Ahora se puede desarrollar la expresión de V, teniendo en cuenta el modelo del sistema y que los términos que no dependen de U no participan en la minimización y se pueden eliminar, se llega a:

$$\frac{1}{2}\ U^T H\ U + F\ U \tag{3-5}$$

La cual puede ser resuelta eficientemente con esta expresión analítica, ya que el problema es lineal y no tiene restricciones en las variables

$$U = -\frac{1}{2} H^{-1} F \tag{3-6}$$

Y cerrar el problema para el instante k_i, repitiendo los pasos anteriores para el siguiente instante.

Uno de los atractivos de la estrategia MPC es la facilidad para incluir restricciones en las trayectorias de las variables, confinadas en unas tolerancias, limitaciones en la sobreoscilación o en la señal de control en forma de desigualdades matriciales e incluirlas en el problema de optimización.

En el caso de querer por ejemplo satisfacer restricciones en el estado o en las entradas, se puede escribir

$$x_{mín} < x < x_{máx} \quad \text{y} \quad u_{mín} < u < u_{máx} \tag{3-7}$$

Expresándolas en forma de desigualdades matriciales queda

$$A_x x(k) \leq b_x \tag{3-8}$$

$$A_u u(k) \leq b_u \tag{3-9}$$

Así escritas, hay que desarrollarlas un poco para obtener

$$\begin{bmatrix} A_x & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & A_x & 0 \\ 0 & 0 & 0 & A_x \end{bmatrix} X \leq \begin{bmatrix} b_x \\ \vdots \\ b_x \\ b_x \end{bmatrix} \rightarrow \hat{\mathrm{A}}_x\, X \leq \hat{b}_x \rightarrow\ \hat{\mathrm{A}}_x\, (G_x x(0) + G_u U + G_w W) \leq \hat{b}_x$$

$$\hat{\mathrm{A}}_x\, G_u U \leq \hat{b}_x - \hat{\mathrm{A}}_x G_x x(0) - \hat{\mathrm{A}}_{\mathrm{x}} \mathrm{G}_{\mathrm{w}} \mathrm{W} \quad \forall\, \mathrm{k} \in [1, \mathrm{N}] \tag{3-10}$$

$$\begin{bmatrix} A_u & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & A_u & 0 \\ 0 & 0 & 0 & A_u \end{bmatrix} U \leq \begin{bmatrix} b_u \\ \vdots \\ b_u \\ b_u \end{bmatrix} \rightarrow \hat{\mathrm{A}}_x\, U \leq \hat{b}_u \quad \forall\, \mathrm{k} \in [0, \mathrm{N} - 1] \tag{3-11}$$

Y así integrando la (3-10) y la (3-11) en una única desigualdad que agrupe todas las restricciones como restricción en la U, que es la variable de control y aquella que minimizamos en la función de coste.

$$\begin{bmatrix} \hat{\mathrm{A}}_x G_u \\ \hat{\mathrm{A}}_u \end{bmatrix} U \leq \begin{bmatrix} \hat{b}_x - \hat{\mathrm{A}}_x G_x x(0) - \hat{\mathrm{A}}_{\mathrm{x}} \mathrm{G}_{\mathrm{w}} \mathrm{W} \\ \hat{b}_u \end{bmatrix} \rightarrow \hat{\mathrm{A}} U \leq \hat{b} \tag{3-12}$$

Por lo tanto el problema se resume en resolver

$$\frac{1}{2}\, U^T H\, U + F\, U \tag{3-13}$$

$$\text{Sujeto a: } \hat{\mathrm{A}} U \leq \hat{b}$$

Para lo cual se suele utilizar un algoritmo de programación cuadrática que incluya las restricciones lineales de la variable manipulable.

Tal y como recogen numerosos trabajos (E. F. Camacho y C. Bordons 2007) y se enunciaba en la introducción, la eficacia de esta estrategia está comprobada, si bien dado que la implementación está basada en el manejo de operaciones matriciales de un orden en ocasiones elevado, se demanda un coste computacional que llega a ser la gran desventaja de usar este método en el caso de problemas muy complejos o con tiempos de muestreo bajos, por lo que en el apartado siguiente se verá un algoritmo que proporciona mejores tiempos de respuesta.

4 PROGRAMACIÓN CUADRÁTICA PARALELA

4.1 Introducción de la problemática

Como se presentaba en el capítulo anterior, a veces resolver el problema de optimización asociado a la función de coste que proporciona la acción de control es computacionalmente costoso, por lo que las aplicaciones donde implementar MPC pueden verse limitadas y no dar cabida a los sistemas de dinámicas rápidas.

Sabiendo que en todos los problemas de control digital el objetivo es dar respuesta antes de finalizar cada periodo de muestreo, puede resultar útil contar con la contribución de Matthew Brand et al (2011). En el citado trabajo se presenta un algoritmo iterativo para solucionar más rápido problemas de programación cuadrática, llamado programación cuadrática paralela (PQP por sus siglas en inglés). Este algoritmo cuenta además con el aliciente de ajustarse bien a los propósitos del presente trabajo, ya que se puede someter a paralelización de grano fino y por tanto ejecutar adecuadamente el algoritmo en la GPU, de manera que se consiga una implementación que ofrezca una rapidez mayor.

Al finalizar el capítulo siguiente se hará la comparativa entre distintas implementaciones de MPC; estas serán calcular el tiempo asociado a una programación cuadrática (QP), el tiempo asociado a una programación cuadrática paralela (PQP) en una GPU y el tiempo asociado a una programación PQP sin GPU, ya que también ofrece mejoras de velocidad incluso cuando se implementa en un solo procesador. Los resultados esperados serán obtener $tiempo\ PQP\ con\ GPU < tiempo\ PQP\ sin\ GPU < tiempo\ QP$, demostrándose la eficacia de las distintas estrategias y estableciendo un método mediante el cual la implementación de un algoritmo PQP en GPU posibilita utilizar MPC para sistemas con dinámicas rápidas, uno de los ámbitos que no podía cubrir cuando se desarrolló.

4.2 Implementación de la solución

Según se había expuesto en el capítulo anterior, tras plantear la función de coste y las restricciones lineales en forma de desigualdades, se había llegado a que había que resolver el problema primario

$$\frac{1}{2}\, U^T H\, U + F\, U \tag{4-1}$$

$$Sujeto\ a\colon \hat{A}U \leq \hat{b}$$

Antes de proceder con su resolución, es preciso hacer las siguientes suposiciones:

Suposición 1: $H > 0$, es decir, H es una matriz definida positiva, de manera que el problema de optimización sea convexo y se encuentre una solución.

Suposición 2: el problema cuadrático en la ecuación (4-1) es factible, lo que significa que existe una solución para el problema QP que satisface las restricciones. Hay que tener en cuenta que pueden surgir problemas no factibles, por ejemplo, en el caso de que las referencias varíen con el tiempo, en cuyo caso habrá que añadir una variable de holgura que dilate la región factible.

En lo que sigue y para facilitar el manejo de las restricciones generales del problema QP propuesto en (4-1), se propone el algoritmo PQP como una ley actualización multiplicativa usando la forma dual de Lagrange de la ecuación (4-1).

Para el problema primario de (4-1), hay que considerar la forma dual dada por

$$\min_{y}\{F(y) = \frac{1}{2}y^T Q_d y + y^T h_d\} \tag{4-2}$$

$$Sujeto\ a: y \geq 0$$

Donde $y \in \mathbb{R}^{N_y}$ es la variable dual, y $Q_d \geq 0$ (es decir, es semidefinida positiva) y h_d se obtienen bajo la suposición 1 como

$$Q_d = \hat{A}H^{-1}\hat{A}^T \tag{4-3}$$

$$h_d = \hat{b} + \hat{A}H^{-1}F^T \tag{4-4}$$

El óptimo U^* del problema primario en ecuación (4-1) puede ser obtenida del óptimo y^* del problema dual de la ecuación (4-2) usando la siguiente relación:

$$U^* = -H^{-1}(F^T + \hat{A}^T y^*) \tag{4-5}$$

Una vez que tenemos el problema así expresado, se resuelve el problema dual de la ecuación (4-2) implementando repetidas iteraciones de la siguiente regla, parando el proceso cuando se alcance la precisión deseada en la solución

$$y_i \leftarrow y_i \left[\frac{h_i^- + (Q^- y)_i}{h_i^+ + (Q^+ y)\_i}\right] \tag{4-6}$$

Donde $Q^+ = \max(Q_d, 0) + diag(r)$

$Q^- = \max(-Q_d, 0) + diag(r)$

$h^+ = \max(h_d, 0)$

$h^- = \max(-h_d, 0)$

Cabe precisar que $max\ (a, b)$ es cogido elemento a elemento; $diag\ (a)$ es una matriz diagonal formada a partir del vector a; y r es un vector no negativo especificado como $r = \max((diag(Q_d), \max(-Q_d, [\,], 2))$. Para mayor detalle de cómo obtener dichas expresiones y los criterios de convergencia del algoritmo, se recomienda consultar el texto original (Matthew Brand et al, 2011. *A Parallel Quadratic Programming Algorithm for Model Predictive Control*) del que se extrae la presente información.

5 PRESENTACIÓN Y RESOLUCIÓN DEL PROBLEMA MPC

5.1 Problema propuesto

Llegados a este punto, solo queda utilizar y sacar provecho de todas las herramientas y metodologías expuestas para resolver un problema de control predictivo.

En concreto, se resolverá un problema de dimensión 1, aunque se añaden fácilmente más dimensiones siguiendo las indicaciones del capítulo 3 para el caso general.

El problema elegido aborda la gestión de un almacén, en el cual la variable a controlar será el estado del mismo, es decir cuántas unidades se encuentran en él.

Se precisa entonces estimar la demanda, que disminuirá las unidades almacenadas y habrá que controlar la petición de unidades al proveedor, teniendo en cuenta que una gestión improvisada supondría una utilización descuidada e ineficiente del almacén.

El objetivo por lo tanto es mantener las unidades almacenadas apropiadas para satisfacer a la demanda sin que ello suponga tener el almacén atestado.

Un modelo del sistema sería:

$$x(k+1) = x(k) + u(k) - w(k), \tag{5-1}$$

donde $x(k)$ es el estado actual del almacén, $u(k)$ son las unidades pedidas al proveedor en el instante k, $w(k)$ son las unidades demandadas por los clientes en el instante k y la unidad de tiempo elegido para el sistema es el día.

Ahora que se cuenta con un modelo base de la presente metodología de control, se pueden hacer las predicciones a lo largo de un horizonte de predicción N dado un estado inicial, sabiendo que la variable de control serán las unidades pedidas y que las unidades demandadas no se pueden saber a priori, por lo que serán estimadas.

Siguiendo con la teoría del capítulo 3 tiene y estableciendo el horizonte de predicción en 5 días se tiene

$$X = \begin{bmatrix} x(1) \\ x(2) \\ x(3) \\ x(4) \\ x(5) \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} x(0) + \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} u(0) \\ u(1) \\ u(2) \\ u(3) \\ u(4) \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 \\ -1 & -1 & -1 & 0 & 0 \\ -1 & -1 & -1 & -1 & 0 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix} \begin{bmatrix} w(0) \\ w(1) \\ w(2) \\ w(3) \\ w(4) \end{bmatrix} \tag{5-2}$$

Una vez el modelo establecido, se propone la siguiente función de coste que penalizará el tener demasiadas unidades almacenadas, matriz Q, y el pedir demasiadas unidades al proveedor, matriz R. Por simplicidad se han elegido valores de $Q = R = 1$, pero igualmente se pueden dar otros valores

$$V = \sum_{k=0}^{4}(x(k+1)^T\, x(k+1) + u(k)^T\, u(k)) \tag{5-3}$$

Desarrollando la función V, tal y como indicaba el capítulo 3 se llega a

$$V \;=\; 2FU \;+\; U'H\,U \tag{5-4}$$

La función anterior es equivalente a optimizar la función:

$$½\, U'HU \;+\; FU \tag{5-5}$$

En este caso se pretende además minimizar la función satisfaciendo unas restricciones en las que tanto las unidades almacenadas como las unidades pedidas no superen unos límites:

$$-5 \le x(k) \le 10 \tag{5-6}$$

$$-1 \le u(k) \le 1 \tag{5-7}$$

Equivalentemente se puede escribir

$$\begin{bmatrix} 1 \\ -1 \end{bmatrix} x(k) \le \begin{bmatrix} 10 \\ 5 \end{bmatrix} \tag{5-8}$$

$$\begin{bmatrix} 1 \\ -1 \end{bmatrix} u(k) \le \begin{bmatrix} 10 \\ 1 \end{bmatrix} \tag{5-9}$$

Por último hay que expresar ambas restricciones como una restricción en la variable manipulada U antes de resolver

$$\frac{1}{2}\, U^T H\, U + F\, U \tag{5-10}$$

Sujeto a: $\hat{A}U \le \hat{b}$

Problema que ha de ser resuelto a cada instante ya que se va tomando la primera componente del vector u calculado y a continuación se va desplazando la ventana de las predicciones una unidad hacia delante antes de volver a empezar todo el proceso.

5.2 Resolución utilizando programación cuadrática convencional

La forma más inmediata de resolver el problema de programación cuadrática planteado es haciendo uso del comando *quadprog* de MATLAB, ya destinado a tal tarea. Se trata de una función potente que permite solucionar el problema con y sin restricciones, además de incluir y ser posible configurar otras opciones.

Aquí se puede ver el fragmento de código más relevante, estando el resto disponible para su consulta en el anexo A.

Resolución del problema MPC con quadprog

```
% Construcción de la función de coste
H = Gu'*Q_hat*Gu + R_hat;
F = x'*Gx'*Q_hat*Gu + W'*Gw'*Q_hat'*Gu;

% Restricciones escritas en forma matricial
AU = [Ax_hat*Gu;Au_hat];
bU = [bx_hat - Ax_hat*Gx*x - Ax_hat*Gw*W;bu_hat];

UMPC = quadprog(H,F,AU,bU);

% Se aplica solo la primera componente
u = UMPC(1);
x = A*x+B*u+D*Disturb(k);
```

La función *quadprog* recibe como parámetros las matrices de la función de coste y las matrices de las restricciones.

Note que tras resolver el problema de optimización solo la primera componente de la acción de control calculada es aplicada para que el estado evolucione y a continuación se empezaría de nuevo para el siguiente instante de simulación.

Así planteado y, una vez cargado en el programa los datos del problema, la simulación del sistema se ejecuta en un tiempo de 1.4384 s de media.

5.3 Resolución utilizando programación cuadrática paralela

Alternativamente, se puede hacer uso de lo expuesto en el capítulo 4 para resolver el problema gracias a la programación cuadrática paralela. Antes de aplicarla es preciso reformular el problema de optimización en la forma dual y seguir los pasos que allí se expusieron.

La implementación propuesta está desarrollada en el anexo B, siendo el fragmento de más interés:

Resolución del problema con PQP y sin implementación en GPU

```
% Paso 1: poner el sistema de la forma
%
% min_u 1/2*U'*H*U + F'*U
%  s.t. AU*U <= bU

% Paso 2: fijar tolerancia y número de iteraciones
max_iter = inf;
tol = 1e-7;

% Obtener la forma dual
% min_y 1/2*y'Q_d*y + y'*h_d
%   s.t. y >= 0
Q_d = AU*inv(H)*AU';
h_d = bU + AU*inv(H)*F';
n = size(Q_d,1);
```

```
% Estimación inicial
y0 = ones(n,1);

% Valor de r
r = max(diag(Q_d),max(-Q_d,[],2));

Qp = max(Q_d,0) + diag(r);
Qm = max(-Q_d,0) + diag(r);
hp = max(h_d,0);hm = max(-h_d,0);

% Paso 3: Iterar hasta encontrar el óptimo
y = y0;
it = 0;
tr = 1;
while tr==1
y_prev = y;
y = y.*(hm + Qm*y)./(hp + Qp*y);
it = it + 1;
diff = max(abs(y - y_prev));
if it >= max_iter
   return;
end
if diff <= tol
    tr = 0;
end
end

% Paso 4: Devolver la solución en u
uPQP = -inv(H)*(F' + AU'*y);

% Se aplica solo la primera componente
u = uPQP(1);
x = A*x+B*u + D*Disturb(k);
```

En el fragmento mostrado se observa que hay que resolver el problema de optimización asociado a la forma dual cada vez que se actualiza el estado y se quiera calcular la entrada que el controlador deba realizar en el siguiente instante.

Hay que comentar que la estimación inicial no es muy relevante y que se converge rápido al óptimo según la tolerancia fijada. De nuevo, cada vez que se recupera la señal de control para el horizonte temporal establecido en ese instante, interesa solo la primera componente y se actualizan a continuación todas las variables.

En las mismas condiciones que en la resolución del problema por el método anterior, ahora se tarda 0.6978 s de media en ejecutar.

5.4 Resolución utilizando kernels y programación cuadrática paralela

A partir de la implementación anterior y acorde al objeto de este trabajo, se combinan todas las herramientas consideradas con el propósito de conseguir resultados aún más rápidos.

De este modo, se usa el Run and Time de MATLAB para localizar el código que más tarda en ejecutarse y que

ralentiza la aplicación. En este caso, se encuentra que la sentencia $y = y * (hm + Qm * y)/(hp + Qp * y)$ es la que más tarda en ejecutarse, teniendo además la característica de que al estar formada por operaciones matriciales, cumple adecuadamente con las condiciones exigidas a la hora de ser ejecutada en una unidad de procesamiento gráfico.

La siguiente etapa pasa por escribir dicha sentencia en un *kernel* utilizando CUDA, y así sustituir el código equivalente en MATLAB con la perspectiva de que este método suponga una mejora.

El código escrito en CUDA se encuentra en el anexo C y en el se puede ver cómo se sustituye la sentencia antes mencionada. Para ello, cada hilo calcula una componente del vector y resultado, tomando como datos la componente que corresponda al identificador del hilo. Los hilos declarados en exceso y que por tanto se salgan de la dimensión de las variables de entrada serán anulados.

Una vez hecho esto, en la aplicación de MATLAB hay que cargar el *kernel*, de manera que sea tratado como un objeto y así configurar sus atributos (número de hilos y bloques que se necesitan), pasar los datos a la GPU y, a cada vez que se requiera hacer el cálculo de la sentencia sustituida, llamar al *kernel* para su evaluación.

Con todos estos cambios disponibles en el anexo D, lo más relevante se muestra a continuación:

Resolución del problema con PQP y con implementación en GPU

```
% Cargamos el kernel
kernel = parallel.gpu.CUDAKernel('pqp.ptx','pqp.cu');

% Configuramos los hilos
kernel.ThreadBlockSize = [1,1,1];
kernel. GridSize = [n,1,1];

% Paso 1: poner el sistema de la forma
    %
    % min_u 1/2*U'*H*U + F'*U
    % s.t. AU*U <= bU

% Paso 2: fijar tolerancia y número de iteraciones
max_iter = inf;
tol = 1e-7;

% Obtener la forma dual
Q_d = AU/H*AU';
h_d = bU + AU/H*F';
n = size(Q_d,1);

% Estimación inicial
y = ones (n,1);
```

```
% valor de r
r = max(diag(Q_d),max(-Q_d,[],2));
Qp = max(Q_d,0) + diag(r);
Qm = max(-Q_d,0) + diag(r);
hp = max(h_d,0);
hm = max(-h_d,0);

% Paso 3: Iterar hasta encontrar el óptimo
y = y0;
it = 0;
tr = 1;
while tr==1
  y_prev = y;

  % Se pasan variables kernel
  Qp = single(Qp);
  Qm = single(Qm);
  hp = single(hp);
  hm = single(hm);
  n = uint16(n);
  y = single(y);

  d_hp = gpuArray(hp);
  d_hm = gpuArray(hm);
  d_Qm = gpuArray(Qm);
  d_Qp = gpuArray(Qp);
  d_y = gpuArray(y);
  d_n = gpuArray(n);

  % Se aplica el kernel
  d_y = feval(kernel,d_y,d_hm,d_hp,d_Qm,d_Qp,d_n);

  % Se recupera resultado
  y = gather(d_y);
  y = double(y);
  it = it + 1;
  diff = max(abs(y - y_prev));
  if it >= max_iter
    return;
  end
  if diff <= tol
  tr = 0;
  end
end

% Paso 4: Devolver la solución en u
uPQP = -eye(N)/H*(F' + AU'*y);

% Se aplica solo la primera componente
u = uPQP(1);
x = A*x + B*u + D*Disturb(k);
```

Conviene señalar que el objeto *kernel* se crea y se configura solo una vez, antes de empezar las iteraciones. Como detalle de la implementación, se ve que hay que ajustar el formato de los datos antes de que sean pasados a la GPU. Esto es porque en MATLAB se trabaja con doble precisión, ocupando 8 bytes y en CUDA las variables están declarada como flotante, ocupando 4 bytes o como entero, ocupando 2 bytes.

Al recuperar el resultado del *kernel*, se fuerza a que sea de tipo doble para que no haya problemas de tipo de datos.

Las distintas implementaciones anteriores se llevan a cabo en una CPU Intel® Core™ i3 a 1.7 GHz y en una GPU Nvidia® GeForce® 920M.

Después de proceder con su ejecución, se tiene un tiempo de 0.9706 s de media.

6 CONCLUSIONES Y FUTURAS LÍNEAS

6.1 Conclusiones

Tras haber simulado y resuelto el problema MPC de tres formas diferentes se ha comprobado que efectivamente el trabajo de Matthew Brand et al (2011) con el algoritmo de programación paralela, supone una mejora al algoritmo de optimización dado por la programación cuadrática convencional siempre que el problema tenga una dimensión reducida.

Además se confirman los beneficios de complementar el procesamiento de la aplicación con una GPU, escribiendo parte del código en ella.

Como contribución de este trabajo está el presentar la comparativa de estos tres métodos elegidos para la aceleración de las aplicaciones en las que la metodología MPC supone resolver un problema de optimización cada vez que se quiera predecir el estado del sistema, con el coste computacional que ello supone.

La siguiente tabla resume las simulaciones realizadas empleando el horizonte de predicción de 5 días y, con el objetivo de sacar más conclusiones, se simula también incrementando el horizonte de 50 en 50 días.

Tabla 6-1. Resultados

	Tiempo empleado		
Horizonte de predicción	MPC resuelto con QP	MPC resuelto con PQP	MPC resuelto con PQP y CUDA
N = 5	1.4384 s	0.6978 s	0.9706 s
N = 50	4.8163 s	23.1787 s	3.5486 s
N = 100	11.2359 s	103.2264 s	6.1877 s
N = 150	25.6132 s	359.2239 s	16.3059 s
N = 200	50.2100 s	714.3067 s	24.7207 s
N = 250	88.8827 s	1225.1705 s	61.1775 s
N = 300	139.5259 s	2322.2025 s	143.7387 s
N = 350	214.9441 s	2474.7462 s	228.9937 s
N = 400	315.2899 s	4218.0414 s	244.4721 s
N = 450	417.0601 s	5113.4325 s	324.2032 s
N = 500	546.5981 s	6797.5969 s	506.1090 s

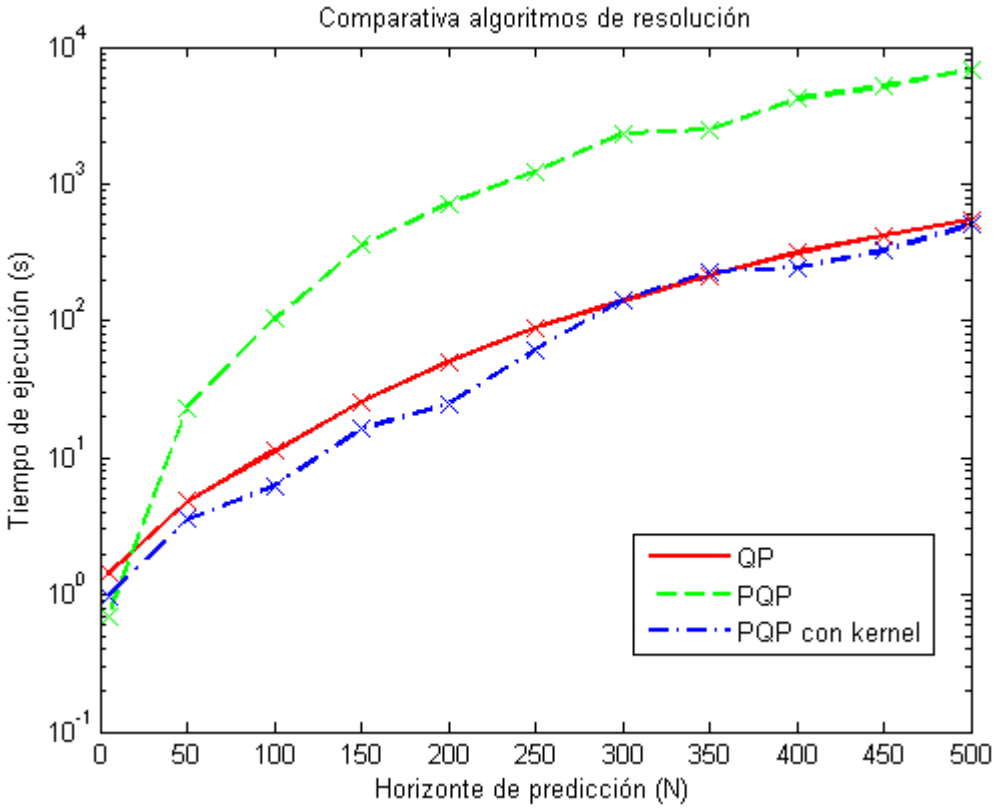

Figura 6-1. Resultado de las simulaciones para distintos horizontes de predicción.

A la vista de los resultados, utilizar el algoritmo de programación cuadrática paralela supone emplear menos tiempo que la programación cuadrática convencional solo hasta cierto punto, ya que hay un momento en el que la dimensión del problema lleva asociada que la sentencia identificada como crítica en el apartado 5.4 penalice el tiempo de ejecución, disparándose el coste de ejecución, y es en consecuencia preferible sin lugar a dudas la programación cuadrática de MATLAB. Por lo tanto, PQP ejecutado en la CPU solo se recomienda en el caso de que el problema tenga un horizonte de predicción moderado.

La combinación del algoritmo de programación cuadrática paralela y escribir parte de su código en CUDA para que lo ejecute la GPU supone ejecutar la aplicación más rápido que con QP, aunque para N suficientemente alto se pierde la mejoría. Se puede atribuir este hecho a que la función de QP está muy depurada y optimizada, sin embargo la transferencia repetida de matrices cada vez más grandes desde la memoria del ordenador a la memoria del dispositivo hace que los tiempos de ejecución de ambos se igualen. Con respecto a PQP, la sentencia crítica antes mencionada lastra el tiempo de ejecución, por lo que el hecho de ser llevada a cabo con un kernel es un acierto y se mejora notablemente el tiempo de ejecución.

Un punto a tener en cuenta a la hora de utilizar este método con problemas reales es que pueden surgir problemas de optimización que además de restricciones de desigualdad en sus variables, cuenten con restricciones de igualdad entre ellas. Este hecho, habitual en la práctica, no es tratado con facilidad por el método implementado y, en su defecto, hay que pensar alternativas para introducir las restricciones de igualdad.

Como alternativa se puede recurrir al método de los múltiplicadores de Lagrange o incluso directamente sustituir cada restricción de este tipo por dos restricciones de desigualdad que casi fuercen la igualdad, haciendo uso de una cantidad *eps* que bien puede ser fija o bien ser incluida en el problema como variable a minimizar también.

En base a este trabajo, se puede concluir que se puede mejorar la implementación de una aplicación sencilla haciendo uso del algoritmo de programación cuadrática paralela llevando a cabo una parte en una tarjeta gráfica.

En el caso de un problema más real, tal y como se tuvo oportunidad de comprobar en un último experimento, el problema de optimización tenía un tamaño considerable y el objetivo era reducir en lo posible el tiempo de ejecución, ya que para incluso para horizontes de predicción pequeños ($N = 2$) la resolución utilizando el comando *quadprog* tardaba 6h 44min.

Para abordar dicho problema, se tuvieron que acomodar las restricciones de igualdad como desigualdades para que pudiera ser tratado por el método de programación cuadrática paralela y se sustituyó cada problema de optimización por el *kernel* correspondiente.

El resultado de la prueba no fue satisfactorio, señalándose como causa principal aquella que se enunciaba en los comentarios de los resultados obtenidos en el problema sencillo: se debe evitar el uso continuo del puerto PCI-Express para transferir datos siempre que se quiera tener un buen rendimiento. Así, comprobando las dimensiones de las principales variables y sumado al hecho de tener que pasarlas repetidamente desde la memoria principal de la CPU al dispositivo y viceversa, tal y como está recogido en el algoritmo propuesto en el Anexo D, se reúnen las condiciones desfavorables que llevan a perder las mejoras obtenidas al abordar los cálculos más intensivos en paralelo.

Tabla 6-2. Dimensiones de las principales variables según el método empleado para N = 2

QP con ambos tipos de restricciones	**QP sin restricciones de igualdad explícitas**	**PQP y CUDA sin restricciones de igualdad explícitas**
H [354 x 354]	H [354 x 354]	hm [776 x 1]
F [1 x 354]	F [1 x 354]	hp [776 x 1]
Au [708 x 354]	Au [776 x 354]	Qm [776 x 776]
bu [708 x 1]	bu [776 x 1]	Qp [776 x 776]
Aeq [34 x 354]		y [776 x 1]
beq [34 x 1]		

Se entiende entonces el por qué del resultado, y se desprende que, además de controlar las dimensiones del problema, explotar la estructura del mismo es clave para que la aplicación tenga éxito.

6.2 Futuras líneas de trabajo

Según los resultados de este trabajo, queda justificado el uso compartido de CPU y GPU, así como el empleo de un algoritmo de programación cuadrática paralela para resolver el problema de controlar un sistema mediante control predictivo basado en modelo siempre que se cumplan los requisitos establecidos.

Así, con las herramientas presentadas con este trabajo, la expectativa sería utilizarlas para resolver un problema más complejo que el aquí empleado para hacer las demostraciones teniendo siempre presente las conclusiones obtenidas.

REFERENCIAS

[1] ARMYR, DANIEL y DOHERTY, DAN. 2013. Mathworks. *Prototyping Algorithms and Testing CUDA Kernels in MATLAB.* [Consulta 12 marzo 2017]. Disponible en: https://es.mathworks.com/company/newsletters/articles/prototyping-algorithms-and-testing-cuda-kernels-in-matlab.html

[2] BRAND, M. y CHEN, D. 2011. *Parallel Quadratic Programming for Image Processing.*

[3] BRAND, MATTHEW et al. 2011. *A Parallel Quadratic Programming Algorithm for Model Predictive Control.*

[4] F. CAMACHO, E. y BORDONS, C. 2007. *Model Predictive Control*. 2nd. ed. Springer.

[5] GADE-NIELSEN, NICOLAI FOG; DAMMANN, BERND y JORGENSEN, JOHN BAGTERP. 2014. *Interior Point Method on GPU with application to Model Predictive Control.*

[6] GADE-NIELSEN, NICOLAI FOG; JORGENSEN, JOHN BAGTERP y DAMMANN, BERND. 2012. *MPC Toolbox with GPU Accelerated Optimization Algorithms.*

[7] HERNÁNDEZ RODRIGUEZ, GONZALO. 2015 Proyecto fin de carrera: *Control de estabilidad basado en MPC para un vehículo eléctrico con motores en rueda.*

[8] MATHWORKS. 2017. *Run CUDA or PTX Code on GPU.* [Consulta 12 marzo 2017]. Disponible en: https://es.mathworks.com/help/distcomp/run-cuda-or-ptx-code-on-gpu.html

[9] NVIDIA® 2016. *CUDA C Programming guide*. [Consulta 20 febrero 2017]. Disponible en: http://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf

[10] OCAMPO MARTÍNEZ, CARLOS y MAESTRE, JOSÉ M. *Low level implementation of MPC.*

[11] REESE, JILL y ZARANEK, SARAH. 2011. Mathwoks. *GPU Programming in MATLAB.* [Consulta 12 marzo 2017]. Disponible en:https://es.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html

[12] UJALDON, MANUEL. 2014. *Programando la GPU con CUDA.* Curso en el Dpto de Matemáticas e Informática. Junio 23-25, 2014

[13] ZAMBRANO J. y GONZÁLEZ A. 2013. *Implementación de un algoritmo de control predictivo en espacio de estados sobre una plataforma de simulación desarrollada en MATLAB.*

GLOSARIO

Anfitrión (Host). Hardware formado por la unidad central de procesos y la memoria del sistema.

Bloque. Conjunto de hilos.

CPU (Central Processing Unit). La unidad central de procesos en un ordenador es la responsable de los cálculos y controla y supervisa las otras partes del ordenador. La CPU realiza las operaciones lógicas de punto flotante sobre los datos almacenados en la memoria.

CUDA. Tecnología de computación paralela desarrollada por Nvidia que consiste en una arquitectura paralela de computación y librerías para el desarrollo de programación en una GPU.

Dispositivo (Device). Hardware que contiene la unidad de procesamiento gráfico y su memoria asociada.

GPU (Graphics Processing Unit). La unidad de procesamiento grafico es un chip programable originalmente destinado al renderizado de gráficos. La estructura masivamente paralela con la que cuenta la GPU la hace más efectiva que las CPU (al ser su propósito más general) a la hora de procesar grandes bloques de datos a ejecutar en paralelo.

Hilo. Unidad mínima de concurrencia en una GPU.

Kernel. Código escrito especificamente para su ejecución en una GPU. Los kernels son funciones que pueden se llevadas a cabo por un gran número de hilos. El paralelismo se consigue al ejecutar cada hilo el mismo programa sobre datos diferentes.

Malla. Conjunto de bloques.

Núcleo. Unidad independiente que realiza cálculos en un chip CPU o GPU. Los núcleos CPU y GPU no son equivalentes: los primeros están diseñados para programas de propósito general mientras que nos segundos realizan operaciones más especializadas.

ANEXO A

```
% Resolución del problema MPC con quadprog

% Ejemplo de una tienda
% Modelo general de un sistema
%    x(k+1)=Ax(k)+Bu(k)+Dw(k)
% Modelo del sistema
%    x(k+1)=x(k)+u(k)-w(k)

% Definición de las matrices del modelo
A = 1;
B = 1;
D = -1;

% Estado inicial x(0)
x = 5;

% Definición del horizonte temporal y otras variables útiles
N = 5;
nx = size(A,2); % Número de estados.
nu = size(B,2); % Número de entradas.
nw = size(D,2); % Número de perturbaciones.

% Los futuros estados se pueden escribir como
%    X = Gx*x + Gu*U + Gw*W

% Construcción de Gx
for i=1:N
    Gx((i-1)*nx+1:i*nx,:) = A^i;
end

% Construcción de Gu
Gu = zeros(N*nx,N*nu);
for i=1:N
    for j=1:i
        Gu((i-1)*nx+1:i*nx,(j-1)*nu+1:j*nu) = A^(i-j)*B;
    end
end

% Construcción de Gw
Gw = zeros(N*nx,N*nw);
for i=1:N
    for j=1:i
        Gw((i-1)*nx+1:i*nx,(j-1)*nw+1:j*nw) = A^(i-j)*D;
    end
end

% Definición de la función de coste y las perturbaciones esperadas
Q = 1;
R = 1;
W = ones(N,1);

% Construcción de R_hat
R_hat = kron(eye(N),R);
% Construcción de Q_hat
Q_hat = kron(eye(N),Q);
```

```
% Construcción de la función de coste
H = Gu'*Q_hat*Gu + R_hat;
F = x'*Gx'*Q_hat*Gu + W'*Gw'*Q_hat*Gu;

% Se establecen restricciones en x y u del tipo
%     -5 < x(k) < 10 y -1 < u(k) <1
Ax = [1;-1];
bx = [10;5];
Au = [1;-1];
bu = [1;1];

% Se transforma lo anterior en restricciones en U
Au_hat = kron(eye(N),Au);
bu_hat = kron(ones(N,1),bu);
Ax_hat = kron(eye(N),Ax);
bx_hat = kron(ones(N,1),bx);

% Se condensa las restricciones en x y u a solo U, ya que es lo único que
% se puede controlar
AU = [Ax_hat*Gu;Au_hat];
bU = [bx_hat - Ax_hat*Gx*x - Ax_hat*Gw*W;bu_hat];

% Metodología MPC en acción
Simlength = 100; %Tamaño de la simulación
Xhist = x;
Uhist = [];
Disturb = normrnd(0.5,1,Simlength+N,1);
t = 1:Simlength;

tic;
%Bucle de simulación
for k = t
    % Perturbaciones esperadas
    W = Disturb(k:k+N-1) + normrnd(0,0.2,N,1);

    % Se actualiza matrices de control para el estado actual y perturbaciones
    F = x'*Gx'*Q_hat*Gu + W'*Gw'*Q_hat'*Gu;
    bU = [bx_hat - Ax_hat*Gx*x - Ax_hat*Gw*W;bu_hat];

    UMPC = quadprog(H,F,AU,bU);

    % Se aplica solo la primera componente
    u = UMPC(1);
    x = A*x+B*u+D*Disturb(k);
    Xhist = [Xhist x];
    Uhist = [Uhist u];
end

t2 = toc;
fprintf('El tiempo ha sido: %f\n',t2);

Xhist = Xhist(1:Simlength); figure;
plot(t,Xhist,'g',t,Uhist,
```

ANEXO B

```
% Resolución del problema PQP sin implementación en GPU

% Ejemplo de una tienda
% Modelo general de un sistema
%    x(k+1)=Ax(k)+Bu(k)+Dw(k)
% Modelo del sistema
%    x(k+1)=x(k)+u(k)-w(k)

% Definición de las matrices del modelo
A = 1;
B = 1;
D = -1;

% Estado inicial x(0)
x = 5;

% Definición del horizonte temporal y otras variables útiles
N = 5;
nx = size(A,2); % Número de estados.
nu = size(B,2); % Número de entradas.
nw = size(D,2); % Número de perturbaciones.

% Los futuros estados se pueden escribir como
%    X = Gx*x + Gu*U + Gw*W

% Construcción de Gx
for i=1:N
    Gx((i-1)*nx+1:i*nx,:) = A^i;
end

% Construcción de Gu
Gu = zeros(N*nx,N*nu);
for i=1:N
    for j=1:i
        Gu((i-1)*nx+1:i*nx,(j-1)*nu+1:j*nu) = A^(i-j)*B;
    end
end

% Construcción de Gw
Gw = zeros(N*nx,N*nw);
for i=1:N
    for j=1:i
        Gw((i-1)*nx+1:i*nx,(j-1)*nw+1:j*nw) = A^(i-j)*D;
    end
end

% Definición de la función de coste y las perturbaciones esperadas
Q = 1;
R = 1;
W = ones(N,1);


% Construcción de R_hat
R_hat = kron(eye(N),R);
```

```
% Construcción de Q_hat
Q_hat = kron(eye(N),Q);

% Construcción de la función de coste
H = Gu'*Q_hat*Gu + R_hat;
F = x'*Gx'*Q_hat*Gu + W'*Gw'*Q_hat*Gu;

% Se establecen restricciones en x y u del tipo
%     -5 < x(k) < 10 y -1 < u(k) <1
Ax = [1;-1];
bx = [10;5];
Au = [1;-1];
bu = [1;1];

% Se transforma lo anterior en restricciones en U
Au_hat = kron(eye(N),Au);
bu_hat = kron(ones(N,1),bu);
Ax_hat = kron(eye(N),Ax);
bx_hat = kron(ones(N,1),bx);

% Se condensa las restricciones en x y u a solo U, ya que es lo único que
% se puede controlar
AU = [Ax_hat*Gu;Au_hat];
bU = [bx_hat - Ax_hat*Gx*x - Ax_hat*Gw*W;bu_hat];

% Metodología MPC con PQP en acción
Simlength = 100; %Tamaño de la simulación
Xhist = x;
Uhist = [];
Disturb = normrnd(0.5,1,Simlength+N,1);
t = 1:Simlength;

% Se saca del bucle aquello que solo se calcula una vez para no retrasar
Q_d = AU/H*AU';

tic;
%Bucle de simulación
for k = t
    % Perturbaciones esperadas
    W = Disturb(k:k+N-1) + normrnd(0,0.2,N,1);

    % Se actualiza las matrices de control para el estado actual y
    perturbaciones
    F = x'*Gx'*Q_hat*Gu + W'*Gw'*Q_hat'*Gu;
    bU = [bx_hat-Ax_hat*Gx*x - Ax_hat*Gw*W;bu_hat];

    % Paso 1: poner el sistema de la forma
    %
    % min_u 1/2*U'*H*U + F'*U
    %  s.t. AU*U <= bU

    % Paso 2: fijar tolerancia y número de iteraciones
    max_iter = inf;
    tol = 1e-7;

    % Obtener la forma dual
    % min_y 1/2*y'Q_d*y + y'*h_d
    %   s.t. y >= 0
```

```
    % Q_d = AU*inv(H)*AU';
    % h_d = bU + AU*inv(H)*F';
    h_d = bU + AU/H*F';

    n = size(Q_d,1);

    % Estimación inicial
    y0 = ones(n,1);

    % Valor de r
    r = max(diag(Q_d),max(-Q_d,[],2));

    Qp = max(Q_d,0) + diag(r);
    Qm = max(-Q_d,0) + diag(r);
    hp = max(h_d,0);
    hm = max(-h_d,0);

    % Paso 3: Iterar hasta encontrar el óptimo
    y = y0;
    it = 0;
    tr = 1;
    while tr==1
    y_prev = y;
    y = y.*(hm + Qm*y)./(hp + Qp*y);
    it = it + 1;
    diff = max(abs(y - y_prev));
    if it >= max_iter
        return;
    end
    if diff <= tol
        tr = 0;
    end
    end

    % Paso 4: Devolver la solución en u
    uPQP = -eye(N)/H*(F' + AU'*y);

    % Se aplica solo la primera componente
    u = uPQP(1);
    x = A*x+B*u + D*Disturb(k);
    Xhist = [Xhist x];
    Uhist = [Uhist u];
end

t2 = toc;
fprintf('El tiempo ha sido: %f\n',t2);

Xhist = Xhist(1:Simlength);
plot(t,Xhist,'g',t,Uhist,'r')
```

ANEXO C

Programación del kernel para la sustitución de una sentencia en el algoritmo PQP

```
__global__ void pqp(float *y, float *hm, float *hp, float *Qm, float *Qp, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    float num = 0;
    float den = 0;
    float result = 0;

       if (i < n)
       {
              for (int j = 0;j < n;j++)
              {
                  num += Qm[i+j*n] * y[j];
                  den += Qp[i+j*n] * y[j];
              }

              num += hm[i];
              den += hp[i];
              result = num / den;
       }
       y[i] = result;
       return;
}
```

ANEXO D

```
% Resolución del problema PQP utilizando kernel de CUDA

% Ejemplo de una tienda
% Modelo general de un sistema
%    x(k+1)=Ax(k)+Bu(k)+Dw(k)
% Modelo del sistema
%    x(k+1)=x(k)+u(k)-w(k)

% Definición de las matrices del modelo
A = 1;
B = 1;
D = -1;

% Estado inicial x(0)
x = 5;

% Definición del horizonte temporal y otras variables útiles
N = 5;
nx = size(A,2); % Número de estados.
nu = size(B,2); % Número de entradas.
nw = size(D,2); % Número de perturbaciones.

% Los futuros estados se pueden escribir como
%    X = Gx*x + Gu*U + Gw*W

% Construcción de Gx
for i=1:N
    Gx((i-1)*nx+1:i*nx,:) = A^i;
end

% Construcción de Gu
Gu = zeros(N*nx,N*nu);
for i=1:N
    for j=1:i
        Gu((i-1)*nx+1:i*nx,(j-1)*nu+1:j*nu) = A^(i-j)*B;
    end
end

% Construcción de Gw
Gw = zeros(N*nx,N*nw);
for i=1:N
    for j=1:i
        Gw((i-1)*nx+1:i*nx,(j-1)*nw+1:j*nw) = A^(i-j)*D;
    end
end

% Definición de la función de coste y las perturbaciones esperadas
Q = 1;
R = 1;
W = ones(N,1);

% Construcción de R_hat
R_hat = kron(eye(N),R);
```

```
% Construcción de Q_hat
Q_hat = kron(eye(N),Q);

% Construcción de la función de coste
H = Gu'*Q_hat*Gu + R_hat;
F = x'*Gx'*Q_hat*Gu + W'*Gw'*Q_hat*Gu;

% Se establecen restricciones en x y u del tipo
%     -5 < x(k) < 10 y -1 <u(k) <1
Ax = [1;-1];
bx = [10;5];
Au = [1;-1];
bu = [1;1];

% Se transforma lo anterior en restricciones en U
Au_hat = kron(eye(N),Au);
bu_hat = kron(ones(N,1),bu);
Ax_hat = kron(eye(N),Ax);
bx_hat = kron(ones(N,1),bx);

% Se condensa las restricciones en x y u a solo U, ya que es lo único que
% se puede controlar
AU = [Ax_hat*Gu;Au_hat];
bU = [bx_hat - Ax_hat*Gx*x - Ax_hat*Gw*W;bu_hat];

% Metodología MPC con PQP con kernel en acción
Simlength = 100; %Tamaño de la simulación
Xhist = x;
Uhist = [];
Disturb = normrnd(0.5,1,Simlength+N,1);
t = 1:Simlength;

% Se saca del bucle aquello que solo se calcula una vez para no retrasar
Q_d = AU/H*AU';

max_iter = inf;
tol = 1e-7;
n = size(Q_d,1);

% Estimación inicial
y0 = ones(n,1);

% valor de r
r = max(diag(Q_d),max(-Q_d,[],2));

Qp = max(Q_d,0) + diag(r);
Qm = max(-Q_d,0) + diag(r);

% Se pasan estas dos matrices que no cambian a la gpu
Qp = single(Qp);
Qm = single(Qm);
n = uint16(n);
d_Qp = gpuArray(Qp);
d_Qm = gpuArray(Qm);
d_n = gpuArray(n);

% Cargamos el kernel
kernel = parallel.gpu.CUDAKernel('pqp.ptx','pqp.cu');
```

```
% Configuramos los hilos
kernel.ThreadBlockSize = [1,1,1];
kernel.GridSize = [n,1,1];

tic;
%Bucle de simulación
for k = t
    % Perturbaciones esperadas
    W = Disturb(k:k+N-1)+normrnd(0,0.2,N,1);

    % Se actualiza matrices de control para el estado actual y perturbaciones
    F = x'*Gx'*Q_hat*Gu + W'*Gw'*Q_hat'*Gu;
    bU = [bx_hat - Ax_hat*Gx*x - Ax_hat*Gw*W;bu_hat];

    % Paso 1: poner el sistema de la forma
    %
    % min_u 1/2*U'*H*U + F'*U
    % s.t. AU*U <= bU

    % Paso 2: fijar tolerancia y número de iteraciones
    % max_iter = inf;
    % tol = 1e-7;

    % Obtener la forma dual
    % Q_d = AU*inv(H)*AU';
    % h_d = bU + AU*inv(H)*F';
    h_d = bU + AU/H*F';

    % n = size(Q_d,1);

    % Estimación inicial
    % y0 = ones(n,1);
    % valor de r
    % r = max(diag(Q_d),max(-Q_d,[],2));

    % Qp = max(Q_d,0) + diag(r);
    % Qm = max(-Q_d,0) + diag(r);
    hp = max(h_d,0);
    hm = max(-h_d,0);

    % Paso 3: Iterar hasta encontrar el óptimo
    y = y0;
    it = 0;
    tr = 1;
    while tr==1
    y_prev = y;

    % Se pasan variables al kernel
    hp = single(hp);
    hm = single(hm);
    y = single(y);
    d_hp = gpuArray(hp);
    d_hm = gpuArray(hm);
    d_y = gpuArray(y);

    % Se aplica el kernel
    d_y = feval(kernel,d_y,d_hm,d_hp,d_Qm,d_Qp,d_n);
    % y = y.*(hm+Qm*y)./(hp+Qp*y);

    % Se recupera resultado
    y = gather(d_y);
```

```
    y = double(y);
    it = it + 1;
    diff = max(abs(y - y_prev));
    if it >= max_iter
        return;
    end
    if diff <= tol
        tr = 0;
    end
    end

    % Paso 4: Devolver la solución en u
    uPQP = -eye(N)/H*(F' + AU'*y);

    % Se aplica solo la primera componente
    u = uPQP(1);
    x = A*x + B*u + D*Disturb(k);
    Xhist = [Xhist x];
    Uhist = [Uhist u];
end
t2 = toc;
Xhist = Xhist(1:Simlength);
plot(t,Xhist,'g',t,Uhist,'r')
fprintf('El tiempo ha sido: %f\n',t2);
```