Extracción paralela de valores propios en matrices Toeplitz simétricas usando hardware gráfico

Tesis de Máster en Computación Paralela y Distribuida de:

Leandro Graciá Gil

Dirigida por:

Antonio Manuel Vidal Maciá

Índice

1. Introducción

En la actualidad los procesadores están alcanzando sus límites físicos: ya no crecen en frecuencia sino en núcleos, y cada vez más se está convirtiendo en una necesidad rediseñar algoritmos originalmente pensados para una única CPU de forma que puedan aprovechar adecuadamente el hardware disponible hoy en día. Éste es uno de los principales cometidos sobre los que se centra el estudio de este máster en sus distintas ramas.

No obstante además de los procesadores multicore y los clusters se están abriendo otros campos susceptibles a la computación paralela: los procesadores gráficos. Este tipo de hardware, diseñado para la aceleración de cálculos gráficos 3D, posee realmente un tremendo potencial alcanzando valores de hasta 768 GFlops en las últimas tarjetas disponibles. Además presentan un elevado nivel de paralelismo alcanzando incluso los 256 cores a cerca de 1500 MHz cada uno, así como anchos de banda de memoria de hasta 128 GB/segundo dejando atrás a multitud de CPUs.

El objetivo de esta tesis no es otro que tratar de adentrarse en este nuevo campo que se abre para la computación paralela adaptando y resolviendo un problema tradicional como es el cálculo de valores propios, aprovechando las últimas posibilidades ofrecidas por el hardware gráfico moderno. En concreto, se tratará de aprovechar el paralelismo inherente de las GPUs para acelerar la extracción de valores y vectores propios en matrices Toeplitz simétricas utilizando una variación aquí propuesta del método *Shift and Invert 2-way* de Lanczos e implementada usando la arquitectura de programación en GPUs CUDA.

Finalmente y de acuerdo al carácter investigador de esta tesis, se espera que el esfuerzo invertido en su realización converja en la creación de un artículo describiendo los matices más relevantes del proceso, con el objetivo de su posterior publicación.

1.1. Estado del arte

Este trabajo se nutre de la investigación realizada en dos campos distintos: los avances en métodos numéricos para el cálculo de valores propios y las novedades en el campo de la computación de propósito general en GPUs. Este último está resultando especialmente fértil para los investigadores, ya que han conseguido adaptar a GPUs y acelerar todo tipo de cálculos y algoritmos procedentes de prácticamente todas las ramas de la informática y multitud de ingenierías, desde la biotecnología y la astrofísica hasta la meteorología pasando por disciplinas como la criptografía o el reconocimiento del habla.

Existen ejemplos notables al respecto, muchos de ellos relacionados con la simulación numérica y directamente con la resolución de sistemas lineales. Los más importantes pueden encontrarse directamente en la web oficial de CUDA. En estos términos caben destacar artículos como [9], que fue una de las primeras aproximaciones al cálculo de FFTs y descomposiciones QR en la GPU, o el artículo [12], parte del proyecto *FLAME*, que trata de acelerar la resolución de sistemas lineales densos mediante el uso de múltiples aceleradores hardware, entre ellos las GPUs y la arquitectura CUDA ofreciendo buenos resultados.

También merecen mención algunos destacados ejemplos de simulación numérica con GPUs. Por ejemplo [10], donde se utilizan las últimas capacidades del hardware gráfico para acelerar y animar la simulación de fluidos mediante la resolución numérica de las famosas ecuaciones de Navier-Stokes en tres dimensiones, logrando speed-ups de aproximadamente 55x respecto al cálculo en CPU. Otro ejemplo de este tipo de uso puede encontrarse en [2], donde se compara el uso de CPUs y varios métodos de programación de propósito general en GPUs, entre ellos CUDA, para la resolución de las ecuaciones de Euler de dinámica de fluidos obteniendo speed-ups de 29x en 2D y 16x en 3D.

Además de los artículos anteriores, resulta especialmente interesante [4] en el cual las circunstancias del entorno hardware han propiciado la investigación de entornos de precisión mixta, donde sólo algunos de los cálculos más críticos se realizan en doble precisión utilizando simple precisión para el resto con el notable speed-up que esto conlleva. Concretamente, este tema está investigándose en mayor profundidad y parece resultar bastante prometedor, ya que sus resultados pueden extenderse fácilmente a otros entornos y multitud de librerías numéricas ya existentes.

Volviendo a la temática original, esta no es la primera vez que trata de utilizarse el hardware gráfico para el cálculo de valores propios. Existe ya un proyecto publicado en propio SDK oficial de CUDA [8] que paraleliza el tradicional método de la bisección para tratar de aproximar valores propios en un intervalo definido por los discos de Gershgorin. Debe tenerse en cuenta que si bien el algoritmo aquí descrito tiene una finalidad muy similar, el método y la estrategia seguidos difieren significativamente.

Por otra parte, en lo que respecta a la vertiente del cálculo numérico de valores propios esta tesis se basa principalmente en el artículo *Parallel computation of the eigenvalues of symmetric Toeplitz matrices through iterative methods* [13] donde se describe una aproximación al cálculo paralelo de valores propios en matrices Toeplitz simétricas mediante la versión *Shift and Invert 2-way* del algoritmo de Lanczos. Este trabajo toma la casi totalidad de su contenido teórico de este artículo, centrándose más en la adaptación del mismo para su ejecución viable y eficiente en la GPU. Todos los detalles de esta adaptación se describen adecuadamente en la sección 3.1.

En un paso previo hacia el artículo anterior cabe destacar [14], donde de manera similar aunque algo más simple se propone una aproximación de 2 vías para la extracción simultánea de pares de valores propios desde el punto de vista del procesamiento de señal, campo en el que suelen aparecer este tipo de matrices. Este artículo además no se centra en la extracción de todos los valores propios de la matriz sino del más pequeño de éstos. Por ello se gasta la matriz inversa de forma implícita mediante la resolución de sistemas Toeplitz, utilizando para ello el algoritmo de Levinson. Debe tenerse en cuenta que este algoritmo, de coste $\mathcal{O}(n^2)$ al aprovechar la estructura de la matriz, fue diseñado para trabajar con matrices Toeplitz simétricas definidas positivas pudiendo introducir serios errores numéricos si no se cumple esta condición.

También existe un trabajo previo [1] sobre el que se basa [13], el cual se centra en la resolución de sistemas Toeplitz simétricos. Este artículo presenta una aproximación paralela a la resolución del sistema basada en la descomposición LDL^T de matrices de

Cauchy, reduciendo la talla del problema a la mitad con un simple preproceso basado en la transformada discreta del seno normalizada de la matriz. Hecho esto, la obtención de la solución de cualquier sistema para la matriz dada se limita a resolver dos sistemas triangulares. Este método mejora el propuesto en [14] al eliminar el requisito de usar matrices definidas positivas y al reducir la talla del problema con el uso de las matrices de Cauchy. Como se verá posteriormente, ésta es una operación crucial para el proceso propuesto de extracción de valores propios, aunque el método propuesto por este artículo conlleva unos costes de memoria que pueden resultar excesivos para su implementación en algunos hardware gráficos.

Adicionalmente, existen otros artículos relacionados con la resolución de sistemas Toeplitz que pueden resultar especialmente interesantes de cara a mejorar algunos aspectos de esta tesis. Resultan especialmente mencionables artículos como [3], que propone una extensión del algoritmo de Levinson anteriormente mencionado para extender su correcta funcionalidad a matrices Toeplitz simétricas que no requieran ser definidas positivas. Para ello se utilizan técnicas que permitan identificar iteraciones del algoritmo que vayan a presentar inestabilidad numérica, evitando éstas mediante el uso de eliminación Gaussiana. También se propone una aproximación al número de condición de la matriz.

Otro artículo similar y digno de mención es [7] donde se sugieren mejoras a algunas de las propuestas de [3] proponiendo una nueva variación del algoritmo de Levinson con look-ahead para solventar los problemas de inestabilidad numérica y reduciendo su coste asintótico a $\mathcal{O}(n \log^2 n)$. Este tipo de algoritmos de resolución de sistemas Toeplitz son conocidos en la literatura como *asymptotically superfast algorithms*.

Así pues, tal y como se describirá en las secciones 2.1 y 3.1 la rápida resolución de sistemas Toeplitz resulta esencial para la eficiencia del algoritmo, proporcionando estos artículos interesantes alternativas a algunos métodos aquí implementados y dejando la puerta abierta a futuras mejoras o posibles nuevas líneas de investigación.

1.2. Nuevos usos del hardware gráfico

De entre los usos habituales de los computadores modernos uno de los más caros computacionalmente hablando son los gráficos 3D presentes en los videojuegos de hoy en día. Estos, generalmente con el objetivo de conseguir un mayor nivel de detalle y realismo, han ido creciendo en complejidad realizando multitud de cálculos geométricos y vectoriales de coma flotante hasta el punto de ejecutar pequeños programas independientes por cada uno de los píxeles que se muestran en pantalla.

Esta tremenda demanda computacional, imposible de llevar a cabo en tiempo real con la CPU, ha llevado al desarrollo de procesadores gráficos muy potentes pero a la vez muy especializados para la tarea a la que se encomendaban. Tan especializados que hasta hace tan sólo unos pocos años resultaba imposible y prácticamente impensable darles un uso diferente del que habían sido ideados. Para entender mejor la situación actual resulta necesario comprender, aún brevemente, la evolución del hardware gráfico así como los principales motivos que encarecieron este avance.

1.2.1. Evolución de las GPUs

Si bien las tarjetas gráficas existen desde hace bastante tiempo, el primer paso al que quisiera hacer referencia es la aparición no de las propias tarjetas gráficas sino de las *aceleradoras* gráficas. Éstas comenzaron a aparecer a finales de la década de los 90 y supuso toda una revolución al permitir el uso de hardware especialmente dedicado a acelerar algunas de las tareas más paralelizables y más costosas en aquel momento: la rasterización y el filtrado de texturas. Así pues estas primeras tarjetas gráficas eran especialistas en rasterizar triángulos 2D interpolando colores o aplicando texturas con interpolación bilineal. Esto permitía, por ejemplo, aplicar técnicas de degradados de color y sombras difuminadas que mejoraron notablemente la calidad gráfica de las escenas 3D del momento. Pero a pesar de todo, que no es poco, no servían para mucho más.

Unos años más tarde lo que en un principio fueron tarjetas aceleradoras gráficas, conectadas mediante un puente a la tarjeta gráfica y de éstas al monitor, acabaron integrándose en la nueva generación de procesadores gráficos. Estos pasaron de recibir información en forma de triángulos bidimensionales que rasterizar a integrar también el proceso de transformación 3D y proyección al plano de la imagen. Es decir, las nuevas tarjetas gráficas recibían ya directamente con datos 3D. Este paso permitió liberar a la CPU de la mayoría de cálculos geométricos y, además, utilizar procesadores dedicados con unidades SIMD encargadas de realizar estos cálculos vectoriales muy rápidamente.

No mucho después la versatilidad del hardware gráfico comenzó a crecer significativamente con la aparición de las primeras GPUs programables. Al principio éstas permitían tan sólo pequeños programas sin control de flujo y acceso limitado a los recursos que se ejecutarían para cada uno de los vértices de la geometría renderizada. Esto permitió el desarrollo y mejora de nuevas técnicas gráficas como por ejemplo el *Cel Shading* (modelos 3D con aspecto de dibujo animado). A estos programas gráficos que se ejecutaban en la GPU se les llamó *shaders* y empezaron a desarrollarse los primeros lenguajes para para programarlos. Estamos en la época de la versión 2.0 de OpenGL y la aparición de su lenguaje GLSL: *OpenGL Shading Language*.

No obstante una vez se dio el paso al hardware gráfico programable los avances no se limitaron a lo anterior. Los degradados de color y las sombras difuminadas producidas por la iluminación difusa, traídos los gráficos en tiempo real años antes con las primeras aceleradoras, resultaban ahora insuficientes. Se buscaban nuevos avances en técnicas de iluminación: avances que permitiesen modelar correctamente reflejos especulares de luz o simular rugosidad en superficies. Los algoritmos para lograr esto eran bien conocidos desde años atrás pero implicaban replicar a cada píxel los cálculos geométricos que antes se realizaban a nivel de vértice, incrementando los cálculos en varios órdenes de magnitud. Así pues, el hardware dio el siguiente paso con la incorporación de *fragment shaders*.

Este nuevo tipo de programa se ejecutaba por cada fragmento o píxel procesado por la tubería gráfica permitiendo aplicar así una amplísima novedad de nuevos algoritmos y efectos gráficos. Al igual que ocurrió con los *vertex shaders*, al principio los programas de píxel resultaban pequeños, simples y sin control de flujo, aunque podían acceder a los datos de textura almacenados en memoria gráfica. La situación no tardó en avanzar y en apenas un par de años todos los *shaders* fueron capaces de realizar saltos condicionales

así como de leer y escribir datos de textura. Esto nos trae ya a hardware gráfico relativamente reciente, como pueden ser las tarjetas GeForce serie 6 y 7.

Una vez las GPUs fueron programables y en éstas se pusieron ejecutar instrucciones con flujo dinámico así como leer y escribir datos de memoria comenzó a considerarse la posibilidad de pudieran utilizarse para otros fines más allá de la programación gráfica. Se disponía de una tremenda capacidad de cálculo paralelo orientado a operaciones vectoriales en coma flotante de simple precisión. Además la posterior aparición de texturas en coma flotante permitió dar el paso final encontrando una manera de representar datos y almacenar resultados. Había nacido la programación de propósito general en procesadores gráficos.

Al principio la utilización del hardware gráfico para otros fines distintos al original resultaba ser tremendamente tortuoso: se requerían profundos conocimientos de programación gráfica y en gran medida se dependía de extensiones de OpenGL no oficiales soportadas sólo por algunos fabricantes. Además, al estar el hardware diseñado para la ejecución de miles de instancias independientes del mismo programa en paralelo sin comunicación alguna entre éstas, las operaciones de reducción como puede ser la suma de elementos de un vector se tornaban notablemente complicadas y potencialmente ineficientes. Al tener que encapsular toda operación que se desease hacer en operaciones gráficas existentes fue necesario desarrollar unas nuevas técnicas aplicables a este entorno y, aún a pesar de sus dificultades y de las limitaciones del hardware, empezaron ya a lograrse los primeros speed-ups llamativos.

Poco a poco comenzaron a surgir artículos sobre todo tipo de aplicaciones de GPUs a problemas en los que nunca se había planteado su uso. De esta manera NVIDIA, uno de los principales fabricantes de hardware gráfico del mundo, anunció la creación de una nueva arquitectura soportada por su nuevo hardware específica para la programación gráfica de propósito general: CUDA. Bajo las siglas de *Compute Unified Device Architecture* CUDA presenta un nuevo modelo de programación donde toda referencia a la tubería gráfica, así como los antes indispensables conocimientos sobre programación gráfica han desaparecido. Adicionalmente CUDA proporciona nuevas funcionalidades no disponibles anteriormente como es la aparición de una pequeña memoria compartida que junto a ciertos mecanismos de sincronización permite la comunicación parcial entre las distintas instancias del mismo programa (hilos) que se están ejecutando.

Esta abstracción de todo el proceso fue tan importante que la propia NVIDIA creó plataformas específicas para computación de altas prestaciones: las NVIDIA Tesla. Estas plataformas se componen básicamente del mismo hardware que sus productos dedicados a acelerar gráficos 3D modernos, si bien con más memoria y algunas otras mejoras específicas. En cualquier caso, se trata de hardware gráfico que, paradójicamente, ha perdido su salida gráfica para centrarse en la computación de propósito general.

No obstante no es necesario adquirir uno de estos caros dispositivos especializados para la programación de propósito general en CUDA: también es posible utilizar las tarjetas gráficas habituales de NVIDIA para estos fines, y éstas tienen un precio irrisorio para la aceleración que ofrecen en algunos problemas en comparación con soluciones

basadas en clusters. Y así pues se llega a la situación actual, donde la GPUs prometen ser una útil y barata herramienta de computación de altas prestaciones o incluso de supercomputación dadas sus características, las cuales se describen con mayor detalle a continuación.

1.2.2. Características de las GPUs actuales

La arquitectura CUDA está soportada por todas las tarjetas gráficas de NVIDIA desde las GeForce Series 8 así como las NVIDIA Quadro FX, Quadro Plex, las versiones para ordenadores portátiles de todas éstas y las ya mencionadas NVIDIA Tesla. Puede consultarse una lista detallada de los dispositivos que soportan CUDA en la página oficial de NVIDIA.

A continuación se muestran las características oficiales de algunas de las últimas GPUs compatibles disponibles. Cabe destacar que aquellas con capacidades computacionales iguales o superiores a la versión 1.3 soportan aceleración hardware para la coma flotante de doble precisión así como modelos de programación menos restrictivos.

	GeForce 8800 GTX	GeForce 9800 GX2
Número de GPUs integradas	1	2
Núcleos de procesamiento	128	2x128
Frecuencia de los núcleos	1350 MHz	1500 MHz
Cantidad de memoria	768 MB	2x512 MB
Ancho de banda de memoria	86.4 GB/s	2x64 GB/s
Capacidad computacional de CUDA	Versión 1.0	Versión 1.1
Precio actual aproximado	\$100 $\sim$ \$125	\$320 $\sim$ \$375

Cuadro 1: características de GeForce 8800 GTX y GeForce 9800 GX2.

(a) GeForce 8800 GTX

(b) GeForce 9800 GX2

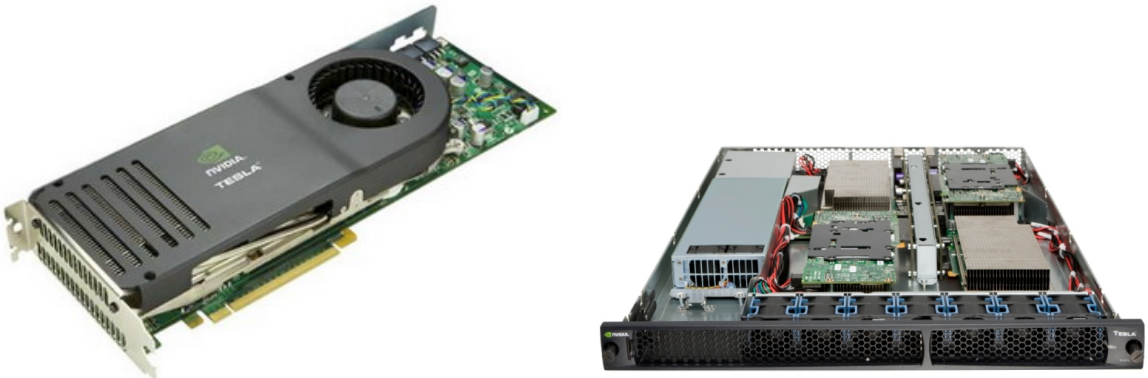
Figura 1: algunas tarjetas gráficas notables de las GeForce Series 8 y 9 (simple precisión).

	GeForce GTX 280	Quadro FX 5800
Número de GPUs integradas	1	1
Núcleos de procesamiento	240	240
Frecuencia de los núcleos	1296 MHz	(sin datos oficiales)
Cantidad de memoria	1 GB	4 GB
Ancho de banda de memoria	141.7 GB/s	102 GB/s
Capacidad computacional de CUDA	Versión 1.3	Versión 1.3
Precio actual aproximado	\$385 $\sim$ \$425	\$3500

Cuadro 2: características de GeForce GTX 280 y Quadro FX 5800.

(a) GeForce GTX 280 (b) Quadro FX 5800

Figura 2: últimas tarjetas gráficas de NVIDIA hasta la fecha (doble precisión).



(a) Tesla C1060 (b) Tesla S1070

Figura 3: novedades de NVIDIA Tesla 10 Series para computación de altas prestaciones.

	Tesla C1060	Tesla S1070
Número de GPUs integradas	1	4
Núcleos de procesamiento	240	4x240
Frecuencia de los núcleos	1296 MHz	1296 MHz
Cantidad de memoria	4 GB	4x4 GB
Ancho de banda de memoria	102 GB/s	4x102 GB/s
Prestaciones en simple precisión	933 GFlops	4.14 TFlops
Prestaciones en doble precisión	78 GFlops	345.6 GFlops
Capacidad computacional de CUDA	Versión 1.3	Versión 1.3

Cuadro 3: características de Tesla C1060 y Tesla S1070.

2. Contexto del problema

Dadas las circunstancias que caracterizan a esta tesis es necesario una adecuada contextualización de los contenidos en los que se basa. Por ello esta sección procede a analizar tanto la base teórica de los métodos numéricos empleados como la estructura detallada y las nociones más importantes de la arquitectura CUDA. Estos conocimientos servirán de base para concluir la sección proponiendo y justificando diversas modificaciones realizadas a los métodos originales descritos en [13], describiendo su funcionamiento e implementación en la sección siguiente.

2.1. Descripción del problema

Tal y como se indica en [13], el problema de los valores propios de matrices reales simétricas se define como:

$$\begin{gathered} Ax = \lambda x \\ \text{donde } A \in \mathbb{R}^{n\times n}, x \neq 0 \in \mathbb{C}^n, \lambda \in \mathbb{R} \end{gathered} \tag{1}$$

Al no existir una solución analítica general para $n >= 5$ a causa del teorema de imposibilidad de Abel, la aproximación al problema se realiza mediante métodos iterativos que tratan de aproximarse a la solución progresivamente en cada paso. No obstante la forma apropiada de resolver este problema depende principalmente de las circunstancias concretas del problema, tales como la estructura y propiedades de la matriz en cuestión. Del mismo modo la forma de resolverlo cambia en función de cuáles, cuántos y con qué precisión quieran extraerse estos valores, existiendo por ejemplo métodos notables para la obtención de únicamente el mayor o el menor de todos ellos.

En este caso el problema se orienta a la extracción de la totalidad de los valores propios de matrices Toeplitz simétricas reales. Este tipo de matrices, típicas de los problemas

de procesamiento de señal, se caracterizan por su peculiar estructura formada por n diagonales que repiten un mismo valor. Esta estructura queda plasmada en la figura 4.

$$\begin{pmatrix} t_1 & t_2 & t_3 & \dots & \dots & t_n \\ t_2 & t_1 & t_2 & \ddots & & \vdots \\ t_3 & t_2 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & t_2 & t_3 \\ \vdots & & \ddots & t_2 & t_1 & t_2 \\ t_n & \dots & \dots & t_3 & t_2 & t_1 \end{pmatrix}$$

Figura 4: estructura de una matriz Toeplitz simétrica.

Resulta fácilmente apreciable el hecho de que este tipo de matrices en realidad sólo contienen n valores diferentes. Esto lleva en la práctica a la representación de los datos usando un único vector $t \in \mathbb{R}^n$ reduciendo en un orden de magnitud el coste espacial.

2.1.1. Algoritmo *Shift and invert 2-way Lanczos*

Volviendo a la extracción de valores propios, [13] propone el proceso de intervalos independientes utilizando el método *Shift and invert* aplicado al algoritmo de Lanczos. Este método se basa en el cálculo los valores propios de $(A - \sigma I)^{-1}$ en lugar de A para tratar de obtener una rápida convergencia hacia los valores próximos al valor de *shift* σ, que normalmente estará centrado en el intervalo de interés. Estos valores propios guardan una relación directa con los de la matriz original, de forma que:

$$\begin{gathered} \lambda_A = \tfrac{1}{\lambda_\sigma} + \sigma \\ \text{con } Ax = \lambda_A x \quad \text{y} \quad (A - \sigma I)^{-1} x = \lambda_\sigma x \end{gathered} \tag{2}$$

No obstante el proceso de calcular $(A-\sigma I)^{-1}$ resulta costoso y numéricamente inadecuado, por lo que como suele ser habitual en los algoritmos numéricos el uso de la matriz inversa se sustituye por la resolución de un sistema, en este caso un sistema Toeplitz. De esta forma la rápida solución de este tipo de sistemas pasa a ser una pieza vital en la eficiencia del algoritmo.

Adicionalmente y con el fin de aprovechar algunas propiedades adicionales de las matrices Toeplitz se propone una segunda modificación al algoritmo de Lanczos original, la llamada versión de dos vías (*2-way*). Esta modificación aprovecha ciertas propiedades de simetría en la solución del sistema para generar simultáneamente un par de subespacios de Krylov: uno de base simétrica y una de base antisimétrica. Concretamente este método

se basa en la siguiente propiedad de los sistemas Toeplitz simétricos:

$$\begin{aligned}&\text{Sean: } T \in \mathbb{R}^{n\times n} \text{ una matriz Toeplitz simétrica}\\ &\qquad v, w \in \mathbb{R}^n \text{ tales que } Tv = w\\ &\qquad J_n \in \mathbb{R}^{n\times n} = (\delta_{i,n-i+1})_{i,j=1,\dots,n}\\ &\qquad v_s = 0{,}5(v + J_n v) \text{ simétrico}, \quad v_a = v - v_s \text{ antisimétrico}\end{aligned} \tag{3}$$

Entonces v_s resuelve el sistema $Tv_s = 0{,}5(w + J_n w)$ y v_a resuelve $Tv_a = w - w_s$

De esta forma la resolución de un único sistema $Tv = w$ se puede utilizar para incrementar simultáneamente el tamaño de dos subespacios en una misma iteración, uno basado en v_s y otro en v_a, duplicando así la eficiencia en la extracción de valores propios.

Así pues el algoritmo propuesto originalmente por [13] para la extracción de valores propios en un intervalo es el siguiente. Dada $T \in \mathbb{R}^{n\times n}$ Toeplitz simétrica y $\sigma \in \mathbb{R}$ el valor de *shift*:

Algoritmo 1 Shift and invert 2-way Lanczos

1: Sean $p_1 = J_n p_1 \neq 0$ y $q_1 = -J_n q_1 \neq 0$ vectores iniciales con $\|p_1\|_2 = \|q_1\|_2 = 1$
2: Sean $p_0 = q_0 = \vec{0}$ y $\beta_0 = \delta_0 = 0$
3: **for** $k = 1, 2, \dots,$ hasta convergencia **do**
4: $w = p_k + q_k$
5: Resolver el sistema $(T - \sigma I)\, v = w$
6: $v_s = 0{,}5 \cdot (v + J_n v)$ $\qquad$ $v_a = 0{,}5 \cdot (v - J_n v)$
7: $\alpha_k = v_s^T \cdot p_k$ $\qquad$ $\gamma_k = v_a^T \cdot q_k$
8: $v_s = v_s - \alpha_k p_k - \beta_{k-1} p_{k-1}$ $\qquad$ $v_a = v_a - \gamma_k q_k - \delta_{k-1} q_{k-1}$
9: Reortogonalización completa:
$v_s = v_s - p \cdot (p^T \cdot v_s)$ $\qquad$ $v_a = v_a - q \cdot (q^T \cdot v_a)$
10: $\beta_k = \|v_s\|_2$ $\qquad$ $\delta_k = \|v_a\|_2$
11: $p_{k+1} = v_s/\beta_k,$ $\qquad$ $q_{k+1} = v_a/\delta_k$
12: Obtener valores propios a partir de las matrices tridiagonales formadas por α, β (subespacio simétrico) y por γ, δ (subespacio antisimétrico).
13: Aplicar *shift and invert*: $\lambda_T = \frac{1}{\lambda_\sigma} + \sigma$
14: Comprobar convergencia de los valores propios extraídos λ_T.
15: Calcular vectores propios correspondientes utilizando las bases p y q.
16: **end for**

Cabe resaltar que el objetivo inmediato de este método es la obtención de los coeficientes del par de matrices tridiagonales a partir de las cuales se extraerán las aproximaciones a los valores y vectores propios de $(T - \sigma I)^{-1}$. Esta extracción puede llevarse a cabo mediante funciones bien conocidas implementadas en la clásica librería de cálculo numérico *LAPACK*.

2.1.2. Resolución del sistema transformando a matrices tipo Cauchy

Del algoritmo anterior también cabe resaltar la notable importancia del paso de resolución del sistema Toeplitz. Tal y como se ha ido adelantando en capítulos anteriores

la rápida resolución de este tipo de sistemas va a resultar crucial para la eficiencia del algoritmo. De esta manera, [13] se basa en su trabajo previo [1] y decide aplicar una aproximación basada en matrices tipo Cauchy para la resolución del sistema.

Concretamente, dada la transformación discreta del seno normalizada (DST-I, de *Discrete Sine Transform*) definida en forma matricial como

$$\mathscr{S} \in \mathbb{R}^{n\times n} = \frac{1}{\sqrt{\frac{n+1}{2}}} \cdot \left[\sin\left(\frac{ij\pi}{n+1}\right)\right]_{i,j=1,2,\ldots,n} \tag{4}$$

puede aplicarse para transformar el sistema Toeplitz $Tx = b$ en

$$\begin{gathered} C\bar{x} = \bar{b} \\ \text{donde } C = \mathscr{S}T\mathscr{S},\ \ \bar{x} = \mathscr{S}x,\ \ \bar{b} = \mathscr{S}b \end{gathered} \tag{5}$$

La matriz resultante $C = [c_{i,j}]_{i,j=1}^{n}$ es una matriz tipo Cauchy. Estas matrices destacan por la propiedad de que $c_{i,j} = 0$ si $i + j$ es impar. Es decir, cerca de la mitad de sus elementos son nulos. Esta propiedad puede explotarse definiendo una matriz de permutación P que agrupe primero las filas impares para después reunir las pares: $P(x_1, x_3, x_5, \ldots, x_2, x_4, x_6, \ldots)$.

De esta manera, al aplicar esta permutación dos veces sobre la matriz C se consigue expresar ésta como una matriz compuesta por dos bloques de orden $\lceil n/2 \rceil$ y $\lfloor n/2 \rfloor$ respectivamente:

$$PCP^T = \begin{pmatrix} C_0 & 0 \\ 0 & C_1 \end{pmatrix} \tag{6}$$

De esta manera el sistema original puede replantearse de la forma:

$$\begin{gathered} C_j\hat{x}_j = \hat{b}_j,\ \ j = 0, 1 \\ \text{con } \hat{x} = (\hat{x}_0^T\ \ \hat{x}_1^T)^T = P\mathscr{S}x \text{ y } \hat{b} = (\hat{b}_0^T\ \ \hat{b}_1^T)^T = P\mathscr{S}b \end{gathered} \tag{7}$$

Estos sistemas derivados de los bloques pueden resolverse eficientemente en $\mathcal{O}(n^2)$ utilizando la descomposición LDL^T debidamente adaptada a matrices tipo Cauchy. La descripción detallada de este algoritmo puede encontrarse en los apéndices de [13].

Así pues, la resolución propuesta del sistema original $Tx = b$ queda como se describe a continuación:

Algoritmo 2 Resolución de sistemas Toeplitz simétricos aplicando transformación a matrices tipo Cauchy

1: Calcular $C_0, C_1, \hat{b}_0, \hat{b}_1$ mediante la DST-I normalizada $\mathscr{S}$ y la permutación P.
2: Descomponer $C_0 = L_0D_0L_0^T$ y $C_1 = L_1D_1L_1^T$.
3: Resolver los sistemas los sistemas triangulares $L_0D_0L_0^T\hat{x}_0 = \hat{b}_0$ y $L_1D_1L_1^T\hat{x}_1 = \hat{b}_1$.
4: Obtener $x = \mathscr{S}P^T\left(\begin{smallmatrix}\hat{x}_0\\ \hat{x}_1\end{smallmatrix}\right)$.

Cabe resaltar que los dos primeros pasos pueden ser precalculados, pudiendo resolver cualquier sistema para una matriz T dada simplemente almacenando L_0 y L_1 y resolviendo un algunos sistemas triangulares. Otro detalle especialmente relevante es el hecho de que el número de valores positivos de D_0 y D_1 nos indica cuántos valores propios hay menores al σ utilizado. Esta información combinada con la de otros intervalos nos permite averiguar a priori el número de valores propios contenidos en cada uno, así como facilitar técnicas que traten de nivelar la cantidad de valores a extraer en cada intervalo.

De esta manera, el algoritmo general propuesto por [13] y sobre el que se basa el trabajo actual es el siguiente:

Algoritmo 3 Visión general del método de extracción (FSTW Lanczos)

1: Escoger el intervalo $[a, b]$ de extracción (pueden usarse discos de Gershgorin).
2: Dividir $[a, b]$ en subintervalos, a poder ser con el mismo número de valores propios.
3: **for** cada subintervalo, evaluado en paralelo **do**
4: Calcular un valor de *shift* σ, por ejemplo $\sigma = (a + b)/2$.
5: Descomponer la matriz $(T - \sigma I)$ en las matrices tipo Cauchy C_0 y C_1.
6: Obtener la descomposición LDL^T de C_0 y C_1 almacenando las matrices L_0 y L_1.
7: Aplicar *Shift and invert 2-way Lanczos* descrito en el algoritmo 1 para extraer los valores y vectores propios más próximos al valor de *shift* σ.
8: **end for**

Finalmente queda pendiente la selección apropiada de los intervalos de extracción paralelos. El artículo original menciona algunas sugerencias sobre el número máximo de valores propios en cada intervalo, así como técnicas para tratar de balancear la carga. No obstante en la implementación actual se utiliza una librería externa que se encarga de realizar estos cálculos por lo que no se profundizará más en ellos.

2.2. Introducción a CUDA

CUDA, siglas de *Compute Unified Device Architecture*, es un entorno de programación basado en C que permite programar algoritmos para su ejecución en procesadores gráficos. Para ello proporciona un compilador, el *nvcc*, que se encarga de procesar el código nativo de GPU dejando la compilación del código de CPU a otros compiladores como pueden ser el *gcc*. Así mismo también proporciona una serie de herramientas como profilers, debuggers para GPU, guías de ocupación de recursos y librerías adicionales para operaciones lineales y transformadas de Fourier (*CUBLAS* y *CUFFT*). Adicionalmente se proporciona un SDK repleto de ejemplos y aplicaciones de uso de todo tipo.

CUDA ha sido desarrollado por NVIDIA Corporation, famoso fabricante de hardware gráfico y prácticamente líder del mercado en los últimos años con la única competencia real de ATI Corporation, ahora parte del fabricante de procesadores AMD. Como consecuencia, tal y como se indicó en la sección 1.2.2 se requiere hardware gráfico de esta compañía para poder utilizar CUDA, en concreto las tarjetas gráficas GeForce 8, 9 y superiores, NVIDIA Quadro FX y Quadro Plex y el hardware de computación de altas prestaciones NVIDIA Tesla.

Al parecer otras empresas están viendo las posibilidades de negocio que CUDA supone y están tratando de desarrollar tecnologías similares que posiblemente veamos en un futuro. No obstante cabe destacar que si bien CUDA ya ha logrado una muy amplia aceptación en comparación con los intentos previos de computación de propósito general en GPUs, NVIDIA es el fabricante por excelencia de GPUs con altísimas prestaciones sólo seguido por ATI. Sus hardware ya se han extendido a los ordenadores del usuario medio en todo el mundo, tanto en ordenadores de sobremesa como en portátiles. De esta forma una notable cantidad de usuarios posee ya un hardware capaz de ejecutar programas en CUDA aunque no sean conscientes de ello, y este número no hará más que crecer en los próximos meses.

Volviendo a los detalles del modelo computación de CUDA, dado que muchos lectores de esta tesis puede que no estén familiarizados con él se procederá a explicar sus principales detalles: el modelo de ejecución, la jerarquía de memorias y los patrones de acceso óptimo a éstas.

2.2.1. Modelo de ejecución

CUDA propone un modelo de ejecución basada en *kernels*, programas asíncronos de GPU que se lanzan desde la CPU. Internamente éstos ejecutan una malla de bloques de hilos de igual tamaño que tienen capacidad de comunicarse y sincronizarse. A continuación se mostrarán algunos esquemas explicativos y se entrará en detalle en estas estructuras.

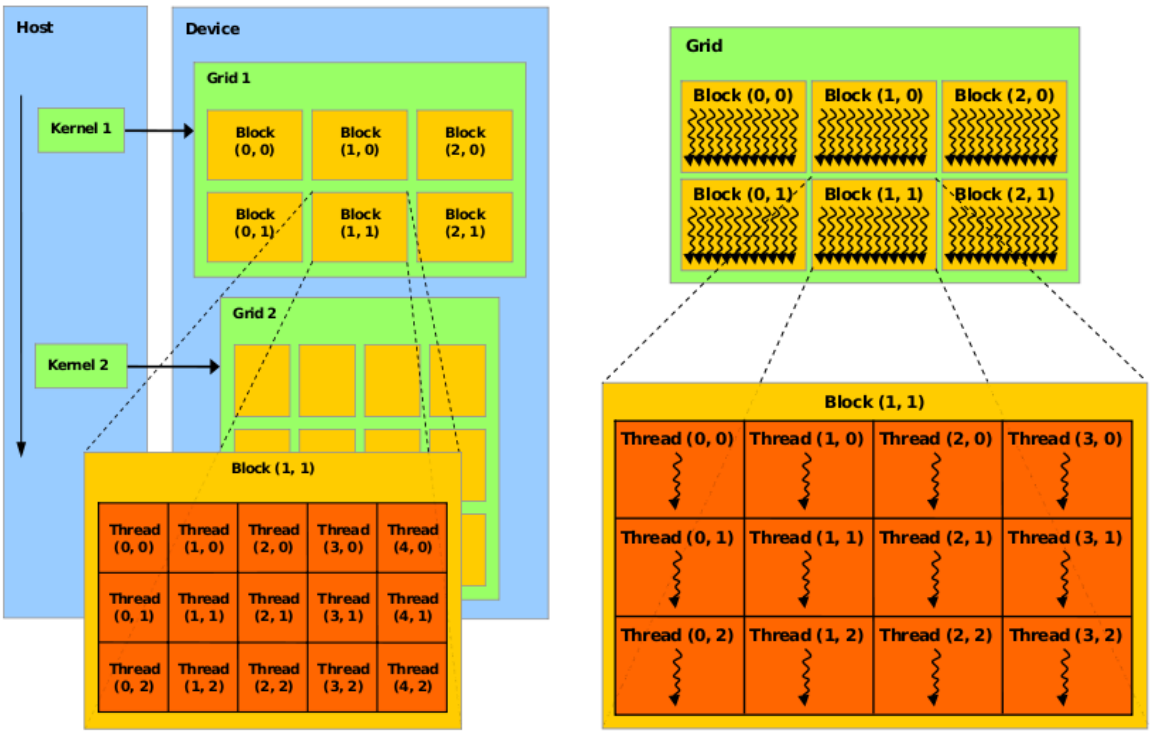


Figura 5: esquema de ejecución de kernels y bloques de hilos en CUDA.

Los principales elementos del modelo de ejecución de CUDA son los siguientes:

- **Kernels**: inician un proceso de ejecución en la GPU y su coste de lanzamiento es prácticamente despreciable. Se ejecutan en paralelo respecto a la CPU, que puede esperar a que se complete su ejecución si lo desea, pero sólo puede lanzarse un *kernel* por cada dispositivo (GPU) a la vez. Dicho de otra manera, los *kernels* son asíncronos con respecto a la CPU pero se ejecutan secuencialmente entre sí.
- **Malla de bloques**: cada *kernel* se lanza sobre una malla unidimensional o bidimensional de bloques de hilos. Por definición estos bloques tienen todos las mismas dimensiones y son independientes entre sí y carecen de un orden de ejecución definido. Del mismo modo los distintos bloques de la malla no pueden comunicarse entre sí ni sincronizarse dentro de una misma ejecución de un *kernel*. Este concepto permite la ejecución paralela ya no de varios hilos como se verá a continuación sino de múltiples bloques por diferentes multiprocesadores que forman parte del hardware gráfico.
- **Bloques de hilos**: constituyen el nivel mínimo en el que varios hilos pueden realizar tareas de comunicación y sincronización. Un bloque de hilos se compone de hasta 512 hilos independientes que ejecutan un mismo programa organizados estructuralmente en una, dos o tres dimensiones. Tal y como se explicará con más detalle en la sección siguiente cada bloque de hilos posee una pequeña memoria compartida de muy rápido acceso que puede usarse para tareas de comunicación o alineación de memoria. También disponen de sincronización tipo barrera mediante la función específica de CUDA *__syncthreads()*.

La estructura de ejecución utilizada en el sentido de dimensiones de malla de bloques y de cada bloque de hilos puede variar en cada *kernel* y se define en la propia llamada de ejecución de éste. Esta información así como los correspondientes índices de bloque e hilo se encuentran disponibles en tiempo de ejecución en las siguientes variables predefinidas:

- **threadIdx**: índices de hilo dentro del bloque actual.
- **blockIdx**: índices de hilo dentro de la malla de bloques.
- **blockDim**: dimensiones de bloque en la ejecución actual, en número de hilos.
- **gridDim**: dimensiones de la malla de bloques de la ejecución actual, en bloques.

Todas estas variables son estructuras que con tienen los elementos x, y, z para referirse a las diferentes dimensiones en las que pueden estructurarse. Además, los nombres de estas variables quedan reservados, del mismo modo tampoco es posible asignarles manualmente ningún valor.

Adicionalmente existe una última variable predefinida llamada *warpSize* donde se define número de hilos concurrentes físicamente por multiprocesador. Ésta resulta especialmente relevante para temas de acceso óptimo a memoria tal y como se verá en la sección 2.2.3.

Por otra parte, la forma de programar sobre esta peculiar estructura es más sencilla de lo que en un principio podría parecer. CUDA introduce nuevas palabras clave que permiten indicar dónde se ejecutarán, si en CPU o GPU, algunas funciones del código. Para ello basta con añadirlas al principio de la declaración de las funciones de C sobre las que se apliquen:

- **__global__**: determina que la función es el punto de entrada de un *kernel*. Esta función se ejecutará en GPU y sólo podrá devolver *void*.
- **__device__**: determina que la función se ejecutará en la GPU. Ésta se convertirá en una función *inline* para el *kernel* (función *__global__*) que la invoque.
- **__host__**: determina que la función se ejecutará en la CPU. Este valor se asume por defecto y puede omitirse sin problemas.

Adicionalmente se aplican algunas fuertes restricciones generales a las funciones ejecutadas en la GPU (definidas como *__global__* y *__device__*):

- Ausencia total de recursividad: toda llamada a función se procesa como si de funciones *inline* se tratase. Puede conseguirse simulando pilas en memoria de GPU.
- No se puede llamar a funciones *__global__* desde la GPU: sólo sirven como puntos de invocación de *kernels*.
- No se puede llamar a funciones *__host__* ya que pertenecen a la CPU. Esta restricción se relaja si se trabaja en modo emulación, donde las funciones de GPU son ejecutadas por la CPU (opción del compilador para depuración).
- No se soporta el uso de variables estáticas.
- No se soportan funciones con número de parámetros variable.
- No se puede reservar memoria dinámica, excepto compartida (se verá con más detalle a continuación en la sección 2.2.2).

Finalmente, para invocar un *kernel* desde la CPU se utiliza la siguiente sintaxis:

```
dim3 grid(ancho, alto);
dim3 bloque(ancho, alto, largo);
función <<< grid, bloque, smem (opcional), stream (opcional) >>> (parámetros...);
```

El parámetro *smem* de memoria compartida se detallará en la sección 2.2.2. El parámetro *stream* en cambio se utiliza en diversos mecanismos de sincronización ofrecidos por CUDA y no se analizará en mayor detalle. Para más detalles consultar la guía oficial de programación en CUDA [11].

Así pues y a modo de recapitulación, CUDA permite indicar fácilmente en programas en código C qué funciones deben de ejecutarse en el procesador gráfico. Para ello basta con añadir unas pocas palabras clave al inicio de la declaración de las mismas. Hecho esto, se invocan desde de la CPU de forma asíncrona definiendo las dimensiones de la malla de bloques y de cada uno de éstos siguiendo la sintaxis específica para ello.

A continuación se muestra un ejemplo meramente ilustrativo de cómo se sumarían los contenidos de un par de matrices idénticas en C y usando CUDA:

Versión CPU

```
void add_matrix(float *a, float *b,
float *c, int n) {
   int i, j, idx;
   for(j=0; j<n; j++) {
      for(i=0; i<n; i++) {
         idx = j * n + i;
         c[idx] = a[idx] + b[idx];
      }
   }
}

void main() {
   ...
   add_matrix(a, b, c, n);
}
```

Versión CUDA

```
__global__ void add_matrix(float *a,
float *b, float *c, int n) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   if(i < n && j < n) {
      int idx = j * n + i;
      c[idx] = a[idx] + b[idx];
   }
}

void main() {
   ...
   dim3 block(blockSize, blockSize);
   dim3 grid(n / block.x, n / block.y);
   add_matrix <<< grid, block >>> (ga, gb, gc, n);
}
```

2.2.2. Jerarquía de memorias

A diferencia de los entornos tradicionales de programación, CUDA posee una jerarquía de memoria de distintos tipos, capacidades, velocidad de acceso y otras propiedades. El adecuado uso de estas memorias es un aspecto crucial para conseguir un buen rendimiento en las aplicaciones desarrolladas. Es por tanto necesario analizar y describir detalladamente las distintas memorias que CUDA pone a disposición del programador.

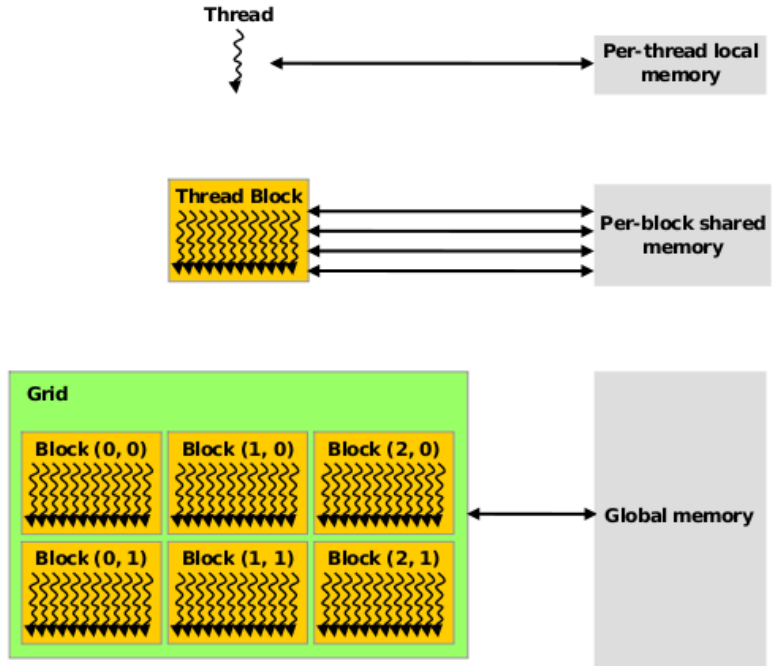


Figura 6: niveles de memoria de CUDA desde el punto de vista de la ejecución.

Las distintas memorias de la jerarquía de CUDA son las siguientes:

- **Memoria global**: la memoria principal del dispositivo, del tamaño que éste especifique. Todos los hilos de cualquier parte del grid de ejecución pueden acceder a esta memoria. En contra de lo que en un principio pudiera pensarse esta memoria carece de caché, por lo que los accesos a la misma deben seguir patrones muy concretos para resultar verdaderamente eficientes. Estos patrones de acceso se analizarán en profundidad en la sección 2.2.3, si bien las restricciones aplicadas se relajan notablemente para aquellos dispositivos que soporten capacidades computacionales de CUDA versión 1.2 o superiores.

  De forma estimada, un acceso a memoria global puede llevar unos 600 ciclos de reloj convirtiéndose fácilmente en el cuello de botella de las aplicaciones en CUDA.

- **Memoria compartida**: es una pequeña memoria *on-chip* de tan sólo 16 KB a la cual tienen acceso todos los hilos de un mismo bloque, permitiendo junto a los métodos de sincronismo la comunicación entre éstos. Esta memoria se caracteriza por ser extremadamente rápida, hasta el punto de alcanzar el mismo coste de acceso que un registro del multiprocesador. No obstante para ello deben respetarse algunas consideraciones de acceso o de lo contrario algunas peticiones pueden ser serializadas. Estos detalles se estudiarán en la sección 2.2.4.

- **Memoria de texturas**: se trata de un espacio de memoria dedicado al acceso a texturas, arrays bidimensionales originalmente destinados a contener información en forma de píxeles. Hoy en día las texturas también pueden almacenar información en coma flotante y utilizarse como espacio de almacenamiento más allá de sus fines en la programación gráfica.

  Esta memoria tiene algunas propiedades que la diferencian de la memoria global. En primer lugar posee una caché especializada en aprovechar la localidad de acceso 2D. También permite configurar la forma de acceso a sus datos: utilizando coordenadas absolutas o normalizadas, repitiendo el último valor al salirse de sus límites, actuando como tablas circulares, etc. Del mismo modo también permite acceder a sus datos realizando automáticamente operaciones de vecino más próximo e interpolación bilineal con respecto a sus contenidos.

  Todas estas operaciones provienen de la aceleración gráfica al procesamiento de texturas. Además se permite así una estrecha colaboración entre CUDA y OpenGL o Direct3D para acelerar cálculos de imagen.

- **Memoria constante**: es una pequeña memoria de solo lectura de 64 KB dedicada a almacenar datos constantes que sirvan de apoyo al programa en ejecución. Es accesible por cualquier hilo de cualquier bloque. Posee una caché que, en caso de acierto, reduce el tiempo de acceso al de un registro en caso de que todos los hilos accedan al mismo dato, escalándose linealmente en caso contrario. En caso de fallo de caché tiene el coste de un acceso a memoria global.

- **Memoria local**: se trata de una memoria de uso interno que se usa de forma auxiliar en caso de necesitar almacenar temporalmente registros del multiprocesador. Sólo se usa en caso de ser estrictamente necesario y al igual que la memoria global carece de caché. Acceder a la misma resulta tan caro como acceder a memoria global, por lo que debe tenerse cuidado de no hacer uso de ella. Esto puede comprobarse observando los resultados ofrecidos por diversas opciones del compilador *nvcc*. Esta memoria por definición no sufre los problemas de patrones de acceso que atañen a la memoria global.

De estas memorias las más importantes y de uso más frecuente son, con diferencia, la memoria global y la memoria compartida. Esta última jugará un papel vital en los métodos de alineación óptimos para un acceso eficiente a la memoria global tal y como se verá en la sección 2.2.3.

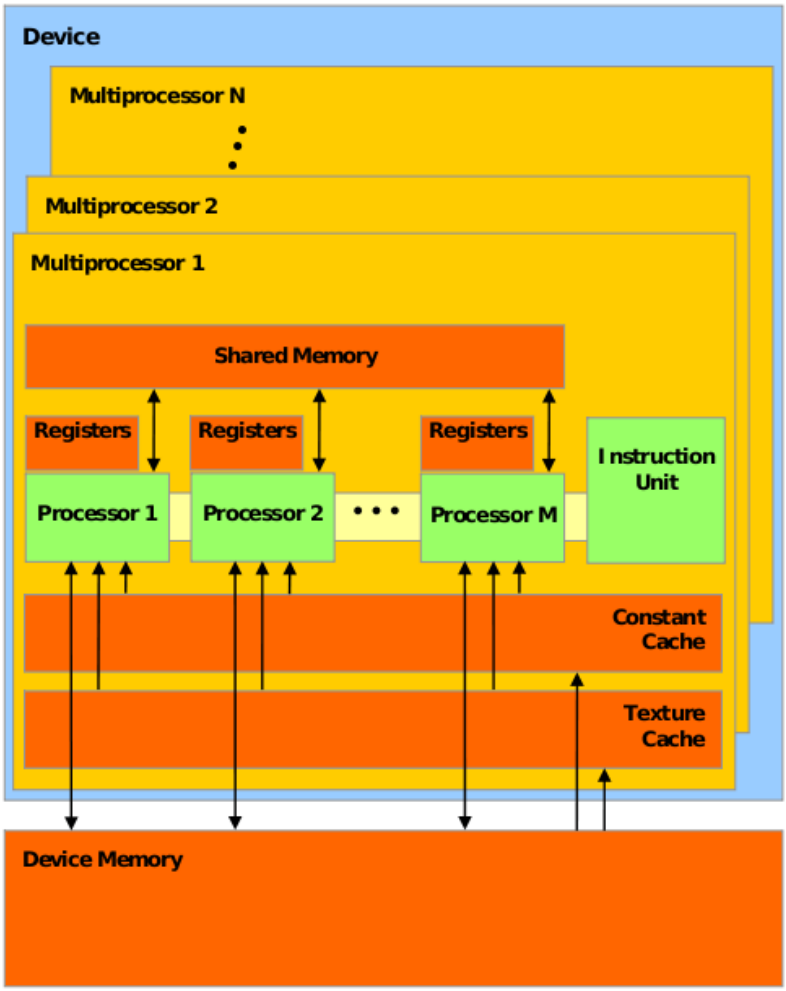


Figura 7: esquema de la arquitectura de las distintas memorias de CUDA.

Desde el punto de vista de la programación, CUDA define una serie de palabras clave para poder referirnos fácilmente a cada una de estas memorias. Para usarlas basta con incluirlas delante de la declaración de la variable sobre la que se apliquen, de forma similar a como ya se hacía con las funciones en la sección 2.2.1.

- **__device__**: define una variable que reside en memoria global. Tiene el tiempo de vida de la aplicación y es accesible desde cualquier hilo del grid así como desde la CPU utilizando las funciones del runtime de CUDA. Se asume por defecto en código de GPU y no es necesario especificarla.

- **__shared__**: define una variable que reside en memoria compartida. Tiene el tiempo de vida de un bloque de hilos y es accesible por todos los hilos pertenecientes a éste. Estas variables no pueden inicializarse en su declaración, al ser implícitamente estáticas y compartidas.

  Es importante resaltar que sólo se garantiza que los cambios realizados sean visibles por otros hilos después de una llamada a la función de sincronización *__syncthreads()*. Del mismo modo a no ser que se declare como volátil (palabra clave *volatile*) el compilador es libre de optimizar las lecturas y escrituras siempre que se cumpla la condición anterior.

  Adicionalmente es posible declarar un array de memoria compartida de forma predefinida en el momento de ejecución del *kernel*. Esta opción permite la posibilidad de manejar datos distribuidos dinámicamente dentro de este espacio de memoria compartida y simular así el soporte de memoria dinámica en *kernels*. Para hacer uso de él debe indicarse el tamaño en bytes de la memoria a reservar en el tercer parámetro de la invocación del *kernel* (ver sintaxis de invocación de *kernels* en la sección 2.2.1 para más detalles) y posteriormente definir un array como *extern __shared__* en la GPU del tipo que consideremos apropiado.

- **__constant__**: define una variable que reside en memoria constante. Ha de ser de sólo lectura y no puede definirse desde la GPU, sino que ha de hacerse desde la CPU mediante funciones específicas de CUDA . Las variables de este tipo son implícitamente estáticas, tienen el tiempo de vida de la aplicación y son accesibles desde cualquier hilo de la malla de ejecución.

2.2.3. Patrones óptimos de acceso a memoria global

Tal y como se ha descrito en la sección anterior las memorias global y compartida poseen ciertos patrones y criterios de acceso que deben respetarse para conseguir un acceso eficiente a las mismas, tanto de lectura como de escritura. En esta sección se exponen los criterios que conciernen al acceso óptimo a la memoria global mostrando varios esquemas explicativos de la guía oficial de programación [11] que ayudarán a comprender numerosas decisiones de diseño en la implementación actual del algoritmo.

El acceso óptimo a la memoria global de CUDA es probablemente el origen de la mayoría de quebraderos de cabeza a la hora de programar usando CUDA debido a lo restrictivo y a veces antinatural que resulta para muchos algoritmos. Por suerte los dispositivos que implementan la versión 1.2 de capacidades computacionales de CUDA relajan la gran mayoría de estas restricciones haciendo la programación en CUDA notablemente más fácil sin perder su eficiencia. Entre estos dispositivos se incluyen todos aquellos que integran soporte para coma flotante de doble precisión.

No obstante y puesto que el proyecto fue desarrollado utilizando una tarjeta GeForce 9800 GX2 de 2 GPUs con capacidad computacional 1.1 a continuación se expondrán y detallarán todas estas restricciones y detalles de las que se hablaba en el párrafo anterior.

Básicamente, CUDA define dos tipos de acceso a memoria global: el óptimamente alineado (de ahora en adelante *coalesced*) y los que no cumplen estos criterios (*uncoalesced*). La diferencia entre un tipo de acceso y otro es crítica: los accesos tipo *coalesced* acceden a los datos de memoria en paralelo realizando una única transacción mientras que los de tipo *uncoalesced* dividen los accesos a memoria hasta en 16 transacciones secuenciales diferentes. Teniendo en cuenta que un acceso a memoria global puede tardar aproximadamente 600 ciclos de reloj, el replanteamiento de todo acceso a memoria global para que cumpla las condiciones de los accesos *coalesced* se convierte en una absoluta necesidad.

Cabe volver a llamar la atención sobre una de las variables predefinidas mencionadas en la sección 2.2.1, la variable *warpSize*. Ésta define el número de hilos que se ejecutan físicamente de forma concurrente en un multiprocesador y tiene especial importancia en los accesos a memoria. Concretamente es especialmente relevante la definición de *half-warp*, que corresponde en la práctica a una agrupación de 16 hilos de índices consecutivos ya que el valor de *warpSize* es 32 para todas las versiones actuales de CUDA. En caso de bloques de hilos de más de una dimensión, los índices de éstos crecen desde el (0,0,0) en la esquina superior izquierda en orden de anchura, altura y profundidad respectivamente.

Así pues, un acceso a memoria se considera *coalesced* en dispositivos con capacidad computacional 1.1 si cumple los siguientes requisitos. Para todos los hilos de un *half-warp*:

- Todos los hilos deben acceder a:

  Palabras de 32 bits, generando una transacción de 64 bytes (las más rápidas).

  Palabras de 64 bits, generando una transacción de 128 bytes.

  Palabras de 128 bits, generando dos transacciones de 128 bytes.

- Las 16 palabras deben estar en el mismo segmento de memoria de tamaño igual al de la memoria transferida. Es decir, si se leen 16 palabras de 32 bits, éstas deben comenzar en una dirección de memoria múltiplo de $16 \times 4 = 64$ bytes.

- Los hilos han de acceder a posiciones consecutivas de memoria según su índice: el k-ésimo hilo del *half-warp* ha de acceder a la k-ésima palabra. No obstante no se requiere la participación de los 16 hilos en el acceso a memoria.

Sincronizar adecuadamente el trabajo de los *half-warps* en los accesos a memoria es la clave para conseguir utilizar eficiente la memoria global. No obstante no siempre es ni mucho menos viable o adecuado para muchos algoritmos, por ejemplo cuando se quieran leer posiciones desplazadas en un vector u obtener sus datos en orden inverso.

Para este tipo de problemas la solución habitual es leer datos de forma síncrona y con accesos de tipo *coalesced* a memoria compartida para después trabajar leyendo directamente de ésta en cualquier posición. En algunos casos más enrevesados también puede surgir la necesidad de mover algunos datos dentro de la propia memoria compartida

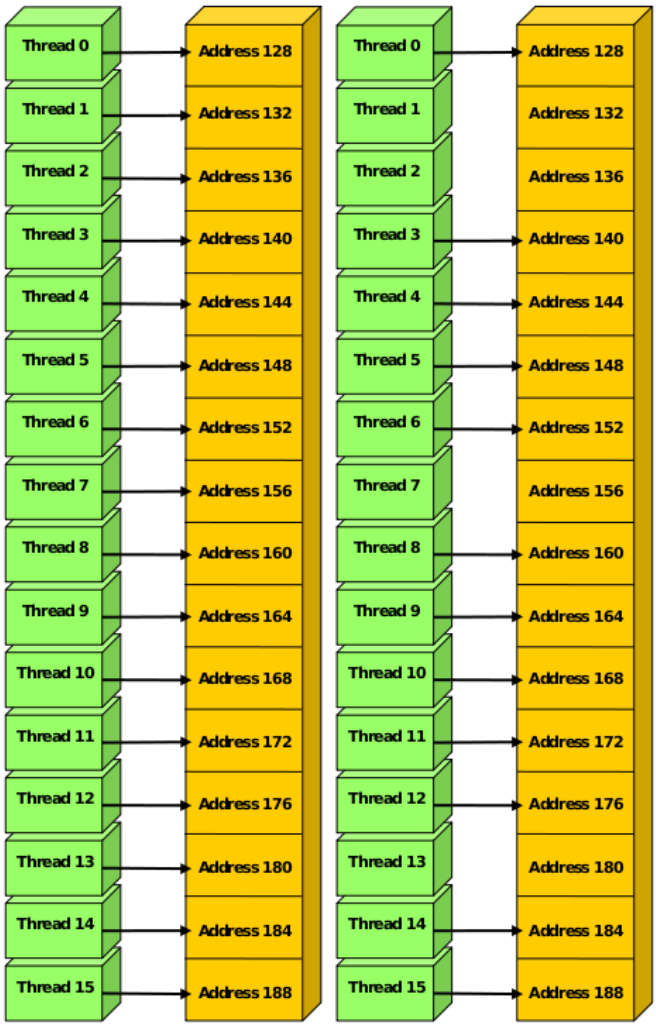


Figura 8: Ejemplos de accesos *coalesced* a memoria global.

con el fin de alinearlos con los $k-$ésimos hilos del *half-warp*. En este punto cabe tener en cuenta los criterios de acceso óptimo a memoria compartida que se describirán en la sección 2.2.4, si bien éstos son más simples y acarrean consecuencias menos significativas.

Así pues a la hora de diseñar los algoritmos a implementar en CUDA, al menos en los diseñados para dispositivos con capacidad computacional 1.0 ó 1.1, será necesario tener siempre en cuenta la minimización del número de acceso a memoria y, en la medida en que sea posible, expresarlos mediante accesos tipo *coalesced* apoyándose en espacios auxiliares en memoria compartida.

Para ayudar al programador en la tarea de identificar los indeseados accesos de tipo *uncoalesced* NVIDIA ha puesto a disposición del público su programa *CUDA Visual Profiler*, disponible para los sistemas operativos Linux, Windows y Mac OS, el cual realiza un exhaustivo análisis de la ejecución de programas en la GPU. Este programa reporta entre otras cosas información detallada del tiempo consumido por los distintos *kernels* así como el número de accesos a memoria que realiza y de qué tipo son. De esta forma es posible localizar fácilmente qué funciones están realizando accesos de tipo *uncoalesced* y cuántas transferencias se están realizando de forma potencialmente incorrecta.

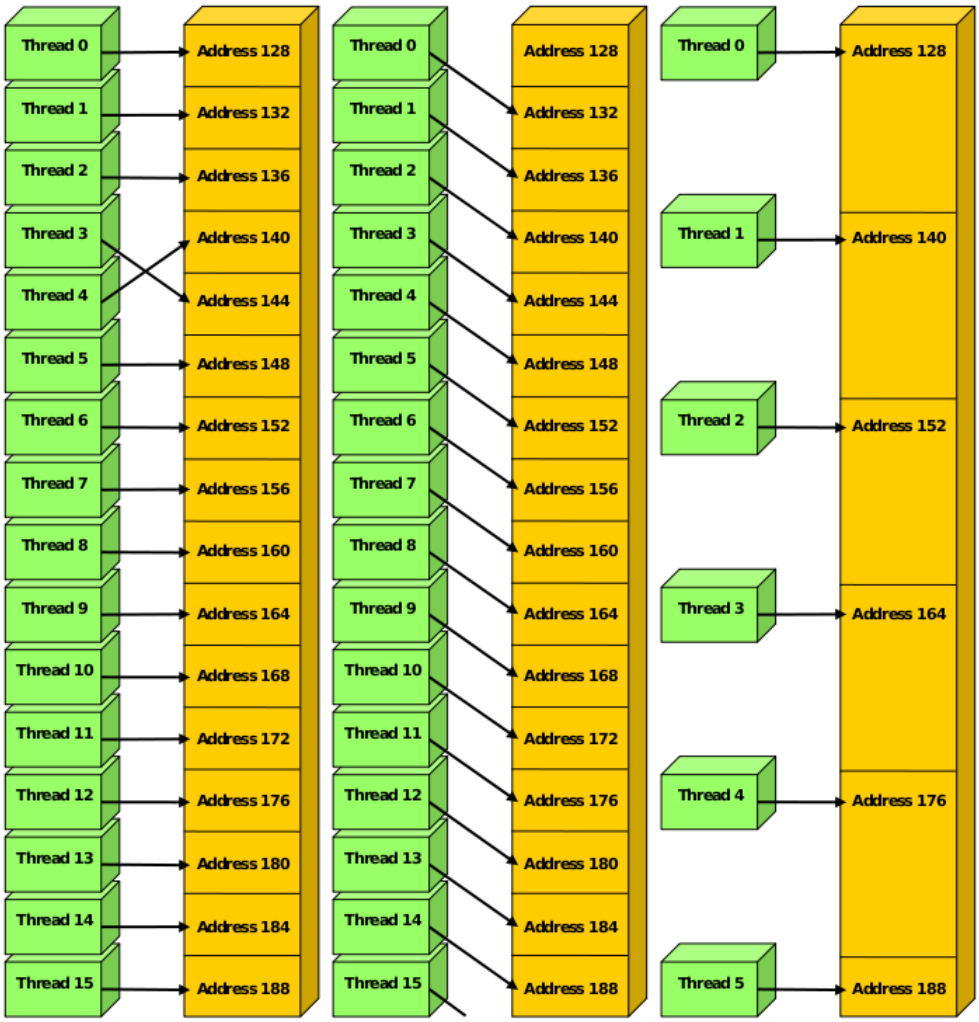


Figura 9: Ejemplos de accesos *uncoalesced* a memoria global.

2.2.4. Patrones óptimos de acceso a memoria compartida

Según la especificación oficial el acceso a memoria compartida puede ser tan rápido como el acceso a un registro del multiprocesador, siempre y cuando sigamos los criterios adecuados. Por suerte y a diferencia de los patrones descritos en la sección anterior, el acceso eficiente a memoria compartida resulta bastante más simple y sus consecuencias aunque existentes son notablemente menos serias.

En este caso no se habla de tipos de acceso a memoria sino de la existencia o no de conflictos de bancos de memoria. La memoria compartida se encuentra dividida en 16 bancos de memoria independientes a los que se puede acceder simultáneamente. Estos bancos se encuentran organizados de manera que palabras sucesivas de 32 bits son asignadas a bancos consecutivos, pudiendo servir éstas en dos ciclos de reloj. De esta manera para lograr un acceso óptimo a la memoria compartida basta con asegurarse de que cada hilo en un *half-warp* accede a bancos distintos de memoria a la vez sin importar su orden. En caso de que esto no sea así los accesos múltiples a un mismo banco se procesarán en serie de forma independiente al resto.

Adicionalmente la memoria compartida ofrece también la posibilidad de un acceso óptimo por broadcasting: si varios hilos de un *half-warp* acceden simultáneamente a un mismo valor de 32 bits en memoria compartida éste será leído por todos ellos en un único acceso sin causar conflicto alguno. Esta opción puede combinarse con anterior permitiendo broadcasting a algunos de los hilos mientras el resto accede a otros bancos diferentes de forma libre de conflictos.

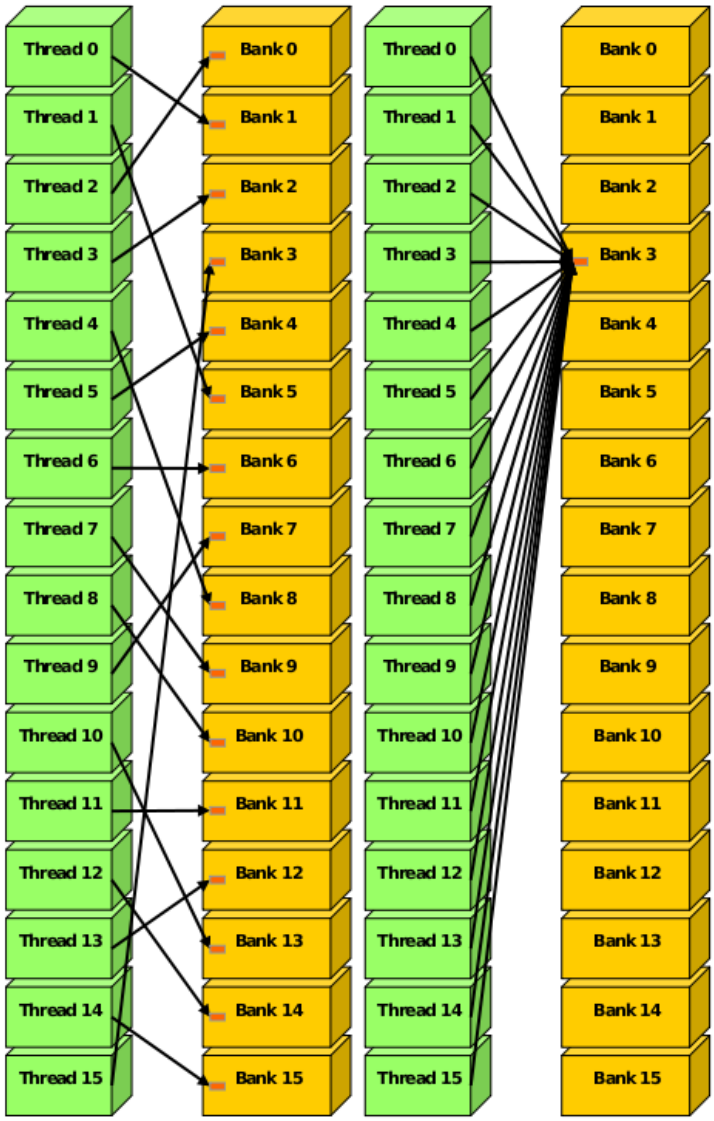


Figura 10: Ejemplos de accesos sin conflictos a memoria compartida.

2.2.5. Limitaciones en la ejecución y técnicas de optimización

Una vez vistos los principios y la estructura tanto del modelo de CUDA como de su jerarquía de memorias deben analizarse otros aspectos más concretos que afectan a la implementación de programas. El objetivo de esta sección no es otro que hacer hincapié en algunas de las limitaciones todavía no expuestas así como algunos consejos de cara a la creación de programas eficientes.

Existe una importante restricción todavía por mencionar que está relacionada con el número de hilos lanzados por bloque y los registros del multiprocesador. En la práctica cada multiprocesador cuenta con un número limitado de registros para servir a todos los hilos lanzados concurrentemente, y en caso de demandarse más de los disponibles la ejecución de los *kernels* afectados funcionará de forma imprevisible o fallará su lanzamiento. Debe tenerse especial cautela con esta situación ya que puede actuar de forma silenciosa y confundir completamente al programador.

El número de registros disponibles por multiprocesador varía según la versión de capacidad computacional de CUDA. Actualmente se define como 8192 registros para las versiones 1.0 y 1.1, duplicándose a 16384 a partir de la versión 1.2. De nuevo, el hardware utilizado para el desarrollo de este proyecto se ve limitado permitiendo un máximo de 8192 registros o lo que es lo mismo, 16 registros si se lanzan 512 hilos por bloque ó 32 registros si se lanzan 256.

Existe una forma principal de solucionar este problema, que en un principio escapa del control del programador. El compilador *nvcc* proporciona la opción *–maxrregcount* que define el número máximo de registros por hilo a utilizarse. De esta forma resultaría posible por ejemplo emplear *–maxrregcount 16* y lanzar 512 hilos sin mayor temor a los problemas comentados anteriormente.

No obstante si bien el compilador trata de ajustarse a las restricciones impuestas en caso de no poder cumplirlas hará uso automáticamente de la memoria local, anteriormente descrita en la sección 2.2.2. Tal y como se indicó, esta memoria carece de caché y su acceso es tan costoso como los accesos a memoria global, si bien por definición son todos de tipo *coalesced* al leer elementos individuales.

Por suerte estas situaciones son también fácilmente detectables. Existe otra opción del compilador, la opción *-cubin*, que permite procesar únicamente el código a ejecutar en la GPU generando un archivo de extensión *.cubin* con el código máquina de cada uno de los *kernels*, los datos de memoria constante e información sobre las secciones de datos, requisitos de memoria e instrucciones. Para cada *kernel* bajo la sección *code* del archivo se pueden encontrar datos tales como *lmem*, *smem* y *reg*. Estos indican respectivamente el uso de memoria local, compartida (estática, no se contempla la especificada en tiempo de ejecución) y número de registros por hilo. Así pues, bastará con asegurarse de que, en la medida de lo posible, el valor de *lmem* sea siempre 0.

En cualquier caso si esto no fuera posible existen también algunas técnicas destinadas a reducir el número de registros utilizados por hilo. Una conceptualmente muy simple y que a menudo ofrece muy buenos resultados es la separación de un *kernel* en varios ya que la invocación de éstos acarrea un coste prácticamente nulo. De esta manera al reducir el código evaluado por *kernel* es posible que se relaje el número de registros necesarios para la evaluación de cada uno de ellos.

Otra técnica menos conocida pero bastante eficiente es el uso de la palabra clave de C *volatile*. Al declarar algunas variables como volátiles se indica al compilador que no debe tratar de optimizar el uso de éstas sino almacenarlas inmediatamente en memoria

o recalcular su valor cada vez que se utilicen. Esto aplicado a variables locales evita que el compilador asigne registros concretos a almacenar algunas de ellas pudiendo con algo de habilidad reducir de forma importante el número de registros utilizados por hilo.

A pesar de todo cabe llamar la atención en el hecho de que los mejores resultados no tienen por qué obtenerse con el mayor número de hilos posible. Existen medidas de rendimiento y utilización de los multiprocesadores que van en función del número de hilos, memoria compartida y registros utilizados que pueden alcanzar sus picos en valores que no sean necesariamente el máximo. Para ello NVIDIA puso a disposición de los desarrolladores una simple hoja de cálculo llamada *CUDA Occupancy Calculator* donde introduciendo unos pocos parámetros podemos obtener valores y gráficas que ilustran la situación y pueden ayudar a la toma de decisiones de diseño en distintos hardware.

Se recomienda dedicar algún tiempo a experimentar con distintos valores de número de hilos, registros por hilo y memoria compartida analizando los rendimientos obtenidos ya que los resultados pueden no ser necesariamente los esperados.

3. Implementación del algoritmo

Una vez vistos los fundamentos teóricos del algoritmo a implementar, así como el funcionamiento y los detalles de la arquitectura CUDA es el momento de trasladar estos conocimientos a una implementación real que se adecúe a las circunstancias del hardware y la arquitectura.

El objetivo de esta sección es por tanto explicar y justificar las adaptaciones realizadas al algoritmo original así como analizar detalladamente las estrategias empleadas en el proceso de implementación.

3.1. Adaptación del problema a la arquitectura

Partiendo ahora de una base suficientemente sólida sobre el funcionamiento de CUDA es momento de analizar y posiblemente replantear el algoritmo descrito en la sección 2.1 para adaptarlo a las circunstancias y limitaciones anteriormente descritas.

Antes que entrar en detalles debe tenerse en cuenta que existen dos formas notablemente distintas de plantear el problema. La primera de ellas es utilizando la librería *CUBLAS*, una implementación de la famosa *BLAS* acelerada mediante GPUs y proporcionada por NVIDIA, y expresar todas las operaciones del algoritmo 1 utilizándola. Esta librería está diseñada para ser ejecutada desde la CPU invocando distintos *kernels* mediante sus llamadas.

La ventaja de su uso es que simplifica notablemente la implementación evitando la necesidad de profundizar en los detalles de CUDA y proporcionando probablemente una cierta aceleración respecto a su versión secuencial. Por contra, no es posible utilizar esta librería desde la GPU ni en múltiples hilos a la vez desde CPU ya que se basa en la ejecución de *kernels*, perdiendo de esta manera toda posible paralelización de intervalos.

Motivada por esta última consecuencia, la segunda alternativa consiste en implementar el algoritmo *Shift and Invert 2-way Lanczos* en forma de *kernels* en la GPU. De esta manera asignando un bloque de hilos a cada intervalo diferente se consigue un algoritmo que se adapta mejor a las circunstancias del problema tratando de explotar su paralelismo inherente. Esta alternativa es bastante más complicada de implementar, pero dada la situación resulta la más lógica e interesante a investigar decidiendo por tanto adoptarse.

Además debe también tenerse presente que este trabajo se ha orientado completamente a su implementación en una tarjeta GeForce 9800 GX2 con las posibilidades y limitaciones que ello conlleva. Las características de este hardware fueron ya descritas en la tabla 1, de las cuales merecen especial atención la ausencia de aceleración de coma flotante de doble precisión, el hecho de que se compone de 2 GPUs y los 512 MB de memoria total (global) por cada una de ellas. El uso de este hardware con capacidad computacional de CUDA versión 1.1 también restringe a los criterios y patrones de acceso a memoria global descritos en la sección 2.2.3 así como al número de registros por multiprocesador indicados en la sección 2.2.5.

3.1.1. Limitaciones de memoria

Si se analizan los requisitos de memoria de la aproximación descrita en la sección 2.1.2 se puede observar que para poder resolver los sistemas Toeplitz simétricos es necesario precalcular la descomposición LDL^T de las matrices desplazadas correspondientes al *shift* de cada intervalo siendo necesario almacenar estos valores. En concreto es necesario almacenar dos matrices triangulares de talla aproximadamente $n/2$ por cada uno de los intervalos paralelos que intenten ejecutarse, aunque en la práctica es probable que se requiera el doble de información con el fin de mantener los accesos de tipo *coalesced* a la versión traspuesta de estas matrices.

Si bien puede comprobarse que matrices pequeñas grandes cabrían sin problemas en memoria gráfica, al ascender a tamaños del orden de 5000×5000 elementos encontramos que cada uno de los intervalos requeriría cerca de 50 MB de memoria, sin contar otros usos auxiliares que sin duda aumentarían este valor. Este hecho limitaría drásticamente el número de intervalos lanzados en paralelo en una GPU a apenas unos 10 de entre varios cientos o miles, sacrificando buena parte del paralelismo inherente al problema.

Esta situación llevó a la toma de una decisión de diseño que no acotase de semejante forma la talla o el paralelismo soportados. De esta manera y a su vez para tratar de evitar conflictos derivados de la alineación de memoria en el cálculo de permutaciones y en la transformada discreta del seno, se optó por replantear la resolución de sistemas Toeplitz simétricos utilizando el algoritmo de Levinson-Durbin en lugar de la aproximación por matrices tipo Cauchy.

Este algoritmo de coste temporal $\mathcal{O}(n^2)$ es capaz de resolver los sistemas sin necesidad de expresar explícitamente la matriz Toeplitz en memoria reduciendo así su complejidad espacial a $\mathcal{O}(n)$ y solucionando con ello los problemas de limitación de memoria. No obstante acarrea un problema adicional: puede volverse numéricamente inestable al aplicarse a matrices simétricas que no sean definidas positivas.

Este problema se torna más grave si se tiene en cuenta que no se está intentando resolver la matriz original, sino diversas versiones desplazadas de la misma cuyo desplazamiento consiste justamente en restar valores a la diagonal principal. De esta forma las matrices sobre las que se aplica el algoritmo son perfectas candidatas a introducir y acarrear inestabilidades numéricas en el proceso, hecho especialmente negativo dadas las limitaciones de la coma flotante de simple precisión.

Existen algunas aproximaciones a la resolución de este problema tratando de extender la estabilidad numérica del algoritmo de Levinson a las matrices Toeplitz simétricas indefinidas como por ejemplo el método descrito en [3], ya mencionado en el estado del arte, donde se proponen técnicas de *look-ahead* basadas en eliminación Gaussiana para evitar los pasos que introducen inestabilidad en el resultado. No obstante dada la complejidad que suponía la correcta y eficiente implementación en CUDA de estas modificaciones al método de Levinson original se decidió relegarlas a futuras ampliaciones en función de los resultados obtenidos.

3.1.2. Limitaciones en registros disponibles e hilos por bloque

Otra de las consecuencias de la adopción de este algoritmo está relacionada con las dificultades que surgen de su implementación. Tal y como se define el algoritmo original, el cual se analizará en mayor detalle en la sección 3.4, la resolución de un sistema Toeplitz simétrico requiere una notable cantidad de operaciones en las que se accede a vectores en orden inverso: desde el último elemento hacia atrás hasta el primero. Este tipo de accesos a memoria entra directamente en conflicto con los patrones de acceso a memoria global descritos en la sección 2.2.3, requiriendo diversas operaciones adicionales con la memoria compartida para alinear los datos manteniendo los accesos tipo *coalesced*.

Estas operaciones adicionales y la complejidad algorítmica que conllevan produce un significativo aumento del número de registros necesarios por hilo, especialmente por el hecho de que todas las funciones *__device__* se implementan en la práctica como funciones inline. Las pruebas realizadas muestran, previa aplicación de algunas optimizaciones descritas en la sección 2.2.5, requisitos de hasta 40 registros por hilo. Este valor puede reducirse hasta los 32 registros por hilo aplicando las técnicas de optimización anteriores basadas en la palabra clave *volatile*.

Las limitaciones del hardware a 8192 registros por multiprocesador, combinadas con la necesidad de 32 registros por hilo limitan el número máximo de hilos por bloque a 256, la mitad del máximo posible. A pesar de las diferentes optimizaciones aplicables resulta inviable tratar de reducir este valor a los 16 registros por hilo que permitirían alcanzar los 512 hilos por bloque sin provocar molestos y lentos accesos a memoria local.

Esta reducción del número de hilos por bloque podría tratar de evitarse en nuevos hardware con capacidades computacionales 1.2 o superiores ya que aumenta el número de registros por multiprocesador hasta 16384. No obstante habría que vigilar también que esto no diese lugar a insuficiencias en la memoria compartida, especialmente si se combina con el uso de coma flotante de doble precisión en bloques auxiliares de datos.

3.1.3. Limitaciones de la coma flotante de simple precisión

Otro de los grandes problemas del hardware empleado es su ausencia de aceleración hardware para los cálculos en coma flotante de doble precisión, relegando todas las operaciones aritméticas eficientes al uso de la simple precisión.

Los problemas de precisión numérica son algo recurrente al trabajar con algoritmos numéricos como es el caso y por lo general se abordan siempre directamente utilizando coma flotante de doble precisión, descartando la simple por ser candidata a causar notables efectos nocivos en la convergencia y la calidad del resultado. Este caso no es diferente y debe tratarse con especial cautela, teniendo además el agravante de la posible inestabilidad numérica del algoritmo de Levinson para matrices Toeplitz simétricas indefinidas.

Puesto que tratar de mejorar la precisión del resultado sin realizar cambios serios en el algoritmo o en la forma de representar los datos es algo inviable se han tratado de paliar los efectos que una pobre convergencia pueda tener sobre el proceso de extracción. Una forma de hacerlo ha sido limitando excesivo crecimiento de los subespacios de Krylov que el algoritmo 1 genera mediante la reducción del número de valores propios por intervalo a valores ínfimos, del orden de tan sólo dos o tres.

Esta reducción del número de valores propios por intervalo incrementa notablemente el número de intervalos paralelos, pero también contribuye a reducir el consumo de memoria interno de cada uno de ellos. Desafortunadamente aunque esto ayude a evitar una convergencia catastrófica también actúa como elemento en contra de la eficiencia, ya que el trabajo realizado por la GPU se aprovecha en menor medida dando lugar a un mayor número de operaciones a la larga.

Otra medida a favor de aumentar la convergencia del algoritmo se basa en algunas circunstancias de la base teórica del mismo. La elección del valor de *shift* σ de cada intervalo resulta ser elemento clave en la extracción eficiente de los valores propios del mismo. Esto se debe a que por propia definición de los valores propios y el polinomio característico si el valor σ llegase a coincidir con alguno de los valores propios la matriz se volvería singular produciendo el fallo del proceso de extracción.

Si bien la coincidencia de tales valores resulta extremadamente poco probable incluso bajo la limitada precisión sobre la que se trabaja, valores relativamente cercanos producen notables inestabilidades numéricas en el proceso. Por este motivo así como por los propios problemas de la convergencia en simple precisión se añadió la posibilidad de relanzar automáticamente aquellos intervalos que no han conseguido una convergencia completa.

Este relanzado de los intervalos se realiza hasta un número máximo de veces especificado y utilizando un valor σ aleatorio dentro del intervalo de extracción. Cabe recordar que por cuestiones de eficiencia el σ utilizado por defecto en la primera evaluación de los distintos intervalos es siempre el punto medio de éstos.

Otro problema peculiar del uso de este hardware es su implementación de la simple precisión. Según afirma la guía oficial de programación en CUDA [11] las operaciones de multiplicación y suma son a menudo combinadas en una sola instrucción *FMAD* para aumentar su eficiencia. No obstante esta instrucción trunca el resultado intermedio de la multiplicación, pudiendo introducir poco a poco pequeños errores de precisión. Del mismo modo, las operaciones de división y raíz cuadrada también se implementan de una forma no conforme al estándar IEEE 754, haciéndolo mediante el cálculo del valor recíproco.

Los distintos modos de redondeo del estándar tampoco pueden configurarse dinámicamente, teniendo además casos como los de las operaciones de suma y multiplicación donde solo el redondeo al par más cercano y a cero están soportados obviando el redondeo a $\pm \infty$. Tampoco existe modo alguno de capturar las excepciones generadas por la coma flotante.

Cabe resaltar que en aquellos hardware que la soportan la implementación coma flotante de doble precisión resuelve algunos problemas como el de las instrucciones *FMAD*, las implementaciones no estándar de división y raíz cuadrada y algunos problemas en el soporte de los modos de redondeo.

Así pues las restricciones del hardware empleado, una tarjeta gráfica GeForce 9800 GX2 con capacidad computacional versión 1.1, obligan por cuestiones de viabilidad a adoptar algunas decisiones de diseño que pueden resultar ineficientes o añadir inestabilidad numérica al problema. Partiendo de estos hechos, a continuación se describen los detalles de la implementación del algoritmo con las modificaciones aquí propuestas.

3.2. Estrategias de implementación adoptadas

Tal y como se mencionaba en la sección 3.1 se ha optado por una implementación en la GPU del algoritmo *Shift and Invert 2-way Lanczos* descrito como algoritmo 1 en la sección 2.1.1, de forma que cada bloque de hilos se encargue de procesar en paralelo y de forma independiente uno de los intervalos de extracción de valores propios.

Estos intervalos son precalculados antes de iniciar el proceso de extracción usando una librería externa escrita en lenguaje *fortran 90*. Previo cálculo del mayor de los discos de Gershgorin del problema, esta librería es capaz de devolver una lista de intervalos que contienen valores propios hasta una cantidad máxima que se le especifique permitiendo así balancear adecuadamente la carga computacional.

Al no depender de la GPU de ninguna manera este cálculo puede realizarse plenamente en la CPU siendo en este caso posible el uso interno de las técnicas descritas en la sección 2.1.2. De esta forma, la librería acota el número de valores propios de cada intervalo a partir del número de elementos negativos de la matriz diagonal devuelta por la descomposición LDL^T lo que, combinado con técnicas de búsqueda por bisección, permite aislar rápidamente los intervalos de extracción.

Una vez calculados estos intervalos, debe tenerse en cuenta que a pesar de la decisión de utilizar el algoritmo de Levinson-Durbin por sus requisitos de memoria proporcionales

a la talla del problema, su escalabilidad los a cientos o miles de intervalos generados a causa del escaso número de valores propios de cada uno de ellos que sugería en la sección 3.1.3 entra en conflicto directo con las limitaciones de memoria del hardware.

De esta manera, y dado que en la aunque práctica sí se ejecutan varios múltiples bloques de hilo en paralelo nunca lo haría tal cantidad de ellos se propone el uso de una ventana de ejecución. Ésta tendrá un tamaño limitado por el usuario o las características del hardware empleado y contendrá información de aquellos intervalos que se procesarán en un mismo *kernel*.

De esta manera, independientemente del número de intervalos generados éstos se introducirán en la ventana de ejecución hasta llenarla dejando lugar a otros una vez concluya su convergencia. Gracias a esto, la cantidad de recursos de memoria requeridos crece en función del tamaño de esta ventana en lugar del número de intervalos calculados.

Adicionalmente, esta aproximación basada en la asignación de intervalos a una ventana de ejecución permite una fácil extensión a entornos de múltiples GPUs pudiendo simplemente crear una ventana de ejecución para cada una de ellas mientras se mantiene un sistema de asignación global. Esta simple modificación permitiría duplicar el número de intervalos evaluados en paralelo de forma efectiva.

Otra cuestión pendiente de resolver en la implementación actual es el tema de la verificación de convergencia. Tal y como se describe en algoritmo 1 éste se encarga de la extracción de los valores y vectores propios de las matrices tridiagonales generadas a partir de los datos del subespacio, quedándose luego con aquellos valores propios que sean buenas aproximaciones a los de la matriz original.

La implementación del algoritmo tal y como se especifica en la GPU implicaría varias complejidades añadidas siendo la primera de ellas el cálculo eficiente de los valores propios de las matrices tridiagonales. Puesto que este cálculo es relativamente sencillo y puede llevarse a cabo eficientemente por parte de la CPU haciendo uso por ejemplo de la librería LAPACK, se ha decidido que la implementación de GPU se limite al cálculo de los valores tridiagonales dejando a la CPU el resto. De esta manera el algoritmo original 1 adaptado a la GPU quedaría como se muestra en el algoritmo 4.

La centralización de las extracciones de valores y vectores propios de las matrices tridiagonales así como las comprobaciones de convergencia por parte de la CPU pueden resultar un cuello de botella si no se realizan adecuadamente. Una primera aproximación podría ser la evaluación en múltiples hilos por parte de los diferentes intervalos a fin de reducir el tiempo necesario total.

No obstante existe una manera más simple de lograr mejores resultados: evaluar con la CPU la convergencia de la k-ésima iteración mientras la GPU está calculando los valores tridiagonales de la $(k+1)$-ésima iteración en paralelo. Adicionalmente pueden procesarse varias iteraciones del algoritmo de Lanczos en la evaluación de un mismo *kernel* por lo que en la práctica se hablaría de las iteraciones k-ésima y $(k+\triangle k)$-ésima.

Algoritmo 4 Adaptación del algoritmo Shift and invert 2-way Lanczos a la GPU

Sean $p_1 = J_n p_1 \neq 0$ y $q_1 = -J_n q_1 \neq 0$ vectores iniciales con $\|p_1\|_2 = \|q_1\|_2 = 1$
Sean $p_0 = q_0 = \vec{0}$ y $\beta_0 = \delta_0 = 0$
for $k = k_0, k_0 + 1, \ldots,$ mín$(max_k, k_0 + \triangle k)$ **do**
  $w = p_k + q_k$
  Resolver el sistema $(T - \sigma I)\, v = w$ usando el algoritmo de Levinson.
    $v_s = 0{,}5 \cdot (v + J_n v)$ $v_a = 0{,}5 \cdot (v - J_n v)$
    $\alpha_k = v_s^T \cdot p_k$ $\gamma_k = v_a^T \cdot q_k$
    $v_s = v_s - \alpha_k p_k - \beta_{k-1} p_{k-1}$ $v_a = v_a - \gamma_k q_k - \delta_{k-1} q_{k-1}$
  Reortogonalización completa:
    $v_s = v_s - p \cdot (p^T \cdot v_s)$ $v_a = v_a - q \cdot (q^T \cdot v_a)$
    $\beta_k = \|v_s\|_2$ $\delta_k = \|v_a\|_2$
    $p_{k+1} = v_s / \beta_k,$ $q_{k+1} = v_a / \delta_k$
  Almacenar valores $\alpha_k, \beta_k, \gamma_k, \delta_k$ en memoria global para comunicarlos a la CPU.
end for

Este esquema descompone el proceso de extracción y sincronización entre CPU y GPU en distintas etapas: ejecución del *kernel*, verificación de convergencia y sincronización y copia de resultados. De esta forma el proceso global queda como sigue:

Algoritmo 5 Estructura general de la implementación del algoritmo en la CPU

Calcular el disco de Gershgorin de la matriz de entrada.
Dividir el disco obtenido en intervalos con semejante número de valores propios.
Crear una ventana de ejecución por GPU y reservar memoria.
Llenar las ventanas de ejecución con intervalos usando $\sigma = (a + b)/2$.
while alguna ventana de ejecución tenga elementos **do**
  Evaluar el algoritmo 4 en cada GPU con su propia ventana de ejecución.
  for cada intervalo en las ventanas de ejecución **do**
    for etapas $k - \triangle k, \ldots, k - 1$ con $k > 0$ **do**
      Extraer valores propios y comprobar convergencia.
      Insertar nuevo intervalo en caso de convergencia o relanzar el actual con σ aleatorio si se alcanza max_k, descartándolo tras varios intentos.
    end for
  end for
  for cada intervalo no recién iniciado en las ventanas de ejecución **do**
    Esperar la finalización del kernel correspondiente (sincronización).
    Copiar datos de las matrices tridiagonales y los vectores de Lanczos a la CPU.
  end for
end while
Unir los resultados de los distintos intervalos y ordenarlos.

El código fuente de este algoritmo puede encontrarse en el archivo *gpu_eigen.cu*. Adicionalmente, puede encontrarse una versión secuencial en CPU del proceso completo en el archivo *cpu_eigen.c*. Esta versión implementa el algoritmo 1 mediante la librería BLAS sin hacer uso alguno de la GPU. Por otra parte, las funciones encargadas del cálculo y la convergencia de los valores propios se encuentran en el archivo *eigen.c*.

3.3. Operaciones básicas en la GPU

Una vez visto el algoritmo por la parte de la CPU es momento de centrarse en las operaciones realizadas por el hardware gráfico. Tal y como se ha mencionado anteriormente el objetivo es lograr implementar un *kernel* que realice las operaciones descritas en el algoritmo 4 manteniendo siempre los accesos a memoria de tipo *coalesced* descritos en la sección 2.2.3 y de la forma más eficiente posible.

Para lograr este fin y para obtener un código más comprensible y reutilizable se han definido una serie de operaciones vectoriales y matriciales básicas sobre las que se basan las implementaciones del algoritmo de Levinson y de Shift and Invert 2-way Lanczos. Estas operaciones se basan en un subconjunto de las ofrecidas por BLAS 1 aunque debidamente especializadas al cálculo concreto que realizan para tratar de maximizar su eficiencia.

Las implementaciones de estos métodos pueden encontrarse en el apéndice A. Éstas están tomadas del archivo *gpu_vector_kernel.cu* del código fuente, con la excepción del método *reorthogonalize* que se encuentra en el archivo *toeplitz_kernel.cu* junto a las implementaciones de Levinson y Shift and Invert 2-way.

Varias de estas operaciones requieren apoyarse en bloques temporales de memoria compartida para asegurar los accesos óptimos a memoria global. Para ello y como referencia se ha empleado la macro *blockSize* definida en el archivo de cabecera genérico *toeplitz.h* y asignada al número de hilos utilizados por bloque, en este caso 256 dadas las limitaciones descritas en la sección 2.2.5. Del mismo modo se asume que la variable *tid* contiene el identificador del hilo respecto al bloque local.

A continuación se describen estas operaciones básicas para posteriormente y a partir de ellas analizar las implementaciones de Levinson y Shift and Invert 2-way Lanczos.

3.3.1. Método *axpy*

Este conocido método realiza la operación $y = a \cdot x + y$ donde $a \in \mathbb{R}$ y $x, y \in \mathbb{R}^n$. Su implementación es bastante simple y no requiere especial mención, aunque su estructura sirve como base para los siguientes y más complicados métodos.

Algoritmo 6 Implementación del método *axpy*

1: **for** $b = 0$ con $b < n$ en pasos de tamaño $blockSize$ **do**
2: $\quad i = b + tid$
3: $\quad y_i = y_i + a \cdot x_i$ para $i < n$
4: **end for**
5: Sincronizar hilos.

Como puede verse la estrategia seguida consiste en procesar los vectores por bloques del tamaño del número de hilos y asignar el trabajo de cada componente del vector a uno de ellos. De esta forma puesto que los vectores se encuentran correctamente alineados y cada hilo accede a posiciones consecutivas se consiguen accesos de tipo *coalesced* sin necesidad de hacer uso alguno de la memoria compartida.

3.3.2. Métodos *norm* y *dot*

Estos dos métodos que implementan la norma euclídea y el producto escalar respectivamente se implementan de forma casi idéntica puesto que la única diferencia entre ellos es que *norm* calcula una raíz cuadrada al final mientras que *dot* accede a dos vectores distintos en lugar de a uno. No obstante $\mathcal{O}(n)$ accesos adicionales a memoria global no es algo despreciable por lo que se ha decidido separarlos en dos funciones específicas.

Además estos dos métodos implican un cálculo de reducción, donde un conjunto de valores se proyecta sobre uno solo. En este caso es una reducción de suma, que se realiza aproximadamente en $\mathcal{O}(\log n)$ pasos. Adicionalmente se aprovechan algunas propiedades de usar un número de hilos potencia de dos.

Algoritmo 7 Implementación del método *dot*

Sea br un bloque de tamaño $blockSize$ en memoria compartida.
Sea $result$ un acumulador inicializado a 0.
for $b = 0$ con $b < n$ en pasos de tamaño $blockSize$ **do**
  $i = b + tid$
  $br_i = x_i \cdot y_i$ para $i < n$, con $br_{tid} = 0$ en caso contrario.
  Sincronizar hilos.
  for $s = blockSize$ mientras $s > 1$ **do**
    $s = \lfloor s/2 \rfloor$
    $br_{tid} = br_{tid} + br_{tid+s}$ para $tid < s$.
    Sincronizar hilos.
  end for
  $result = result + br_0$
end for
return $result$

Algoritmo 8 Implementación del método *norm*

Sea br un bloque de tamaño $blockSize$ en memoria compartida.
Sea $result$ un acumulador inicializado a 0.
for $b = 0$ con $b < n$ en pasos de tamaño $blockSize$ **do**
  $i = b + tid$
  $br_i = x_i^2$ para $i < n$, con $br_{tid} = 0$ en caso contrario.
  Sincronizar hilos.
  for $s = blockSize$ mientras $s > 1$ **do**
    $s = \lfloor s/2 \rfloor$
    $br_{tid} = br_{tid} + br_{tid+s}$ para $tid < s$.
    Sincronizar hilos.
  end for
  $result = result + br_0$
end for
return $\sqrt{result}$

Cabe resaltar que en la implementación final de estos métodos se ha realizado una pequeña optimización extrayendo del bucle de reducción las tres últimas iteraciones, haciendo que sean procesadas secuencialmente por el hilo con identificador 0.

3.3.3. Método *dot_reverse_y*

Este método implementa el producto escalar de dos vectores $x \cdot reverse(y)$ con $x, y \in \mathbb{R}^n$ donde al segundo de ellos se accede desde atrás hacia delante con las diversas complicaciones que ello conlleva. No sólo está el problema del orden de acceso a los elementos sino que muy probablemente existan además problemas de alineación de memoria al acceder a los últimos elementos del vector y.

Para subsanar este problema ha sido necesario contar con múltiples bloques en memoria compartida: uno de longitud $2 \times blockSize - 1$ para almacenar e ir desplazando los datos de diferentes accesos al vector x, otro de longitud $blockSize$ que contenga los últimos datos leídos del vector y, y un último de longitud $blockSize$ que permita calcular la reducción por suma. De esta forma el primero solventa los problemas de alineación y tamaño de los vectores mientras que el segundo permite resolver los problemas de orden a los k-ésimos elementos por los k-ésimos hilos.

Algoritmo 9 Implementación del método *dot_reverse_y*

1: Sea bx un bloque de tamaño $2 \times blockSize - 1$ en memoria compartida.
2: Sea by un bloque de tamaño $blockSize$ en memoria compartida.
3: Sea br un bloque de tamaño $blockSize$ en memoria compartida.
4: Sean n_{align} el menor múltiplo de $blockSize >= n$ y $shift = n_{align} - n$.
5: Sea $result$ un acumulador inicializado a 0.
6: Inicializar $bx_{tid} = 0$ y sincronizar hilos.
7: **for** $b = 0$ con $b < n$ en pasos de tamaño $blockSize$ **do**
8: Calcular índices: $xi = b + tid, \quad yi = n_{align} - b - blockSize + tid$
9: Leer datos de x e y:
$bx_{shift+tid} = x_{xi}$ para $xi < n$
$by_{tid} = y_{yi}$ para $yi < n$, con $by_{tid} = 0$ en caso contrario.
10: Sincronizar hilos.
11: Calcular producto escalar: $br_{tid} = bx_{tid} \cdot by_{\ blockSize-tid-1}$
12: Sincronizar hilos.
13: Desplazar contenidos de bx:
$bx_{tid} = bx_{tid+blockSize}$ para $tid < shift$
14: **for** $s = blockSize$ mientras $s > 1$ **do**
15: $s = \lfloor s/2 \rfloor$
16: $br_{tid} = br_{tid} + br_{tid+s}$ para $tid < s$
17: Sincronizar hilos.
18: **end for**
19: $result = result + br_0$
20: **end for**
21: **return** $result$

Este método puede resultar algo confuso. Lo que en realidad trata de hacer es dejar inicialmente un hueco en los datos leídos al inicio del vector x para compensar los problemas de alineación de la última posición de y. Luego tan sólo se procesan los primeros $blockSize$ datos desplazando los no procesados de bx al principio de éste para la siguiente iteración. De esta manera se resuelven los problemas anteriores consiguiendo siempre accesos de tipo *coalesced* independientemente de la talla del problema.

3.3.4. Método *axpy_reverse_x*

Este otro método implementa la operación $y = a \cdot reverse(x) + y$ con $a \in \mathbb{R}$ y $x, y \in \mathbb{R}^n$, donde el vector x se accede desde atrás hacia delante. Aunque conceptualmente es casi idéntica a la primera función descrita su implementación es notablemente más compleja.

Este método sufre los mismos problemas de acceso que el anterior y por tanto puede tratar de aproximarse de la misma manera. No obstante en este caso los resultados de cada bloque procesado ya no son acumulables en un escalar y tampoco pueden almacenarse siempre al instante al no estar necesariamente completos. Para ello, aunque ya no será necesario un bloque auxiliar de reducción, se utilizará un bloque temporal de resultados de tamaño $2 \times blockSize$ sobre el que se desplazarán éstos para alinearlos apropiadamente y completar un bloque entero de ellos antes de almacenarlos de nuevo en memoria global.

Nótese también que a diferencia del caso anterior esta vez es el vector x el que es accedido de forma inversa, por lo que los papeles de los bloques auxiliares bx y by se invierten respecto al método anterior.

Algoritmo 10 Implementación del método *axpy_reverse_x*

1: Sea bx un bloque de tamaño $blockSize$ en memoria compartida.
2: Sea by un bloque de tamaño $2 \times blockSize - 1$ en memoria compartida.
3: Sea br un bloque de tamaño $2 \times blockSize$ en memoria compartida.
4: Sea n_{align} el menor múltiplo de $blockSize \geq n$.
5: Sean $shift = n_{align} - n$ y $lastBlock = blockSize - shift$.
6: **for** $b = 0$ con $b < n$ en pasos de tamaño $blockSize$ **do**
7: Calcular índices: $xi = n_{align} - b - blockSize + tid, \quad yi = b + tid$
8: Leer datos de x e y:
$bx_{tid} = x_{xi}$ para $xi < n$
$by_{shift+tid} = y_{yi}$ para $yi < n$
9: Sincronizar hilos.
10: Calcular axpy:
Si $b > 0$: $br_{tid+lastBlock} = a \cdot bx_{\ blockSize-tid-1} + by_{tid}$
Sino: $br_{tid-shift} = a \cdot bx_{\ blockSize-tid-1} + by_{tid}$ para $tid \geq shift$
11: Sincronizar hilos.
12: Almacenar resultados de la iteración anterior:
Si $b > 0$: $y_{yi-blockSize} = br_{tid}$
13: Desplazar contenidos de by:
$by_{tid} = by_{tid+blockSize}$ para $tid < shift$
14: Desplazar resultados incompletos de br:
Si $b > 0$, $br_{tid} = br_{tid+blockSize}$ para $tid < lastBlock$
15: Sincronizar hilos.
16: **end for**
17: Almacenar datos restantes del bloque de resultados temporales:
$i = n_{align} - blockSize + tid$
$y_i = br_{tid}$ para $i < n$
18: Sincronizar hilos.

3.3.5. Método *axpxb_reverse_x*

Este método implementa la operación $x = b \cdot (a \cdot reverse(x) + x)$ con $a \in \mathbb{R}$ y $x \in \mathbb{R}^n$. Se trata de una extensión del método anterior donde se aprovecha que x e y son el mismo vector procesando la operación al principio y al final de éste a la vez evitando así la necesidad de realizar una copia temporal de x.

Este método requiere introducir un bloque temporal más en memoria compartida: uno para almacenar los resultados de procesar el vector hacia atrás mientras que el anterior se utilizará para procesar los resultados del vector hacia adelante con sus correspondientes desplazamientos como ocurría en el caso anterior.

Algoritmo 11 Implementación del método *axpxb_reverse_x*

1: Sea bf un bloque de tamaño $2 \times blockSize - 1$ en memoria compartida.
2: Sea brf un bloque de tamaño $2 \times blockSize$ en memoria compartida.
3: Sean bb y brb bloques de tamaño $blockSize$ en memoria compartida.
4: Sea n_{align} el menor múltiplo de $blockSize \geq n$.
5: Sean $shift = n_{align} - n$ y $lastBlock = blockSize - shift$.
6: Sean $num\_blocks = \lceil n/blockSize \rceil$ y $max_n = blockSize \times (1 + \lfloor num\_blocks/2 \rfloor)$.
7: **for** $b = 0$ con $b < max_n$ en pasos de tamaño $blockSize$ **do**
8: Calcular índices: $bi = n_{align} - b - blockSize + tid, \quad fi = b + tid$
9: Leer datos de x:
$bb_{tid} = x_{bi}$ para $bi < n$
$bf_{shift+tid} = x_{fi}$ para $fi < n$
10: Sincronizar hilos.
11: **if** $b > 0$ **then**
12: Forward result: $brf_{tid+lastBlock} = b \cdot (a \cdot bb_{\ blockSize-tid-1} + bf_{tid})$
13: Backward result: $brb_{tid} = b \cdot (bb_{tid} + a \cdot bf_{\ blockSize-tid-1})$
14: **else**
15: Forward result: $brf_{tid-shift} = b \cdot (a \cdot bb_{\ blockSize-tid-1} + bf_{tid})$ para $tid \geq shift$
16: Backward result: $brb_{shift} = b \cdot (bb_{tid} + a \cdot bf_{\ blockSize-tid-1})$ para $tid < lastBlock$
17: **end if**
18: Sincronizar hilos.
19: Almacenar resultados *forward* de la iteración anterior:
Si $b > 0$: $x_{fi-blockSize} = brf_{tid}$
20: Si num_blocks es impar y es la última iteración del bucle, salir de él.
21: Almacenar resultados *backward* de esta iteración: $x_{bi} = brb_{tid}$ para $bi < n$
22: Desplazar contenidos *forward* de bf:
$bf_{tid} = bf_{tid+blockSize}$ para $tid < shift$
23: Desplazar resultados incompletos *forward* de brb:
Si $b > 0$, $brf_{tid} = brf_{tid+blockSize}$ para $tid < lastBlock$
24: Sincronizar hilos.
25: **end for**
26: Sincronizar hilos.

Como puede verse este método es más enrevesado todavía ya que necesita realizar desplazamientos en dos bloques distintos: el de datos recorriendo el vector hacia delante, y el de resultados en este mismo sentido.

Además estos resultados se escriben siempre una iteración después de la correspondiente para asegurarse de que estén completos, mientras que los resultados de procesar el vector hacia atrás carecen de este problema y se escriben en la misma. Esto requiriere contemplar cuidadosamente el caso en que el número de casos sea impar para evitar escrituras dobles o inválidas.

Un ejemplo de cómo se procesa un vector, asumiendo por simplicidad en la representación $blockSize = 4$ y 4 hilos por bloque, sería el siguiente:

Situación inicial:

x:	$x_1\ x_2\ x_3\ x_4$	$x_5\ x_6\ x_7\ x_8$	$x_9\ x_{10}\ -\ -$

Primera iteración: (sin incluir desplazamientos de bloque)

Forward block (bf):	$-\ -\ x_1\ x_2$	$x_3\ x_4\ -$
Forward result block (brf):	$r_1 r_2\ -\ -$	$-\ -\ -\ -$

Backward block (bb):	$x_9\ x_{10}\ -\ -$
Backward result block (brb):	$\mathbf{r_9\ r_{10}}\ -\ -$

x:	$x_1\ x_2\ x_3\ x_4$	$x_5\ x_6\ x_7\ x_8$	$\mathbf{r_9\ r_{10}}\ -\ -$

Segunda iteración: (sin incluir desplazamientos de bloque)

Forward block (bf):	$x_3\ x_4\ x_5\ x_6$	$x_7\ x_8\ -$
Forward result block (brf):	$\mathbf{r_1\ r_2\ r_3\ r_4}$	$r_5\ r_6\ -\ -$

Backward block (bb):	$x_5\ x_6\ x_7\ x_8$
Backward result block (brb):	$\mathbf{r_5\ r_6\ r_7\ r_8}$

x:	$\mathbf{r_1\ r_2\ r_3\ r_4}$	$\mathbf{r_5\ r_6\ r_7\ r_8}$	$r_9\ r_{10}\ -\ -$

Cuadro 4: Ejemplo de traza del método *axpxb_reverse_x*

3.3.6. Método *reorthogonalize*

Este último método ya no realiza operaciones entre vectores sino un par de operaciones matriz-vector. Concretamente implementa el cálculo de reortogonalización completa de un vector respecto a una base que en este caso será de un subespacio de Krylov.

Con el fin de evitar problemas en el acceso a memoria de la matriz p y su traspuesta, ésta utiliza un valor de *leading dimension* de $n_{align} \geq n$ múltiplo de $blockSize$. Esto asegura que el principio de cada columna de la matriz se encuentre adecuadamente alineado a las restricciones de los accesos de tipo *coalesced*.

La operación implementada se define formalmente como $v = v - p \times h$ con $p \in \mathbb{R}^{n \times k}$ base del subespacio de dimensión $k, h \in \mathbb{R}^k = p^T \times v$ y $v \in \mathbb{R}^n$. Así pues el cálculo se divide en dos partes bien diferenciadas: $h = p^T \times v$ por un lado y $v = v - p \times h$ por otro.

La implementación de la primera parte se realiza como una serie de productos escalares con el método *dot* con accesos debidamente alineados a memoria mientras que la segunda parte acumula por bloques las columnas de la matriz p escaladas por el i-ésimo elemento de h. De esta forma se asegura que todos los accesos sean de tipo *coalesced*.

Así pues, el método queda implementado como sigue.

Algoritmo 12 Implementación del método *reorthogonalize*

1: Sea h un vector auxiliar de dimensión k en memoria global.
2: Sea n_{align} el menor múltiplo de $blockSize \geq n$.
3: Sea p_i la i-ésima columna de p empezando por p_0.
4: **for** $bk = 0$ con $bk < k$ en pasos de tamaño $half\ warp$ **do**
5: Sea dr un bloque de tamaño $half\ warp$ en memoria compartida.
6: **for** $w = 0$ con $w < half\ warp$ y $bk + w < k$ en pasos de tamaño n_{align} **do**
7: $dr_w = \mathbf{dot}(n, p_i, v)$
8: $w = w + 1, \quad i = i + 1$
9: **end for**
10: Almacenar dr en h: $bi = bk + tid, \quad h_{bi} = dr_{tid}$ con $tid < half\ warp$ y $bi < k$
11: Sincronizar hilos.
12: **end for**
13: **for** $bn = 0$ con $bn < n$ en pasos de tamaño $blockSize$ **do**
14: Sean bh y ba dos bloques de dimensión $blockSize$ en memoria compartida.
15: Inicializar $ba_{tid} = 0$ y sincronizar hilos.
16: Calcular índice: $j = bn + tid$
17: **for** $bk = 0$ con $bk < k$ en pasos de tamaño $blockSize$ **do**
18: Leer datos: $ki = bk + tid, \quad bh_{tid} = h_{ki}$ con $ki < k$
19: Sincronizar hilos.
20: Acumular resultado para $0 \leq i < k$: $ba_{tid} = ba_{tid} + bh_i \cdot p_{i,j}$ con $j < n$
21: **end for**
22: Sincronizar hilos.
23: Restar acumulador al vector v: $v_j = v_j - ba_{tid}$ con $j < n$
24: **end for**
25: Sincronizar hilos.

3.4. Algoritmo de Levinson

Una vez vistas las operaciones básicas es momento de pasar a los algoritmos principales implementados. El primero de ellos es el ya mencionado anteriormente algoritmo de Levinson-Durbin, utilizado para la resolución de sistemas Toeplitz simétricos y a ser posible definidos positivos.

Dados $t \in \mathbb{R}^n$ vector que define los contenidos de la matriz Toeplitz simétrica, $b \in \mathbb{R}^n$ vector de términos independientes, $y \in \mathbb{R}^n$ un vector auxiliar y $x \in \mathbb{R}^n$ vector de incógnitas a calcular, su definición tradicional basada en la descrita en [5] es la siguiente:

Algoritmo 13 Algoritmo de Levinson

1: $t_{1\ldots n-1} = t_{1\ldots n-1}/t_0, \quad b = b/t_0$
2: $x_0 = b_0, \quad y_0 = -t_0, \quad \alpha = -t_0, \quad \beta = 1$
3: **for** $k = 1$ con $k < n$ **do**
4: $\quad \beta = (1 - \alpha^2) \cdot \beta$
5: $\quad \mu = (b_k - {t_{0\ldots k-1}}^T \cdot x_{k-1\ldots 0})/\beta$
6: $\quad y_{0\ldots k-1} = \mu \cdot y_{k-1\ldots 0} + x_{0\ldots k-1}$
7: $\quad x_k = \mu$
8: $\quad$ **if** $k < n - 1$ **then**
9: $\quad\quad \alpha = -(t_k + {t_{0\ldots k-1}}^T \cdot y_{k-1\ldots 0})/\beta$
10: $\quad\quad y_{0\ldots k-1} = \alpha \cdot y_{k-1\ldots 0} + y_{0\ldots k-1}$
11: $\quad\quad y_k = \alpha$
12: $\quad$ **end if**
13: **end for**

Este algoritmo puede encontrarse implementado directamente de esta manera en el archivo *toeplitz.c* ya que es utilizado por la versión CPU de la implementación.

Tal y como se plantea originalmente este algoritmo evita la construcción explícita de la matriz Toeplitz simétrica definida por t en memoria, motivo que como se ha comentado anteriormente ha llevado a la elección de este algoritmo pese a sus problemas. No obstante al operar iterativamente con vectores de cada vez mayor tamaño se tiene el inconveniente de que se generan un elevado número de accesos a memoria.

Esta cantidad de accesos a memoria, aunque sean todos ellos de tipo *coalesced*, pueden afectar al rendimiento de este algoritmo respecto a otros que realicen un menor número de accesos evitando así una situación de cuello de botella.

Volviendo al código de la implementación, como puede observarse aparecen varias operaciones vectoriales básicas en las cuales uno de los operandos es accedido al revés, tal y como se han implementado algunas de las operaciones descritas anteriormente.

Por ello con especial cuidado en la sincronización y tratando de evitar accesos ineficientes a memoria, este algoritmo puede reescribirse en función de estas operaciones básicas quedando de la siguiente manera:

Algoritmo 14 Implementación del algoritmo Levinson en la GPU

1: Sean $\alpha, \mu \in \mathbb{R}$ escalares en memoria compartida.
2: Sean tn_{cache} y b_{cache} bloques de dimensión $half\ warp$ en memoria compartida.
3: Normalizar accediendo por bloques: $tn = t/t_0, \quad b = b/t_0$
4: Cargar primeros $half\ warp$ datos de tn y b a tn_{cache} y b_{cache}.
5: Inicializar $x_0 = b_{cache_0}, \quad y_0 = -tn_{cache_0}, \quad \alpha = -tn_{cache_0}, \quad \beta = 1$
6: **for** $k = 1$ con $k < n$ **do**
7: Sea $kh = (k \mod half\ warp)$
8: Si $kh = 0$ cargar siguientes $half\ warp$ datos a tn_{cache} y b_{cache}.
9: $dot\_result =$ **dot_reverse_y**(k, tn, x)
10: Si $tid = 0$: $\quad \beta = (1 - \alpha^2) \cdot \beta, \quad \mu = (b_{cache_{kh}} - dot\_result)/\beta$
11: Sincronizar hilos.
12: **axpy_reverse_x**(k, μ, y, x)
13: Si $tid = kh$: $\ x_k = \mu$
14: Sincronizar hilos.
15: **if** $k < n - 1$ **then**
16: $dot\_result =$ **dot_reverse_y**(k, tn, y)
17: Si $tid = 0$: $\quad \alpha = -(tn_{cache_{kh}} + dot\_result)/\beta$
18: Sincronizar hilos.
19: $y =$ **axpxb_reverse_x**$(k, \alpha, y, 1)$
20: Si $tid = kh$: $\ y_k = \alpha$
21: **end if**
22: Sincronizar hilos.
23: **end for**

A pesar de todo, el elevado número de accesos a memoria comentado anteriormente junto con las operaciones extra que los cálculos vectoriales básicos realizan para mantener los accesos de tipo *coalesced* a vectores al revés son candidatos a afectar gravemente al rendimiento de este algoritmo tal y como se implementa. Este punto se analizará en mayor detalle en la sección 4.

La implementación en CUDA de este último algoritmo puede encontrarse en el archivo *toeplitz_kernel.cu* del código fuente, así como en el apéndice A de esta tesis.

3.5. Shift and Invert 2-way Lanczos

Finalmente aquí se analizará el algoritmo principal, dedicado a la extracción paralela de los valores de las matrices tridiagonales en la GPU para la posterior aproximación de valores propios en la CPU.

Debe tenerse en cuenta que a diferencia del algoritmo original descrito en 1 éste está diseñado tanto para crear subespacios de Krylov como para incrementarlos. Dicho de otra manera, está adaptado para continuar aumentando un subespacio dado a partir de su k-ésima iteración según las circunstancias de convergencia.

Este proceso ha sido descrito a grandes rasgos en el algoritmo 4, si bien falta precisarlo en mayor detalle y en función de los métodos descritos anteriormente.

De esta forma el algoritmo 4, siendo $k_0 \geq 0$ el tamaño inicial del subespacio, quedaría implementado de la siguiente manera:

Algoritmo 15 Implementación del algoritmo Shift and Invert 2-way Lanczos en la GPU

1: Sea n_{align} el menor múltiplo de $blockSize \geq n$.
2: Sean $w, v, va, vs \in \mathbb{R}^n$ vectores auxiliares en memoria global.
3: Sean $\alpha, \beta, \gamma, \delta \in \mathbb{R}$ escalares en memoria compartida.
4: Si $k_0 = 0$ inicializar p_0 y q_0 con valores simétricos y antisimétricos normalizados, sino leer valores de β y γ de la iteración anterior.
5: Sea $max_{it} = \text{mín}(max_k, min_k + \triangle k)$.
6: **for** $k = min_k + 1$ hasta max_{it} **do**
7: Sumar procesando por bloques: $w = p_{k-1} + q_{k-1}$
8: Resolver sistema $toeplitz(t') \cdot v = w$ usando **levinson** con $t' = t_0 - \sigma, t_1, t_2 \ldots t_{n-1}$
9: Calcular vectores simétrico y antisimétrico vs y va:
$vs = \textbf{axpxb\_reverse\_x}(n, +1, v, \frac{1}{2}), \quad va = \textbf{axpxb\_reverse\_x}(n, -1, v, \frac{1}{2})$
10: Calcular elementos de las diagonales principales α y γ:
$\alpha = \textbf{dot}(n, vs, p_{k-1}), \quad \gamma = \textbf{dot}(n, va, q_{k-1})$
11: **if** $k < max_k$ **then**
12: $\textbf{axpy}(n, -\alpha, p_{k-1}, vs), \quad \textbf{axpy}(n, -\gamma, q_{k-1}, va)$
13: Si $k > 1$: $\textbf{axpy}(n, -\beta, p_{k-2}, vs), \quad \textbf{axpy}(n, -\delta, q_{k-2}, va)$
14: Reortogonalización completa:
$\textbf{reorthogonalize}(n, k, p, vs), \quad \textbf{reorthogonalize}(n, k, q, va)$
15: Calcular los elementos de las subdiagonales β y δ:
$\beta = \|vs\|_2, \quad \delta = \|va\|_2$
16: Escalar procesando por bloques: $p_k = vs/\beta, \quad q_k = va/\delta$
17: **end if**
18: Almacenar $\alpha, \beta, \gamma, \delta$ en memoria global.
19: **end for**

Al igual que ocurría con el algoritmo anterior, su implementación puede encontrarse en el archivo *toeplitz_kernel.cu* del código fuente. Su equivalente secuencial para CPU se encuentra definido en el archivo *cpu_eigen.c*.

Debe tenerse especial cuidado seleccionando el parámetro $\triangle k$ puesto que aunque incrementa el trabajo realizado por la GPU por iteración conviene recordar que todos los cálculos de convergencia de todos los intervalos asignados a los múltiples dispositivos se procesan secuencialmente en la CPU de forma solapada a la ejecución de los *kernels*.

En la actualidad esta comprobación no se encuentra paralelizada en múltiples hilos al no resultar necesario: los cálculos de la GPU requieren más tiempo que los de convergencia. No obstante el incremento de $\triangle k$ podría trasladar el cuello de botella de la ejecución de un sitio a otro, haciendo inútiles las optimizaciones en el código de CUDA.

4. Resultados obtenidos

Los resultados obtenidos muestran notables complicaciones en diversos factores tales como velocidad, precisión y número de valores propios extraídos con éxito. Estos factores serán analizados por separado analizando las causas y consecuencias de cada uno de ellos y en la medida de lo posible planteando soluciones.

Las pruebas se han realizado en un ordenador con procesador Intel Core Duo a 2.33 GHz cada núcleo con un total de 2 GB de memoria RAM. El hardware gráfico empleado, como ya se ha indicado en secciones anteriores, es una GeForce 9800 GX2 cuyas características principales pueden verse en la tabla 1.

Todas las pruebas aquí mostradas han empleado matrices Toeplitz simétricas generadas aleatoriamente de forma $t \in \mathbb{R}^n, t_i \in [-1, 1]$.

De igual forma salvo que se indique lo contrario los resultados corresponden a los valores por defecto de 2 valores propios por intervalo, $\triangle k = 2$, un dígito de precisión ($tolerancia = 10^{-1}$), hasta 5 intentos por intervalo y un tamaño máximo para el subespacio de Krylov de 10 (5× valores propios contenidos).

4.1. Resultados de tiempos y aceleración

En primer lugar se analizarán los resultados de tiempos obtenidos y aceleración respecto a la versión secuencial de CPU. Para ello se muestra una tabla con los tiempos de estas dos versiones según el número máximo de valores propios por intervalo.

Talla \ λ	Versión CPU					Versión GPU				
	2	5	10	50	100	2	5	10	50	100
$n = 600$	7.19	12.29	13.80	3.81	2.20	8.57	11.36	10.34	3.22	2.31
$n = 800$	17.48	31.35	30.24	7.50	4.48	23.67	25.73	23.85	5.65	4.09
$n = 1000$	43.21	61.47	59.74	13.89	7.89	69.40	56.00	45.72	9.98	6.79
$n = 1200$	118.62	118.65	114.36	26.48	14.14	175.87	115.26	80.08	20.70	11.20
$n = 1400$	292.81	227.22	176.02	42.56	23.44	370.63	221.95	135.26	32.14	17.73
$n = 1600$	613.82	408.00	269.30	64.14	33.26	679.21	362.94	202.14	46.96	24.13
$n = 1800$	1136.83	681.29	414.91	88.26	44.35	1011.90	506.67	287.67	62.18	32.74
$n = 2000$	1653.63	934.26	555.39	130.59	68.80	1409.37	731.46	406.04	88.83	50.61

Cuadro 5: comparación de tiempos (seg) según valores propios por intervalo.

Dado que los tiempos para tallas pequeñas no resultan especialmente relevantes o significativos se ha optado por incluir únicamente tiempos a partir de $n = 600$. Del mismo modo, por motivos relacionados con la precisión que se justificarán en la próxima sección no se ha considerado necesario extraer datos para tallas superiores.

A partir de los valores de la tabla anterior se obtiene la siguiente gráfica de speed-ups:

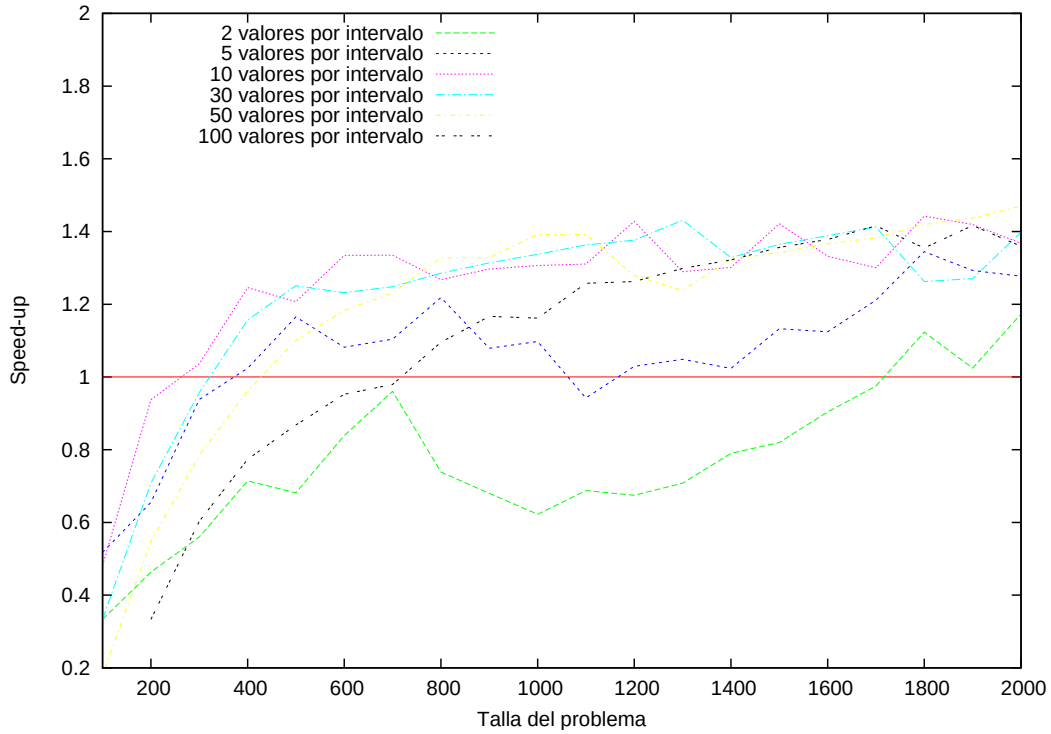


Figura 11: speed-ups obtenidos para diferente número de valores propios por intervalo.

Debe recordarse que por limitaciones de la precisión disponible tal y como se discutió en la sección 3.1.3 se utiliza por defecto un máximo de tan sólo 2 valores propios por intervalo. Las consecuencias respecto a la eficiencia en la extracción de elegir cada una de estas posibilidades se analizarán en la sección siguiente.

En cualquier caso y limitándose al aspecto de tiempos y speed-ups se puede observar como para el primer caso de 2 valores propios por intervalo la aceleración es inexistente durante la mayor parte de las tallas, comenzando a apreciarse únicamente a partir de aproximadamente $n = 1700$. Aumentando esta primera cantidad tan solo un poco se consigue una notable mejora de tiempos respecto a esta primera aproximación logrando obtener ya aceleraciones con el hardware gráfico.

No obstante estos valores son demasiado pobres para lo que cabría esperar: de hecho ni siquiera en el caso más optimista se alcanzan aceleraciones de 1,5x. Se han realizado diversas tareas de *profiling* tratando de localizar el cuello de botella del problema, verificando que no existe ni un solo acceso a memoria de tipo *uncoalesced* en toda la resolución del problema para cualquier talla o entrada de datos probados. No obstante la herramienta *CUDA Visual Profiler* muestra un muy elevado número de accesos a memoria por parte del algoritmo de Levinson consumiendo éste de media cerca del 99 % del tiempo de ejecución de la GPU.

Recordemos que este algoritmo evita la construcción explícita de la matriz en memoria, pero esta ventaja no es gratuita. Volviendo a la implementación descrita en el algoritmo 14 puede verse que internamente se compone de las operaciones básicas *dot_reverse_y*,

axpy_reverse_x y *axpxb_reverse_x*. Estas operaciones tal y como se definen en la sección 3.3 acceden a un total de $2n$, $3n$ y $2n$ elementos en memoria global respectivamente.

Teniendo en cuenta este número de accesos en las operaciones básicas así como $4n$ accesos extra a elementos para la normalización de los vectores t y b, y aproximadamente $2 \times 16 \times \lceil \frac{n}{16} \rceil$ para la caché de valores, asumiendo un aprovechamiento máximo de las lecturas y escrituras es posible calcular el número de accesos a elementos de memoria global para resolver un sistema Toeplitz simétrico de dimensión n:

$$\sum_{k=1}^{n-1}(5k + 4k) - 4(n-1) + 6n = \frac{9n^2 + 4n + 8}{2} \tag{8}$$

Si bien encontrar que el número de accesos a elementos es de orden $\mathcal{O}(n^2)$ resulta algo previsible no lo es tanto encontrar el elevado factor que lo multiplica: 9. Este factor, consecuencia de los continuos accesos a los vectores completos por parte de las operaciones básicas, amplifica innecesariamente la cantidad de accesos a memoria atacando justamente en el principal cuello de botella de CUDA.

Además de este problema se encuentran las operaciones adicionales necesarias para mantener los accesos de tipo *coalesced* a los vectores al revés. Éstas no son realizadas por la versión de CPU, limitándose en la mayoría de los casos a acceder a datos en una caché que la GPU no tiene. Esta sobrecarga adicional también contribuye a la notable ineficiencia de este algoritmo en hardware gráfico.

Cabe resaltar que dada la implementación del algoritmo 14 las previsiones descritas en la ecuación 8 son todavía optimistas, puesto que de forma cíclica cada 16 iteraciones del bucle interno el número de hilos que participarán en los accesos a memoria realizados por las últimas iteraciones de las operaciones básicas oscilará de tan sólo uno, desaprovechando notablemente el ancho de banda, a los 16 hilos del *half warp*. Esta es una consecuencia de la estructura iterativa e incremental del propio algoritmo contra la que poco se puede hacer.

4.2. Resultados de precisión y eficiencia en la extracción

Tal y como se describió en la sección 3.1.3, el uso de la coma flotante de simple precisión con las peculiaridades de su implementación en el hardware tiene un gran impacto en el proceso de extracción afectando notablemente a la cantidad y a la calidad de los valores propios extraídos.

Además, estos problemas se acrecientan con la talla del problema debido a la pérdida adicional de precisión producida en algunos acumuladores, como pueden ser los utilizados en los cálculos de producto escalar. Éstos se ven especialmente acentuados con el uso interno de instrucciones *FMAD* que combinan multiplicaciones y sumas consiguiendo un mayor rendimiento a costa de truncar el resultado intermedio de la multiplicación.

De esta forma el porcentaje de valores propios correctamente extraídos en función de los distintos niveles de precisión requerida son los siguientes:

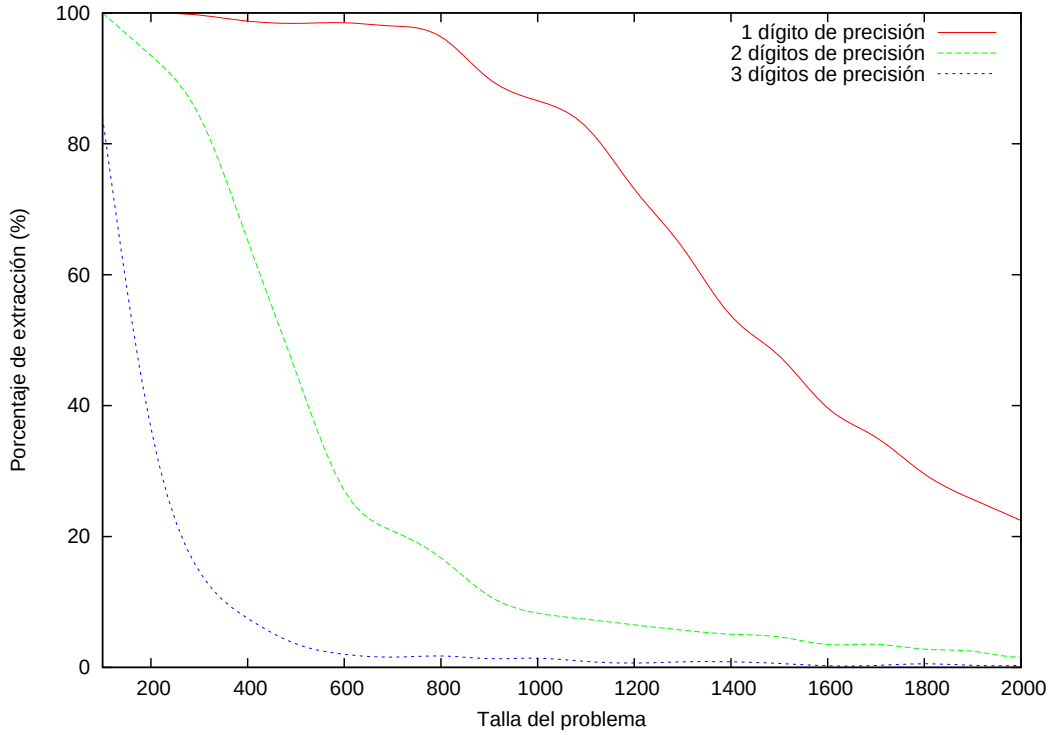


Figura 12: porcentaje de valores propios extraídos en función de la precisión solicitada.

Un valor propio falla en el proceso de ser extraído si no se alcanza la convergencia del intervalo que lo contiene en ninguno de los intentos realizados, por defecto 5 veces más la inicial. Además tampoco deberá ser hallado en caso de convergencia parcial (sólo de algunos valores del intervalo).

Como puede apreciarse para tallas de $n = 2000$ el número de valores propios correctamente extraídos, incluso con una tolerancia tan baja como es la de un sólo dígito de precisión, es tan pequeño que analizar casos de tallas mayores simplemente no resulta útil. No se debe olvidar que el objetivo original de esta implementación es la extracción de todos los valores propios de una matriz Toeplitz simétrica dada. Una situación en la que falla la extracción del 70 % de estos valores simplemente se encuentra fuera de lugar.

Se han realizado multitud de pruebas al respecto tratando de mejorar la cantidad de valores extraídos. La primera de ellas fue aumentar el tamaño máximo del subespacio de Krylov a, por ejemplo, 15 y 20 veces el número de valores propios que contenía con el consecuente crecimiento de los requisitos de memoria. Las pruebas mostraron que incrementar el tamaño del subespacio más allá de los valores por defecto no mejoró de forma apreciable el número de valores propios extraídos.

Otra prueba realizada fue aumentar el número máximo de intentos por intervalo. En este caso sí había un ligero incremento del número de valores propios extraídos, pero el esfuerzo resultaba totalmente contraproducente puesto que la gran mayoría aquellos intervalos que no convergían seguían sin hacerlo solo que ocupando durante más tiempo más recursos y ralentizando notablemente la ejecución.

El último de los intentos realizados para mejorar la convergencia y la extracción fue precisamente la selección de un número máximo de dos valores propios por intervalo, siendo éste el caso con peores prestaciones de la sección anterior.

Volviendo al caso más estable de un único dígito significativo, si se analiza el comportamiento de los distintos intervalos en función del número de valores propios que contienen y de la talla del problema se obtienen la siguiente gráfica:

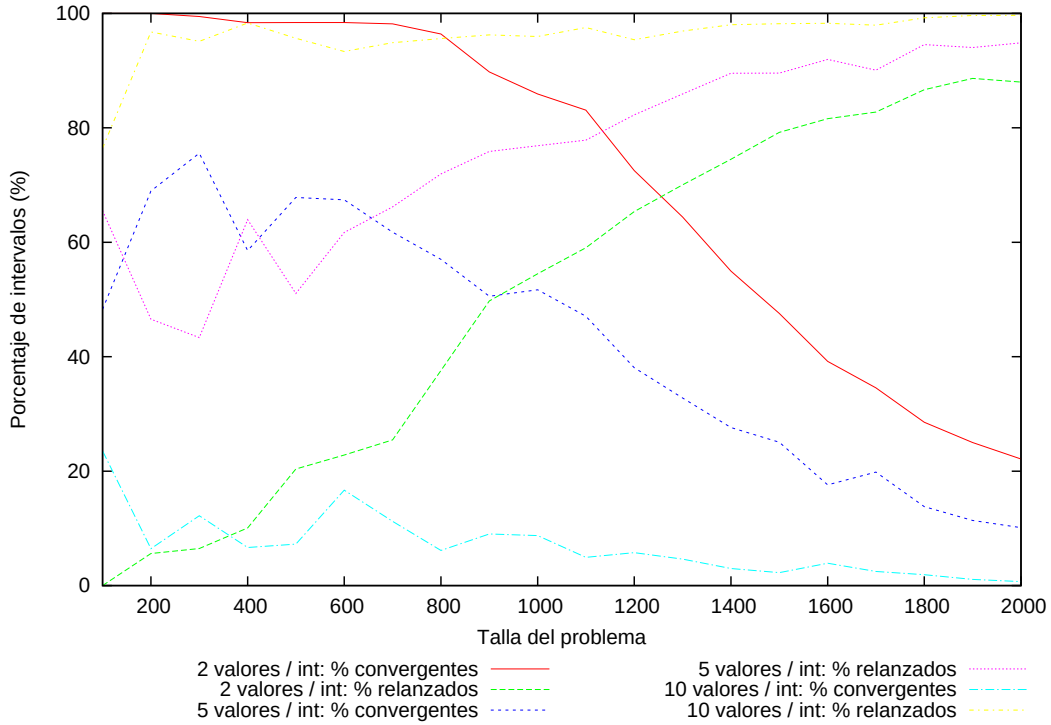


Figura 13: porcentaje de intervalos convergentes y relanzados según la talla del problema.

Puede verse claramente como al aumentar aunque sea un poco el número de valores propios por intervalo la cantidad de reintentos se dispara de la misma forma que el número de intervalos que alcanzan la convergencia cae drásticamente.

Este hecho parece implicar una desafortunada situación: aquellos parámetros que hacen al algoritmo eficiente también destrozan su convergencia por la precisión insuficiente de coma flotante utilizada, justificando los valores por defecto seleccionados a pesar de su ineficiencia en pro de su funcionalidad.

Debe tenerse también en cuenta que curiosamente las mayores eficiencias se dan en cantidades de valores propios por intervalo que, según se acaba de mostrar, realizan un mayor número de cálculos al relanzar varias veces prácticamente todos sus intervalos. Este hecho podrá entenderse mejor al observar la siguiente gráfica.

Así pues, de forma adicional a los datos anteriores también puede representarse el número medio de reintentos para cada situación.

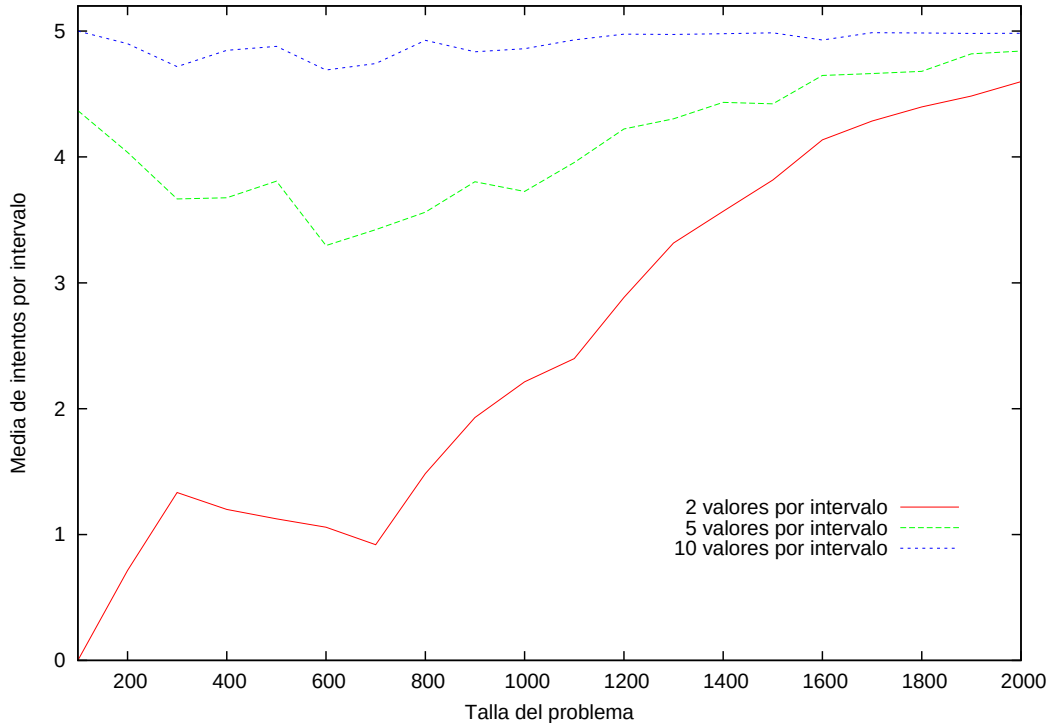


Figura 14: número medio de intentos por intervalo en función de la talla.

Como es de esperar, el número medio de intentos se aproxima rápidamente a 5, el máximo por defecto, según avanza el número de intervalos que no alcanzan la convergencia. No obstante estos datos permiten hacerse una idea de la cantidad de relanzamientos y el tiempo extra de computación que ellos implican, aunque éste está acotado debido a la limitación en el número de relanzamientos.

Es esta limitación la que evita que las ejecuciones con un mayor número de valores propios por intervalo sean ineficientes. Es más, al aumentar el número de valores propios por intervalo y consecuentemente reducir la cantidad total de intervalos a evaluar se está implícitamente reduciendo el número de relanzamientos con el notable ahorro que ello conlleva.

Ésta podría ser pues una buena explicación de los tiempos descritos en la tabla 5.

5. Conclusiones y trabajos futuros

A lo largo del documento se ha presentado una alternativa al método de extracción de valores propios de matrices Toeplitz simétricas descrito en [13], adaptándolo a las circunstancias de la programación en GPUs según el modelo de CUDA.

Estas adaptaciones así como las decisiones de diseño tomadas en consecuencia se han visto principalmente afectadas por las circunstancias y limitaciones del hardware empleado, una tarjeta gráfica GeForce 9800 GX2 con soporte para coma flotante de simple precisión y las limitaciones del acceso a memoria que CUDA impone para los dispositivos con capacidad computacional versión 1.1.

De entre los cambios realizados destaca la necesidad de emplear el algoritmo de Levinson-Durbin para resolver sistemas Toeplitz en lugar de la aproximación por matrices tipo Cauchy descrita en [13] y [1]. Ésta elección se ha visto motivada por las limitaciones de memoria gráfica del dispositivo que impedían almacenar las matrices resultantes de la descomposición LDL^T permitiendo así reducir el coste espacial a tan sólo $\mathcal{O}(n)$ elementos.

No obstante el empleo de este algoritmo tiene sus consecuencias: puede ser numéricamente inestable cuando las matrices de los sistemas a resolver no son simétricas definidas positivas, hecho bastante habitual dadas las circunstancias donde se emplea, y además realiza un notable número de accesos a memoria del orden de $\mathcal{O}(9n^2)$. Estos accesos a memoria atacan justamente en el mayor cuello de botella de CUDA, los accesos a memoria global.

Además, la implementación del algoritmo de Levinson requiere llevar a cabo diversas operaciones en las que se operan con vectores a los que se accede desde atrás hacia delante. Como consecuencia las restricciones en el acceso a memoria del hardware llevan a realizar numerosas operaciones adicionales para mantener los llamados accesos de tipo *coalesced* y evitar pérdidas catastróficas de eficiencia. Este tipo de operaciones no necesitan ser realizadas por la versión secuencial en la CPU, la cual se limitará en la mayoría de los casos a acceder a datos en memoria caché.

Todos estos hechos contribuirán a que la implementación del algoritmo de Levinson en el hardware utilizado resulte poco eficiente siendo de hecho el cuello de botella del método aquí presentado.

Adicionalmente a los problemas derivados de las limitaciones de memoria y del propio algoritmo de Levinson también se han encontrado notables problemas de precisión. En concreto, las limitaciones de la coma flotante de simple precisión soportada por el hardware utilizado han llevado a obtener, tal y como se ha visto, unos paupérrimos resultados de convergencia y precisión en los valores extraídos.

Estos problemas de precisión producen numerosos relanzamientos de intervalos que a su vez ralentizan considerablemente los tiempos de ejecución, evitando también poder ajustar otros parámetros que mejorarían la eficiencia como es el caso del número de valores propios por intervalo.

A todo esto se añaden los problemas de reducción de la precisión conforme aumenta la talla del problema. Éstos impiden escalar el algoritmo a tallas mayores donde se obtienen mejores prestaciones temporales debido a la pérdida casi total de la convergencia.

En definitiva, este trabajo ha tratado de aprovechar las prestaciones de un hardware gráfico para calcular eficientemente los valores propios de una matriz Toeplitz simétrica intentando superar las limitaciones implícitas a éste. Por desgracia estas limitaciones han resultado demasiado poderosas de cara a la obtención de buenos resultados y de una eficiencia a la altura de las expectativas.

No obstante sería un error pensar que las GPUs son inadecuadas para estas tareas en base a los resultados obtenidos. Nada más alejado de la realidad. Los últimos dispositivos presentados han dado ya el salto a la doble precisión y han eliminado la gran mayoría de las restricciones que han afectado negativamente al diseño y a la implementación del método aquí presentado.

Estos nuevos dispositivos al disponer de mayor cantidad de memoria permitirían replantear la decisión de utilizar el algoritmo de Levinson volviendo a la idea original de resolver sistemas triangulares de matrices tipo Cauchy, lo que podría hacerse eficientemente realizando aproximadamente $\mathcal{O}(n^2)$ accesos a memoria en lugar de los $\mathcal{O}(9n^2)$ del algoritmo de Levinson.

Por otra parte y aunque probablemente no sea lo más adecuado, se podría tratar de continuar con la línea de este algoritmo adaptando algunas de las técnicas propuestas por [3] y [7] para eliminar los problemas de inestabilidad numérica aprovechando también la eliminación de la mayoría de restricciones del acceso a memoria. En este caso las operaciones básicas aquí descritas podrían implementarse de forma mucho más simple ya que no existiría especial problema en acceder a los vectores en orden inverso.

En cuanto a los problemas de convergencia el uso de la coma flotante de doble precisión mejoraría notablemente la situación actual permitiendo procesar intervalos con un mayor número de valores propios, obteniendo unos mejores márgenes de precisión y pudiendo extender en la práctica el problema a tallas mayores donde realmente pueda ser de utilidad. Además se reduciría notablemente el tiempo de ejecución al disminuir el número de intervalos relanzados.

Por otra parte, sería también interesante plantear los cálculos de convergencia que realiza la CPU de forma multihilo o incluso, en la medida de lo posible, trasladar parte de éstos a la propia GPU mediante *kernels* adicionales.

Así pues, se espera que la experiencia extraída de la realización de esta tesis sirva para mejorar los resultados obtenidos convergiendo en la realización de un artículo de investigación donde se detallen las conclusiones finales del trabajo aquí iniciado. Así mismo se espera que esta tesis pueda servir como primer paso para el desarrollo de una nueva línea de investigación basada en el aprovechamiento de las prestaciones del hardware gráfico, que ofrece un potente rendimiento a un coste realmente bajo en comparación con otras soluciones tradicionales como el uso de clusters.

Por último, quisiera agradecer la confianza depositada por los profesores del máster en Computación Paralela y Distribuida de la Universidad Politécnica de Valencia en mis proyectos de computación de propósito general en GPUs, y en especial la inestimable ayuda y paciencia de mi director de tesis Antonio Manuel Vidal Maciá sin cuyo apoyo, referencias y concisas explicaciones esta tesis no habría sido posible.

A. Apéndice: código fuente de los *kernels*

A continuación se muestra el código fuente en CUDA de los distintos métodos descritos en las secciones 3.3, 3.4 y 3.5. La documentación generada para estos métodos puede encontrarse en el directorio *doc* del proyecto en los formatos HTML y LaTeX. Esta última ha sido también exportada con el nombre de *User Manual.pdf* [6] al directorio raíz del proyecto.

Cabe resaltar que la macro T se utiliza como macro para definir el tipo de datos de la coma flotante utilizada, $float$ por defecto dadas las circunstancias del hardware.

Del mismo modo la macro $blockSize$ se evalúa al número de hilos por bloque establecido en 256 y $align(n)$ devuelve el menor múltiplo de éste $>= n$.

Código fuente de *axpy*

```
/**
 * Calculates the dot product of x and y vectors.
 *
 * \param n Size of x and y vectors.
 * \param x First vector.
 * \param y Second vector.
 * \param swork Shared memory workspace. Requires blockSize floats.
 * \return Thread 0 of the block returns the dot product result.
*/
__device__ void axpy(unsigned int n, T a, const T *x, T *y) {

   // Blocksize should be aligned to half_warp
   unsigned int tid = threadIdx.x;

   // Process vectors in blocks
   for(unsigned int b=0; b<n; b+=blockSize) {

      // Calculate per-block axpy (all memory accesses should be aligned)
      unsigned int i = b + tid;
      if(i < n) y[i] += a * x[i];
   }

   __syncthreads();
}
```

Código fuente de *norm*

```
/**
 * Calculates the 2-norm (length) of a given vector.
 * Should be faster than sqrt(dot(x, x)) since reads each component only once.
 *
 * \param n Size of x vector.
 * \param x Input vector.
 * \param swork Shared memory workspace. Requires blockSize floats.
 * \return Thread 0 of the block returns the dot product result.
*/
__device__ T norm(unsigned int n, const T *x, T *swork) {

   // Allocate temporary blocks in shared memory and calculate thread id
```

```
   unsigned int tid = threadIdx.x;
   T *br = swork; // Temporary result block (size: blockSize)
   T result = (T)0.0;

   // Process vector in blocks
   for(unsigned int b=0; b<n; b+=blockSize) {
      unsigned int i = b + tid;

      // Calculate per-block sum of squares
      if(i < n) {
         T aux = x[i];
         br[tid] = aux * aux;
      } else br[tid] = (T)0.0;
      __syncthreads();

      // Sum temporary result elements
      // Speed-up hack: some reductions are done in parallel,
      //                while small sets are reduced sequentially
      int s = blockSize;
      while(s > 8) {
         s >>= 1;
         if(tid < s) br[tid] += br[tid + s];
         __syncthreads();
      }

      if(tid == 0) {
         for(unsigned int j=0; j<s; ++j)   result += br[j];
      }
      __syncthreads();
   }

   if(tid == 0) {
      #ifdef SINGLE
      result = sqrtf(result);
      #else
      result = sqrt(result);
      #endif
   }
   __syncthreads();

   // Thread 0 returns the correct value
   return result;
}
```

Código fuente de *dot*

```
/**
 * Calculates the dot product of x and y vectors.
 *
 * \param n Size of x and y vectors.
 * \param x First vector.
 * \param y Second vector.
 * \param swork Shared memory workspace. Requires blockSize floats.
 * \return Thread 0 of the block returns the dot product result.
*/
__device__ T dot(unsigned int n, const T *x, const T *y, T *swork) {

   // Allocate temporary blocks in shared memory and calculate thread id
```

```
   unsigned int tid = threadIdx.x;
   T *br = swork; // Temporary result block (size: blockSize)
   T result = (T)0.0;

   // Process vector in blocks
   for(unsigned int b=0; b<n; b+=blockSize) {
      unsigned int i = b + tid;

      // Calculate per-block dot product
      if(i < n) br[tid] = x[i] * y[i];
      else br[tid] = (T)0.0;
      __syncthreads();

      // Sum temporary result elements
      // Speed-up hack: some reductions are done in parallel,
      //                while small sets are reduced sequentially
      int s = blockSize;
      while(s > 8) {
         s >>= 1;
         if(tid < s) br[tid] += br[tid + s];
         __syncthreads();
      }

      if(tid == 0) {
         for(unsigned int j=0; j<s; ++j)   result += br[j];
      }
      __syncthreads();
   }

   // Thread 0 returns the correct value
   return result;
}
```

Código fuente de *dot_reverse_y*

```
/**
 * Calculates the dot product of x and reverse(y) vectors.
 *
 * \param n Size of x and y vectors.
 * \param x First vector.
 * \param y Second vector.
 * \param swork Shared memory workspace. Requires 4 * blockSize - 1 floats.
 * \return Thread 0 of the block returns the dot product result.
*/
__device__ T dot_reverse_y(unsigned int n, const T *x, const T *y, T *swork) {

   // Allocate temporary blocks in shared memory
   T *bx = swork;                   // Forward vector block (size: 2 * blockSize - 1)
   T *by = bx + blockSize * 2 - 1;  // Backward vector block (size: blockSize)
   T *br = by + blockSize;          // Temporary result block (size: blockSize)

   // Calculate thread id and aligned vector size
   unsigned int tid = threadIdx.x;
   unsigned int n_align = align(n);
   unsigned int shift = n_align - n;
   T result = 0.0;

   // Reset shifted contents of forward vector
```

```
   bx[tid] = (T)0.0;
   __syncthreads();

   // Process vectors in blocks
   for(unsigned int i=0; i<n; i+=blockSize) {

      // Read forward and reverse blocks in parallel into shared memory
      unsigned int xi = i + tid;
      unsigned int yi = n_align - i - blockSize + tid;

      if(xi < n) bx[shift + tid] = x[xi];
      if(yi < n) by[tid] = y[yi]; else by[tid] = (T)0.0;
      __syncthreads();

      // Calculate per-block dot product
      br[tid] = bx[tid] * by[blockSize - tid - 1];
      __syncthreads();

      // Shift contents of forward block
      if(tid < shift) bx[tid] = bx[tid + blockSize];

      // Sum temporary result elements
      // Speed-up hack: some reductions are done in parallel,
      //                while small sets are reduced sequentially
      int s = blockSize;
      while(s > 8) {
         s >>= 1;
         if(tid < s) br[tid] += br[tid + s];
         __syncthreads();
      }

      if(tid == 0) {
         for(unsigned int j=0; j<s; ++j)   result += br[j];
      }
      __syncthreads();
   }

   // Thread 0 returns the correct value
   return result;
}
```

Código fuente de *axpy_reverse_x*

```
/**
 * Performs y = a * reverse(x) + y operation.
 *
 * \param n Size of x and y vectors.
 * \param a Scalar factor.
 * \param x First vector.
 * \param y Second vector, modified with result. Cannot be x.
 * \param swork Shared memory workspace. Requires 5 * blockSize - 1 floats.
*/
__device__ void axpy_reverse_x(unsigned int n, T a, const T *x, T *y, T *swork) {

   // Allocate temporary blocks in shared memory
   T *bx = swork;                   // Backward vector block  (size: blockSize)
   T *by = bx + blockSize;          // Forward vector block   (size: 2 * blockSize - 1)
   T *br = by + 2 * blockSize - 1;  // Temporary result block (size: 2 * blockSize)
```

```
   // Calculate thread id and aligned vector size
   unsigned int tid = threadIdx.x;
   unsigned int n_align = align(n);
   unsigned int shift = n_align - n;
   unsigned int lastblock = blockSize - shift;

   // Process vectors in blocks
   for(unsigned int i=0; i<n; i+=blockSize) {

      // Read forward and reverse blocks in parallel into shared memory
      unsigned int xi = n_align - i - blockSize + tid;
      unsigned int yi = i + tid;

      if(xi < n) bx[tid] = x[xi];
      if(yi < n) by[shift + tid] = y[yi];
      __syncthreads();

      // Calculate per-block axpy
      if(i > 0) {
         // Append to previous temporary results
         br[tid + lastblock] = a * bx[blockSize - tid - 1] + by[tid];
      } else {
         // Skip shift gap and put contents at block start
         if(tid >= shift) br[tid - shift] = a * bx[blockSize - tid - 1] + by[tid];
      }
      __syncthreads();

      // Store temporary results
      if(i > 0) y[yi - blockSize] = br[tid];

      // Shift contents of forward block
      if(tid < shift) by[tid] = by[tid + blockSize];

      // Shift contents of result block
      if(i > 0 && tid < lastblock) br[tid] = br[tid + blockSize];
      __syncthreads();
   }

   // Store remaining contents of temporary result block
   unsigned int yi = n_align - blockSize + tid;
   if(yi < n) y[yi] = br[tid];
   __syncthreads();
}
```

Código fuente de *axpxb_reverse_x*

```
/**
 * Performs x = b * (a * reverse(x) + x) operation.
 *
 * \param n Size of x vector.
 * \param a Scalar factor.
 * \param x Input vector.
 * \param y Output vector. Can be x.
 * \param b Result scaling value.
 * \param swork Shared memory workspace. Requires 6 * blockSize - 1 floats.
*/
__device__ void axpxb_reverse_x(unsigned int n, T a, const T *x, T *y, T b, T *swork) {
```

```
// Allocate temporary blocks in shared memory
T *bb = swork;                    // Backward vector block (size: blockSize)
T *bf = bb + blockSize;           // Forward vector block  (size: 2 * blockSize - 1)
T *brf = bf + blockSize * 2 - 1;  // Temporary forward result block  (size: blockSize * 2)
T *brb = brf + blockSize * 2;     // Temporary backward result block (size: blockSize)

// Calculate thread id and aligned vector size
unsigned int tid = threadIdx.x;
unsigned int n_align = align(n);
unsigned int shift = n_align - n;
unsigned int lastblock = blockSize - shift;

// Process vectors in blocks
unsigned int num_blocks = n_align / blockSize;
unsigned int max_n = blockSize * (1 + (num_blocks >> 1));
for(unsigned int i=0; i<max_n; i+=blockSize) {

   // Read forward and reverse blocks in parallel into shared memory
   unsigned int bi = n_align - i - blockSize + tid;
   unsigned int fi = i + tid;

   if(bi < n) bb[tid] = x[bi];
   if(fi < n) bf[shift + tid] = x[fi];
   __syncthreads();

   // Calculate per-block axpxb
   if(i > 0) {
      // Append to previous temporary results
      brf[tid + lastblock] = b * (a * bb[blockSize - tid - 1] + bf[tid]);

      // Backward results doesn't need any appending
      brb[tid] = b * (bb[tid] + a * bf[blockSize - tid - 1]);

   } else {
      // Skip shift gap and put contents at block start
      if(tid >= shift) brf[tid - shift] = b * (a * bb[blockSize - tid - 1] + bf[tid]);

      // Store backward results at block start
      if(tid < lastblock) brb[tid] = b * (bb[tid] + a * bf[blockSize - tid - 1]);
   }
   __syncthreads();

   // Store temporary results from forward result block
   if(i > 0) y[fi - blockSize] = brf[tid];

   // Skip writing last backward result in last iteration if number of blocks is even
   if(i + blockSize >= max_n && !(num_blocks & 1)) break;

   // Store temporary results from backward result block
   if(bi < n) y[bi] = brb[tid];

   // Avoid unnecessary shifting in last iteration
   if(i + blockSize >= max_n) break;

   // Shift contents of forward block
   if(tid < shift) bf[tid] = bf[tid + blockSize];
```

```
      // Shift contents of forward result block
      if(i > 0 && tid < lastblock) brf[tid] = brf[tid + blockSize];
      __syncthreads();
   }
   __syncthreads();
}
```

Código fuente de *reorthogonalize*

```
/**
 * Reorthogonalize a vector to a Krylov subspace base.
 *
 * \note This function requires blockSize >= half_warp.
 *
 * \param n Size of v vector and number of rows of p matrix.
 * \param k_size Current size of Krylov subspace. Number of columns of p matrix.
 * \param p Pointer to a column-ordered matrix defining the base of the Krylov subspace.
 * \param v Pointer to the vector to be reorthogonalized.
 * \param h Work array of size k_size.
 * \param swork Shared memory workspace. Requires 2 * blockSize floats.
*/
__device__ void reorthogonalize(unsigned int n, unsigned int k_size, const T *p,
   T *v, T *h, T *swork) {

   // Calculate thread identifier and define block size
   unsigned int tid = threadIdx.x;
   unsigned int n_align = align(n);

   // Calculate h = p' * v
   const T *pk = p;
   for(unsigned int bk=0; bk<k_size; bk+=half_warp) {

      // Temporary result block for dot products
      __shared__ T dr[half_warp];

      // Calculate h in blocks aligned to optimal memory access
      for(unsigned int w=0; w<half_warp && bk+w<k_size; ++w, pk+=n_align) {

         // Calculate k-th element of h = p' * v
         T aux = dot(n, pk, v, swork);
         if(tid == 0) dr[w] = aux;
         __syncthreads();
      }

      // Store h data block
      unsigned int bi = bk + tid;
      if(tid < half_warp && bi < k_size) h[bi] = dr[tid];
      __syncthreads();
   }

   // Calculate vs -= p * h
   for(unsigned int bn=0; bn<n; bn+=blockSize) {
      unsigned int j = bn + tid;

      // Allocate temporary blocks in shared memory
      T *bh = swork;          // Temporary block for aligned read of h contents (size: blockSize)
      T *ba = bh + blockSize; // Temporary result accumulation block (size: blockSize)
```

```
      // Clear accumulator block
      ba[tid] = (T)0.0;
      __syncthreads();

      // For each k-block of columns...
      for(unsigned int bk=0; bk<k_size; bk+=blockSize) {
         unsigned int k = bk + tid;

         // Read h data into temporary block
         if(k < k_size) bh[tid] = h[k];
         __syncthreads();

         // Use k-th element to scale k-th column of p and add to accumulator
         for(unsigned int ci=0; ci<k_size; ++ci) {
            if(j < n) ba[tid] += bh[ci] * p[ci * n_align + j];
         }
      }
      __syncthreads();

      // Substract accumulator block results to v vector
      if(j < n) v[j] -= ba[tid];
   }

   __syncthreads();
}
```

Código fuente de *levinson*

```
/**
 * Solve a symmetric Toeplitz system of the form toeplitz(t) * x = b
 *
 * \note This function requires blockSize >= half_warp.
 *
 * \param n Size of [td t], x and y vectors (size of Toeplitz matrix).
 * \param td Diagional value of the symmetric Toeplitz matrix.
 * \param t Vector defining the rest of symmetric Toeplitz matrix values.
 * \param b Right-hand side of the system. Will be normalized by td.
 * \param x Vector that will contain the solution of the system.
 * \param work Work vector of size 2 * align(n).
 * \param swork Shared memory workspace. Requires 6 * blockSize - 1 floats.
*/
__device__ void levinson(unsigned int n, T td, const T *t, T *b, T *x, T *work, T *swork) {

   // Calculate thread identifier and define block size
   unsigned int tid = threadIdx.x;
   unsigned int n_align = align(n);

   // Check parameters
   if(n == 0) return;

   // Assign workspace
   T *y  = work;
   T *tn = y + n_align;

   // Initialize auxiliar variables
   __shared__ T b_cache[half_warp], tn_cache[half_warp];
   __shared__ T alpha, mu;
```

```
T beta = (T)1.0;

// Divide t elements by td in parallel
T factor = (T)1.0 / td;
for(unsigned int i=0; i<n-1; i+=blockSize) {
   unsigned int ti = i + tid;
   if(ti < n-1) {
      T aux = t[ti] * factor;
      if(ti < half_warp) tn_cache[ti] = aux;
      tn[ti] = aux;
   }

   __syncthreads();
}

// Divide b elements by td in parallel
for(unsigned int i=0; i<n; i+=blockSize) {
   volatile unsigned int bi = i + tid;
   if(bi < n) {
      T aux = b[bi] * factor;
      if(bi < half_warp) b_cache[bi] = aux;
      b[bi] = aux;
   }

   __syncthreads();
}

// Initialize x, y elements
if(tid == 0) {
   x[0] = b_cache[0];
   y[0] = -tn_cache[0];
   alpha = -tn_cache[0];
}
__syncthreads();

// Main loop
for(int k=1; k<n; ++k) {

   // Reload b & tn vector caches in parallel
   unsigned int k_half_warp = k & half_warp_mask;
   if(k_half_warp == 0) {
      if(tid < half_warp) {
         unsigned int i = k + tid;
         if(i < n) {
            b_cache[tid] = b[i];
            tn_cache[tid] = tn[i];
         }
      }
      __syncthreads();
   }

   // Scalar values are processed by thread 0
   T dot_result = dot_reverse_y(k, tn, x, swork);
   if(tid == 0) {
      beta = ((T)1.0 - alpha * alpha) * beta;
      mu = (b_cache[k_half_warp] - dot_result) / beta;
   }
```

```
      __syncthreads();

      axpy_reverse_x(k, mu, y, x, swork);
      if(tid == k_half_warp) x[k] = mu;
      __syncthreads();

      if(k < n-1) {
         dot_result = dot_reverse_y(k, tn, y, swork);
         if(tid == 0) alpha = -(tn_cache[k_half_warp] + dot_result) / beta;
         __syncthreads();

         axpxb_reverse_x(k, alpha, y, y, (T)1.0, swork);
         if(tid == k_half_warp) y[k] = alpha;
      }

      __syncthreads();
   }
}
```

Código fuente de *si2w*

```
/**
 * Shift and invert 2-way Lanczos routine for Toeplitz matrices.
 * Calculates symmetric and skew-symmetric tridiagonal values,
 * but leaves final eigenvalue calculation to CPU side.
 *
 * \note This function requires blockSize >= half_warp.
 *
 * \param n      Size of the symmetric Toeplitz matrix.
 * \param td     Main diagonal value of the symmetric Toeplitz matrix.
 * \param t      Remaning n-1 values from symmetric Toeplitz matrix.
 * \param min_k  Starting size of the Krylov subspace (starting value 0).
 *               Can be used to resume a previous extraction.
 * \param num_it Number of iterations to perform in the Krylov subspace.
 * \param max_k  Maximum size of the Krylov subspace. Iterations will halt if reached.
 * \param sigma  Shift value for centering eigenvalue extraction (depends on interval).
 * \param p      Symmetric lanczos vectors of Krylov subspace (size align(n) x max_k).
 * \param q      Skew-symmetric lanczos vectors of Krylov subspace (size align(n) x max_k).
 * \param m      Structure array containing values from symmetric and skew-symmetric
 *               tridiagonal matrices (size max_k).
 * \param work   Workspace array of size >= max(5 * align(n), 4 * align(n) + max_k).
*/
__device__ void si2w(unsigned int n, T td, T *t, unsigned int min_k, unsigned int num_it,
   unsigned int max_k, T sigma, T *p, T *q, tridiagonal_entry *m, T *work) {

   // Check params
   if(min_k > max_k || num_it == 0 || n == 0) return;

   // Calculate thread identifier
   volatile unsigned int tid = threadIdx.x;
   volatile unsigned int n_align = align(n);

   // Allocate required shared memory
   extern __shared__ T swork[];
   __shared__ tridiagonal_entry mk;

   // Read only beta and gamma values from previous iteration
   if(min_k > 0) {
```

```
   if(tid < 4) ((T *)&mk)[tid] = ((T *)&m[min_k - 1])[tid];
   __syncthreads();
}

// Initialize Lanczos vectors
if(min_k == 0) {
   const T isq2 = (T)0.7071068;
   for(unsigned int b=0; b<n; b+=blockSize) {
      int i = b + tid;
      if(i == 0)        { p[i] = isq2; q[i] = isq2;  }
      else if(i == n-1) { p[i] = isq2; q[i] = -isq2; }
      else if(i < n)    { p[i] = (T)0.0; q[i] = (T)0.0;  }

      __syncthreads();
   }
}

// Assign work array (n_align-sized blocks)
// 0: w, va
T *w  = work;
T *va = work;
// 1: vs
T *vs = work + n_align;
// 2: v, workspace of orthogonalize (1 max_k block)
T *v  = work + n_align * 2;
T *work_ort = v;
// 3: workspace of levinson (2 n_align blocks)
T *work_lev = work + n_align * 3;

// Main loop
T *pk = p + min_k * n_align;
T *qk = q + min_k * n_align;
unsigned int max_it = min(max_k, min_k + num_it);
__syncthreads();

unsigned int k;
for(k=min_k+1; k<=max_it; ++k) {

   // w = pk + qk
   for(unsigned int b=0; b<n; b+=blockSize) {
      unsigned int i = b + tid;
      if(i < n) w[i] = pk[i] + qk[i];
   }
   __syncthreads();

   // Solve (toeplitz([td t]) - s * I) * v = w
   levinson(n, td - sigma, t, w, v, work_lev, swork);

   // Calculate vs and va vectors
   axpxb_reverse_x(n,  (T)1.0, v, vs, (T)0.5, swork);
   axpxb_reverse_x(n, -(T)1.0, v, va, (T)0.5, swork);

   // Dot product: ak = vs * pk, gk = va * qk
   T aux_alpha = dot(n, vs, pk, swork);
   T aux_gamma = dot(n, va, qk, swork);

   if(tid == 0) { mk.alpha = aux_alpha; mk.gamma = aux_gamma; }
```

```
      __syncthreads();

      // Avoid unnecessary calculations if reached maximum subspace size
      if(k < max_k) {

         // vs -= ak * pk, va -= gk * qk
         axpy(n, -mk.alpha, pk, vs);
         axpy(n, -mk.gamma, qk, va);

         // vs -= b(k-1) * p(k-1), va -= d(k-1) * q(k-1)
         if(k > 1) {
            // mk.beta and mk.delta from previous iteration (not updated yet)
            axpy(n, -mk.beta,  pk - n_align, vs);
            axpy(n, -mk.delta, qk - n_align, va);
         }

         // Reorthogonalization
         reorthogonalize(n, k, p, vs, work_ort, swork);
         reorthogonalize(n, k, q, va, work_ort, swork);

         // bk = norm(vs), dk = norm(va)
         T aux_beta  = norm(n, vs, swork);
         T aux_delta = norm(n, va, swork);
         if(tid == 0) { mk.beta = aux_beta; mk.delta = aux_delta; }
         __syncthreads();

         // p(k+1) = vs / bk, q(k+1) = va / dk
         pk += n_align; qk += n_align;
         T ibk = (T)1.0 / mk.beta;
         T idk = (T)1.0 / mk.delta;

         for(unsigned int b=0; b<n; b+=blockSize) {
            volatile unsigned int i = b + tid;
            if(i < n) {
               pk[i] = vs[i] * ibk;
               qk[i] = va[i] * idk;
            }
         }
         __syncthreads();
      }

      // Store tridiagonal results in global memory
      if(tid < 4) ((T *)&m[k-1])[tid] = ((T *)&mk)[tid];
      __syncthreads();
   }
}
```

Código fuente de la función auxiliar *memcpy4*

```
/**
 * Perform a coalesced raw copy from aligned addresses with data sizes multiple of 4.
 *
 * \param dest Source address. Assumed to be aligned to 16 * sizeof(float)
 *             if points to global memory.
 * \param src  Destination address. Assumed to be aligned to 16 * sizeof(float)
 *             if points to global memory.
 * \param size Data size in bytes. Should be always a multiple of 4.
*/
```

```
__device__ void memcpy4(void *dest, void *src, unsigned int size) {

   unsigned int tid = threadIdx.x;
   float *destf = (float *)dest;
   float *srcf  = (float *)src;

   __syncthreads();

   size >>= 2;
   for(unsigned int b=0; b<size; b+=blockSize, srcf+=blockSize, destf+=blockSize) {
      unsigned int i = b + tid;
      if(i < size) destf[tid] = srcf[tid];
   }

   __syncthreads();
}
```

Código fuente de *si2w_global*

```
/**
 * Entry point for parallel shift-and-invert 2-way Lanczos evaluation.
 *
 * \param n            Size of the symmetric Toeplitz matrix.
 * \param td           Main diagonal value of the symmetric Toeplitz matrix.
 * \param t            Remaning n-1 values from symmetric Toeplitz matrix.
 * \param k            Current value of subspace size iterator.
 *                     Real subspace size will be calculed using interval data.
 * \param inc_k        Number of iterations to perform in the Krylov subspace.
 * \param max_k        Maximum size of the Krylov subspace. Iterations will halt if reached.
 * \param exec_window  Pointer to execution window containing GPU-side descriptors
 *                     of the intervals to be calculated.
*/
__global__ void si2w_parallel(unsigned int n, T td, T *t, unsigned int k, unsigned int inc_k,
   unsigned int max_k, interval_gpu_info *exec_window) {

   // Copy interval descriptor into shared memory (coalesced)
   __shared__ interval_gpu_info i;
   memcpy4(&i, exec_window + blockIdx.x, sizeof(interval_gpu_info));

   // Skip already processed intervals
   if(i.processed) return;

   // Call shift-and-invert 2-way Lanczos method
   unsigned int k_real = k - i.start_k;
   if(k_real <= max_k) {
      si2w(n, td, t, k_real, inc_k, max_k, i.sigma, i.p, i.q, i.m, i.workspace);
   }
}
```

Referencias

[1] M. O. Bernabeu, P. Alonso, and A. M. Vidal, *A multilevel parallel algorithm to solve symmetric Toeplitz linear systems*, The Journal of Supercomputing, 44 (2008), pp. 237–256.

[2] T. Brandvik and G. Pullan, *Acceleration of a 3d euler solver using commodity graphics hardware*, in 46th AIAA Aerospace Sciences Meeting, University of Cambridge, Jan. 2008.

[3] T. F. Chan and P. C. Hansen, *A look-ahead Levinson algorithm for indefinite Toeplitz systems*, SIAM Journal on Matrix Analysis and Applications, 13 (1992), pp. 490–506.

[4] D. Göddeke, R. Strzodka, and S. Turek, *Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations*, Parallel Algorithms Appl, 22 (2007), pp. 221–256.

[5] G. H. Golub and C. F. van Loan, *Matrix computations*, Johns Hopkins University Press, 3rd, ed., 1996.

[6] L. Graciá, *Multi-GPU symmetric Toeplitz Eigenvalue Extractor user manual.*

[7] M. H. Gutknecht and M. Hochbruck, *Look-ahead Levinson and Schur algorithms for non-Hermitian Toeplitz systems*, Numerische Mathematik, 70 (1995), pp. 181–227.

[8] C. Lessig, *Eigenvalue computation with cuda*, cuda sdk 2.0 whitepaper, NVIDIA Corporation, Oct. 2007.

[9] M. P. McGraw-Herdeg, D. P. Enright, and B. S. Michael, *Benchmarking the nvidia 880gtx with the cuda development platform.*

[10] J. Molemaker, J. M. Cohen, S. Patel, and J.Ñoh, *Low viscosity flow simulations for animation*, (2008).

[11] NVIDIA Corporation, *NVIDIA CUDA Programming Guide Version 2.0.*

[12] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. van de Geijn, *Solving dense linear systems on platforms with multiple hardware accelerators.* FLAME Project Working Note #32.

[13] A. M. Vidal, V. M. Garcia, P. Alonso, and M. O. Bernabeu, *Parallel computation of the eigenvalues of symmetric Toeplitz matrices through iterative methods*, Journal of Parallel and Distributed Computing, 68 (2008), pp. 1113–1121.

[14] H. Voss, *A symmetry exploiting Lanczos method for symmetric Toeplitz matrices*, Numerical Algorithms, 25 (2000), pp. 377–385. Mathematical journey through analysis, matrix theory and scientific computation (Kent, OH, 1999).