Universidad Central de Venezuela

Facultad de Ciencias

Escuela de Computación

IMPLEMENTACIÓN DE MÉTODOS DE KRYLOV PARA RESOLVER SISTEMAS LINEALES EMPLEANDO CUDA

Br. Daysli E. Carmona S.

Zenaida Castillo, Tutora

Caracas, 02 de noviembre de 2012

Universidad Central de Venezuela

Facultad de Ciencias

Escuela de Computación

IMPLEMENTACIÓN DE MÉTODOS DE KRYLOV PARA RESOLVER SISTEMAS LINEALES EMPLEANDO CUDA

Br. Daysli E. Carmona S.

Zenaida Castillo, Tutora

Caracas, 02 de noviembre de 2012

IMPLEMENTACIÓN DE MÉTODOS DE KRYLOV PARA RESOLVER SISTEMAS LINEALES EMPLEANDO CUDA

Br. Daysli E. Carmona S.

Trabajo Especial de Grado presentado

ante la ilustre Universidad Central de Venezuela

como requisito parcial para optar al título de

Licenciado en Computación.

Con todo mi amor dedico este trabajo a Dios y a mis Padres

Agradecimientos

Gracias Dios por la gran misericordia que has tenido para conmigo, por la fortaleza que me has dado para afrontar todas las dificultades hasta el último momento en este largo camino y en mi vida.

Gracias Mami y Papi por todo su amor que día a día me han demostrado desde mi infancia hasta hoy, por el apoyo para continuar y de cerrar ciclos, por su esfuerzo para llevarme hasta donde estoy y por el aguante que tuvieron al esperar ver plasmado este esfuerzo, los amo con toda mi alma.

Gracias a mis dos hermanos Daysi y Darnys, les agradezco por lo buenos hermanos que han sido conmigo por siempre querer lo mejor para mi y que gracias a sus experiencias sembraron en mi espíritu de avance.

A mi precioso sobrinito Luis Daniel, que es la luz que alumbra y activa a mi y a mi familia, gracias cielito por ser tan lindo y amoroso con tu tía Days, te amo mucho mi principe.

A mi amado esposo Carlos Guía, mi amor muchas gracias eres la bendición mas hermosa que Dios haya podido colocar en mi camino durante los años de mi carrera, eres una gran inspiración y la persona que más admiro en en campo de la Computación en toda la Facultad de Ciencias y en Venezuela, gracias por estar a mi lado y por haberte convertido en mi guía en lo que respecta la carrera, por ser tan buen novio, amigo y ahora esposo, TE AMO!.

Gracias a la Profesora Carmen Elena Vera, primeramente por haber dado a luz al hombre que hoy llena de felicidad mi corazón, y seguidamente por toda la ayuda que me ha brindado desde el momento en que me conoció, por su dedicación e invaluables consejos que me dio para que finalizara este Trabajo de Grado, muchas gracias Sra. Elena, por hoy ser my mother in law, la quiero mucho.

Al Profesor Carlos D. Guía muchas gracias Sr. Carlos por todo su apoyo que me brindó y por todo el cariño que me ha demostrado y que por cierto es recíproco, lo quiero mucho.

Gracias al profesor Ferenc Szigeti, por estar ahi conmigo en todo, por ayudarme a ver mi verdadera vocación y mostrarme el rico y maravilloso arte que es el mundo del vino, gracias querido Ferenc, ahora si tenemos tiempo para compartir y catar muchos vinos. Gracias por el cariño especial que me tienes, por sentirme como la hija que jamás tuviste, espero jamás defraudarte, te quiero!.

Gracias a la profesora Zenaida Castillo por todo su apoyo en momentos de crisis, por su habilidad para hacer mover las cosas en cortos períodos de tiempo y por mostrar interés en que yo hoy pueda decir: me Gradue! y no decir, hice la carrera pero solo me faltó terminar la tesis, muchas gracia prof. por reafirmar esa gran diferencia.

Gracias al profesor Otilio Rojas, si prof, usted fue uno de los pocos profesores de computación quien me inspiro a terminar, a que el trabajo bien valía la pena, gracias a su confianza hoy terminé y puedo decir tranquila que el trabajo cumple con el método científico y sólo por eso hoy agradezco todas sus palabras alentadoras.

Gracias al profesor Rhadamés Carmona por su buena intención en que todo saliera bien, por su disponibilidad en los momentos que lo necesite y por ayudarme a dar este último paso, muchas gracias profe.

Gracias a mis grandes amigos, Jhonattan Piña y Tahiris Márquez, ustedes mejor que que nadie pudieron ver de cerca cada etapa de mi carrera. Gracias por todo su apoyo incondicional, por los ricos momentos compartidos y por cada sonrisa que me brindaron en todos y cada uno de los días que estuvimos juntos, los quieroooooo!.

Finalmente gracias a todos los amigo que nombro rápidamente en estas líneas y que saben que de cierta forma contribuyeron en hacer posible este logro. Digo sus nombres, Alejandro M, Roberto M, Christiam M, Jackson, Yuraima O y Carlos CE, y si no recuerdo a alguien más pido disculpas por ello, Muchas gracias a todos!.

Resumen

Implementación de métodos de Krylov para resolver Sistemas lineales empleando CUDA

Daysli E. Carmona S.

Zenaida Castillo, Tutora

Universidad Central de Venezuela

Una gran cantidad de problemas que aparecen en las ciencias y en la ingeniería conducen a la resolución de grandes sistemas de ecuaciones lineales dispersos. En la actualidad, dichos problemas pueden ser resueltos eficientemente utilizando métodos iterativos de proyección sobre subespacios de Krylov, en gran medida, por su capacidad de ser implementados en ambientes paralelos. Sin embargo, el éxito de estos métodos reposa cada vez más en la escogencia de un buen precondicionador.

Con la incorporación de procesadores paralelos de cómputo intensivo en las tarjetas gráficas modernas, junto con la arquitectura y modelo de programación CUDA, es posible realizar cómputo numérico a gran escala evitando elevados costos de adquisición. En éste trabajo se propone implementar el precondicionador SPAI y los métodos GMRES y TFQMR en la GPU utilizando CUDA, para resolver grandes sistemas lineales dispersos.

Índice General

Introducción

Una gran cantidad de problemas que aparecen en las ciencias y en la ingeniería conducen a la resolución de sistemas de ecuaciones lineales de la forma:

$$Ax = b, \tag{0.1}$$

donde $A \in \mathbb{R}^{n\times n}$ es la matriz que representa a los coeficientes del sistema lineal, $x \in \mathbb{R}^n$ es el vector que representa a las variables o incógnitas y $b \in \mathbb{R}^n$ es el vector que representa a los términos independientes. En particular, el interés en este trabajo es resolver (0.1) cuando la matriz A es no singular, es decir, existe solución única, $x = A^{-1}b$ [24].

Para resolver el sistema mostrado en (0.1) existen dos clases de métodos, los llamados métodos *directos* y los métodos *iterativos*. Un método directo es aquel que permite obtener una solución en un número finito (conocido) de pasos. Usualmente se asocian a factorizaciones matriciales y tienen un costo computacional (operaciones punto flotante) de orden n^3 con respecto al tamaño de la matriz [9]. Por otro lado, los métodos iterativos parten de una solución inicial x_0 y generan una sucesión de vectores $\{x_k\}_{k=0}^{\infty}$ (soluciones aproximadas), que idealmente converge a la solución x^* del sistema, con un costo computacional de orden lineal con respecto al número de entradas no cero de la matriz A por iteración [21].

Cuando los sistemas son suficientemente pequeños y densos utilizar métodos directos suele ser la opción más adecuada. Sin embargo, estos métodos se vuelven prohibitivos, en términos de tiempo de cómputo y requerimientos de memoria, a medida que el tamaño de la matriz A se hace grande. Adicionalmente, en la mayoría de los problemas de interés la matriz de los coeficiente es dispersa, tómese por ejemplo los

sistemas lineales provenientes de la discretización de ecuaciones en derivadas parciales. Es por ello que desde mediados del siglo pasado, se ha buscado resolver estos problemas a través de métodos iterativos, ya que estos no alteran la matriz A y sólo requieren mantener unos pocos vectores de tamaño n en un momento dado, más aún, el proceso iterativo se puede suspender cuando se ha obtenido una precisión deseada o cuando se ha realizado un cierto número de iteraciones [13]. Los métodos iterativos más importantes para resolver el sistema (0.1) en la actualidad están basados en procesos de proyección sobre subespacios de Krylov, en gran medida, por su capacidad de ser implementados en ambiente paralelos[24].

Las propiedades espectrales de la matriz A pueden afectar la robustez de los métodos iterativos, lo cual, a pesar de su atractivo intrínseco, disminuye la aceptación de dichos métodos en aplicaciones industriales. Es por ello que éste tipo de métodos suele acompañarse de técnicas de *precondicionamiento*, lo que significa transformar el sistema original en otro que posea la misma solución. De ésta forma se tiende a mejorar la robustez y velocidad de convergencia de los métodos iterativos.

En la actualidad existen diferentes técnicas de precondicionamiento, y en los últimos años se han desarrollado vigorosamente las basadas en inversas aproximadas dispersas. Esto se debe a que en ocasiones es conveniente aproximar la matriz A^{-1} directamente en lugar de aproximar la matriz A, como es en el caso de precondicionadores basados en factorizaciones incompletas [6]. En general, la fiabilidad de las técnicas iterativas depende mucho más de la calidad del precondicionador que del método de subespacio de Krylov utilizado [24].

Actualmente, los equipos de alto rendimiento, como los clústeres de computadoras, tienen un elevado costo de adquisición. Sin embargo, hoy en día es posible adquirir, a un bajo costo, tarjetas gráficas para computadoras personales. Dichas tarjetas incorporan un potente chip, con procesadores especializados, para cálculos altamente paralelos de cómputo intensivo. Estos procesadores se encuentran en la Unidad de Procesamiento Gráfico (GPU) y pueden ser administrados eficientemente en conjunto con la Unidad Central de Procesamiento (CPU) para realizar cálculo masivo a bajo costo [10].

Cuando se desea realizar computo numérico a gran escala en una computadora

personal, la tendencia actual es sustituir el "procesamiento centralizado" en la CPU por el "coprocesamiento repartido" entre la CPU y la GPU (lo que introduce el término de GPGPU[1]). Esto conlleva a la aparición de nuevas tecnologías o arquitecturas vectoriales y paralelas como es el caso de la arquitectura CUDA[2], introducida por la compañía NVIDIA a finales del año 2006, con el fin de explotar la capacidad de cómputo de la GPU [11].

Dada la necesidad de resolver grandes sistemas lineales en el menor tiempo de cómputo posible, así como explotar las características de la GPU utilizando la arquitectura CUDA, un grupo de desarrolladores trabajó en la implementación de dos bibliotecas de algoritmos paralelos (publicadas en el año 2010). Dichas biblioteas poseen interfaces flexibles de alto nivel para mejorar la productividad del programador, además, permiten la ejecución de métodos tanto en la CPU como en la GPU. La primera de ellas, llamada Thrust [20], que implementa eficientemente estructuras de datos y algoritmos básicos en paralelo. La segunda, denominada Cusp [4], una biblioteca de álgebra lineal dispersa en CUDA, la cual ofrece una serie de operaciones entre las que se encuentran eficientes implementaciones de producto matriz-vector (SpMV). Cusp también provee dos métodos iterativos basados en subespacio de Krylov: el método del Gradiente Conjugado (CG) y el método de BICGSTAB, así como también los precondicionadores diagonal y AINV.

En este trabajo, se propone implementar en la GPU usando CUDA, los métodos iterativos del subespacio de Krylov GMRES [25] y TFQMR [18] y el precondicionador SPAI[3] para resolver sistemas lineales generales (no simétricos). Se usan las librerías Cusp y Thrust para implementar los métodos planteados y las operaciones que sean necesarias, de manera que la interacción entre los componentes ya existentes en las librerías con los que se desarrollarán sea transparente. Se harán comparaciones de tiempos de ejecución en ambos entornos de programación para los métodos ofrecidos por la librería Cusp y los métodos planteados.

[1] Cálculo de Propósito General en Unidades de procesamiento Gráfico (General-Purpose Computing on Graphics Processing Units)

[2] Compute Unified Device Architecture

[3] Inversa Aproximada Dispersa

Organización

El documento se organiza de la siguiente manera: en el capítulo 1, se presenta una introducción a la teoría de álgebra lineal necesaria, lo cual incluye los métodos de proyección, métodos de Krylov, precondicionamiento de métodos y la construcción del precondicionador SPAI. La programación de la GPU utilizando la arquitectura CUDA es estudiada en el capítulo 2, en el se muestran los aspectos relevantes de la arquitectura y una breve introducción a su programación. Luego, en el capítulo 3 se describe el diseño y se detalla la implementación en CUDA de los métodos de Krylov propuestos. Seguidamente, en el capítulo 4 se presentan los experimentos y resultados numéricos, así como comparaciones en términos de número de iteraciones y tiempo de ejecución entre las diferentes arquitecturas. Finalmente, las conclusiones se encuentran en el capítulo 5 y en el capítulo 6 se proponen los trabajos.

Notación

La notación usada en el resto de este trabajo es la siguiente: Letras mayúsculas denotan matrices, letras latinas minúsculas representan vectores y letras griegas minúsculas representan escalares. La letra I se reserva para la matriz identidad y el $j-$ésimo vector canónico se denota e_j. La notación A^{-1} se utiliza para la inversa de A, A^T para la transpuesta y A^* para la matriz conjugada transpuesta.

Objetivos

A continuación se presentan los objetivos de este trabajo Especial de Grado.

Objetivo General

Evaluar la capacidad de paralelismo masivo de las GPUs modernas para resolver grandes sistemas de ecuaciones lineales, de manera eficiente y robusta, utilizando la arquitectura CUDA como plataforma de desarrollo.

Objetivos Específicos

- Estudiar y analizar las librerías numéricas de libre distribución Cusp y Thrust para el desarrollo eficiente de métodos iterativos basados en subespacios de Krylov y precondicionadores de matrices.

- Implementar de forma paralela y secuencial, haciendo uso de las librerías Cusp y Thrust, los métodos iterativos basados en subespacios de Krylov, GMRES y TFQMR.

- Implementar el algoritmo de construcción del precondicionador SPAI, tanto en la GPU como en la CPU, haciendo uso de la arquitectura CUDA. Además, el precondicionador resultante debe poder ser utilizado tanto en los métodos propuestos como en los existentes en la biblioteca Cusp.

- Seleccionar ejemplos de matrices dispersas que surgen de diferentes aplicaciones reales, utilizando las colecciones de matrices de la universidad de Florida [14] y Matrix Market [8].

- Evaluar el comportamiento de los métodos iterativos basados en subespacios de Krylov propuestos y aquellos existentes en la biblioteca Cusp, tanto en paralelo como secuencial, para determinar la precisión y tiempo de cómputo necesario en ambos casos.

- Analizar el comportamiento de la construcción paralela del precondicionador SPAI utilizando la arquitectura CUDA, comparando la precisión y velocidad de construcción con respecto a su correspondiente implementación secuencial.

Capítulo 1

Métodos de proyección para sistemas lineales

En este capítulo, se discuten los métodos de proyección en general, seguidamente se describen los métodos iterativos de proyección sobre subespacios de Krylov, y finalmente se discuten las técnicas de precondicionamiento que son usadas para mejorar la convergencia y la velocidad de estos métodos de proyección.

Las técnicas iterativas más importantes para resolver grandes sistemas de ecuaciones lineales, están basadas en procesos de proyección. Un proceso de proyección consiste en la obtención de una aproximación a la solución buscada en un subespacio de menor dimensión; es decir, consiste en hallar una solución aproximada al sistema (0.1) a partir de un subespacio de $\mathbb{R}^n$. Si $\mathcal{K}_m$ es el subespacio de búsqueda de dimensión m, entonces, en general, se deben imponer m restricciones para extraer dicha aproximación. Una forma común de describir dichas restricciones es imponer m condiciones de ortogonalidad, específicamente, se restringe al vector residual $b - Ax$ a que sea ortogonal con respecto a m vectores linealmente independientes. De esta forma, se define un subespacio $\mathcal{L}_m$ también de dimensión m, llamado subespacio de restricciones [24].

Existen dos clases amplias de métodos de proyección, los métodos *ortogonales*, en los que el subespacio $\mathcal{L}_m$ es el mismo subespacio $\mathcal{K}_m$; y los métodos *oblicuos*, en los que los subespacios $\mathcal{L}_m$ y $\mathcal{K}_m$ son diferentes. Un resultado particular de proyección oblicua se da cuando la matriz es cuadrada y $\mathcal{L}_m = A\mathcal{K}_m$, ya que el vector $\tilde{x}$ es el resultado del proceso de proyección sobre $\mathcal{K}_m$ y ortogonal a $\mathcal{L}_m$ con el vector inicial

x_0, si y solo si, $\tilde{x}$ minimiza la norma-2 del vector residual $(b - Ax)$ con $x \in (x_0 + \mathcal{K}_m)$, para mayores detalles ver [24].

1.1. Métodos de Subespacio de Krylov

En esta sección se describe un conjunto de métodos de proyección que utilizan como el subespacio de búsqueda $\mathcal{K}_m$ el subespacio de Krylov para resolver el sistema (0.1), y que han sido utilizados con gran éxito desde la década de los 50.

Un *método de subespacio de Krylov* es un método iterativo de proyección donde el subespacio de búsqueda $\mathcal{K}_m$ es el subespacio de Krylov y son de la forma:

$$\mathcal{K}_m(A) = span\{r_0, Ar_0, A^2r_0, \ldots, A^{m-1}r_0\}, \tag{1.1}$$

donde $r_0 = b - Ax_0$, y x_0 es el iterado inicial del proceso.

Los distintos métodos de Krylov que existen en la actualidad, se derivan de las diferentes formas de escoger el subespacio de restricciones $\mathcal{L}_m$. Dos importantes familias que existen entre las variantes de los métodos basados en subespacios de Krylov serán descritas en el transcurso de este capitulo. La primera de ellas es la basada en el método de Arnoldi, esta elige el subespacio de restricciones $\mathcal{L}_m = \mathcal{K}_m$ o la variación del residual mínimo $\mathcal{L}_m = A\mathcal{K}_m$ y seguidamente se hablará de la segunda que se inspira en los métodos de Lanczos, basada en definir $\mathcal{L}_m$ como el subespacio de Krylov asociado con A^T, es decir, $\mathcal{L}_m = \mathcal{K}_m(A^T, r_0)$ o $\mathcal{L}_m = A\mathcal{K}_m(A^T, r_0)$ [24].

1.1.1. Método de Arnoldi

El método de Arnoldi [1] es un método de proyección ortogonal sobre subespacios de $\mathcal{K}_m$ para matrices generales no Hermitianas. Este procedimiento se introdujo en el año 1951 inicialmente como un método directo para reducir matrices densas a forma de Hessenberg. Arnoldi presentó su método de esta manera, pero indicó que los autovalores de la matriz de Hessenberg obtenidos en un número de pasos menor a n pudiera

proporcionar autovalores de A. Posteriormente se descubrió que esta estrategia conduce a una técnica eficaz para aproximar autovalores de matrices grandes y dispersas. En aritmética exacta, una variante del algoritmo puede verse en el Algoritmo 1.1

En cada paso del algoritmo se multiplica al vector de Arnoldi anterior v_j por A y luego se ortonormaliza el vector resultante w_j con respecto a todos los vectores v_i anteriores utilizando un procedimiento estándar de Gram-Schmidt. Si en algún momento w_j se hace 0, el algoritmo se detiene.

Se asume que el Algoritmo 1.1 no se detiene hasta el paso m-ésimo. Entonces los vectores $v_1, \ldots, v_m$ forman una base ortonormal del espacio de Krylov, para un demostración ver [24].

$$\mathcal{K}_m = span\{v_1, Av_1, \ldots, A^{m-1}v_1\}.$$

1. Escoger un vector v_1 de norma 1
2. Para $j = 1, 2, \ldots, m$ Hacer:
3. | Calcular $h_{ij} = \langle Av_j, v_i \rangle$ para $i = 1, 2, \ldots, j$
4. | Calcular $w_j = Av_j - \sum_{i=1}^{j} h_{ij} v_i$
5. | $h_{j+1,j} = \|w_j\|_2$
6. | Si $h_{j+1,j} = 0$ entonces: Parar
7. | $v_{j+1} = \frac{1}{h_{j+1,j}} w_j$
8. FinPara

Algoritmo 1.1: Arnoldi básico

Si se denota como V_m la matriz de orden $n \times m$ cuyas columnas son los vectores $v_1, \ldots, v_m$, como $\bar{H}_m$ como la matriz Hessenberg de $(m+1) \times m$ cuyas entradas distintas de cero h_{ij} son definidas por el Algoritmo 1.1, y como H_m la matriz que se obtiene eliminando la última fila de $\bar{H}_m$. Entonces, las siguientes relaciones se cumplen:

$$AV_m = V_m H_m + w_m e_m^T \tag{1.2}$$

$$= V_{m+1}\bar{H}_m \tag{1.3}$$

$$V_m^T A V_m = H_m \tag{1.4}$$

Una demostración para las relaciones anteriores se encuentra en [24].

Como se mencionó anteriormente, el algoritmo de Arnoldi asume aritmética exacta. Sin embargo, se puede ganar mucha estabilidad utilizando el algoritmo de Gram-Schmidt modificado o la ortogonalización de Householder en lugar del algoritmo estándar de Gram-Schmidt. Una buena ilustración y detalles de estos algoritmos pueden verse en
citesaad.

1.1.2. Método de Residual Mínimo Generalizado (GMRES)

El método del Residual Mínimo Generalizado[1] GMRES, fue propuesto por Saad y Schultz en 1986 [25], es un método de proyección oblicua que se basa en considerar $\mathcal{K}_m = \mathcal{K}_m(A, r_0)$ y $\mathcal{L}_m = A\mathcal{K}_m$ donde $\mathcal{K}_m(A, r_0)$ es el subespacio de Krylov de dimensión m y v_1 se define como $v_1 = \frac{r_0}{\beta}$, donde $\beta = \|r_0\|_2$.

Como se ha visto, en el método de Arnoldi se reduce una matriz no simétrica a Hessenberg superior. GMRES aprovecha esta transformación que hace el método de Arnoldi para la construcción de soluciones aproximadas en las cuales, la norma del residual será mínima con respecto a $x_0 + \mathcal{K}_m$ que es la restricción basada en la aproximación del residual mínimo.

Dado que cualquier vector x en $x_0 + \mathcal{K}_m$ se puede escribir como

$$x = x_0 + V_m y \tag{1.5}$$

[1]Método del Residual Mínimo Generalizado (Generalized Minimum Residual Method)

donde y es un vector m-dimensional, entonces, se define

$$J(y) = \|b - Ax\|_2 = \|b - A(x_0 + V_m y)\|_2 \tag{1.6}$$

la relación de la ecuación (1.3) se convierte en

$$\begin{aligned} b - Ax &= b - A(x_0 + V_m y) \\ &= r_0 - AV_m y \\ &= \beta v_1 - V_{m+1}\bar{H}_m y \\ &= V_{m+1}(\beta e_1 - \bar{H}_m y) \end{aligned} \tag{1.7}$$

Como los vectores columna de V_{m+1} son ortonormales, entonces

$$J(y) = \|b - A(x_0 + V_m y)\|_2 = \left\|\beta e_1 - \bar{H}_m y\right\|_2 \tag{1.8}$$

La aproximación de GMRES es el vector único de $x_0 + \mathcal{K}_m$ que minimiza a la ecuación (1.8) mostrada anteriormente. Utilizando las ecuaciones (1.5) y (1.8), esta aproximación es $x_m = x_0 + V_m y_m$ donde y_m minimiza la función $J(y)$, es decir

$$x_m = x_0 + V_m y_m \tag{1.9}$$

$$y_m = \text{argmin}_y \left\|\beta e_1 - \bar{H}_m y\right\|_2 \tag{1.10}$$

La minimización de y no es costosa de calcular ya que requiere la solución de un problema de mínimos cuadrados de $(m+1) \times m$, donde m es tipicamente pequeño. El Algoritmo 1.2 muestra una implementación de GMRES utilizando Gram-Schmidt modificado en el proceso de Arnoldi.

Para resolver el problema de mínimos cuadrados del paso 19 del Algoritmo 1.2, es común transformar la matriz de Hessenberg en una triangular superior utilizando una secuencia de rotaciones de Givens [24], donde cada matriz de rotación es de la forma

1. Sea $\bar{H}_m = \{h_{ij}\}_{1 \leq i \leq m+1, 1 \leq j \leq m}$, una matriz de $(m+1) \times m$
2. $r_0 := b - Ax_0$
3. $\beta := \|r_0\|_2$
4. $v_1 := \frac{r_0}{\beta}$
5. $\bar{H}_m := 0$
6. Para $j = 1, 2, \ldots, m$ Hacer:
7. | $w_j := Av_j$
8. | Para $i = 1, 2, \ldots, j$ Hacer:
9. | | $h_{ij} := \langle w_j, v_i \rangle$
10. | | $w_j := w_j - h_{ij} v_i$
11. | FinPara
12. | $h_{j+1,j} := \|w_j\|_2$
13. | Si $h_{j+1,j} = 0$ entonces:
14. | | $m := j$
15. | | Ir al paso 19
16. | FinSi
17. | $v_{j+1} = \frac{1}{h_{j+1,j}} w_j$
18. FinPara
19. $y_m := \text{mín}_y \left\| \beta e_1 - \bar{H}_m y \right\|_2$
20. $x_m := x_0 + V_m y_m$

Algoritmo 1.2: Residual Mínimo Generalizado (GMRES) con Gram-Schmidt Modificado

1. $r_0 := b - Ax_0$
2. $\beta := \|r_0\|_2$
3. $v_1 := \frac{r_0}{\beta}$
4. Generar las matrices V_m y $\bar{H}_m$ utilizando el algoritmo de Arnoldi con v_1 como vector inicial.
5. $y_m := \text{mín}_y \left\| \beta e_1 - \bar{H}_m y \right\|_2$
6. $x_m := x_0 + V_m y_m$
7. Terminar Si x_m satisface el criterio de parada
8. $x_0 := x_m$
9. Ir a 1

Algoritmo 1.3: Residual Mínimo Generalizado (GMRES) con reinicio

$$\Omega_i = \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & s_i & & & \\ & & & -s_i & c_i & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{pmatrix} \begin{matrix} \\ \\ \\ \leftarrow \text{fila } i \\ \leftarrow \text{fila } i+1 \\ \\ \\ \\ \end{matrix} \tag{1.11}$$

y los coeficientes c_i y s_i cumplen con $c_i^2 + s_i^2 = 1$ y son escogidos para eliminar $h_{i+1,i}$ en cada paso utilizando la ecuación (1.12)

$$s_i = \frac{h_{i+1,i}}{\sqrt{(h_{i,i}^{(i-1)})^2 + h_{i+1,i}^2}}, c_i = \frac{h_{i,i}^{(i-1)}}{\sqrt{(h_{i,i}^{(i-1)})^2 + h_{i+1,i}^2}}. \tag{1.12}$$

Si se denota como Q_m al producto de las matrices de rotación

$$Q_m = \Omega_m \Omega_{m-1} \dots \Omega_2 \Omega_1$$

y

$$\begin{aligned} \bar{R}_m &= Q_m \bar{H}, \\ \bar{g}_m &= Q_m(\beta e_1). \end{aligned}$$

Entonces, dado que Q_m es unitaria, se tiene que

$$\min_y \left\| \beta e_1 - \bar{H}_m y \right\|_2 = \min_y \left\| \bar{g}_m - \bar{R}_m y \right\|_2. \tag{1.13}$$

La solución de (1.13) se obtiene resolviendo el sistema triangular que resulta de eliminar la última fila de $\bar{R}_m$ y $\bar{g}_m$. Además, la norma del residual no es más que el valor absoluto del último elemento de $\bar{g}_m$. Más aún, es posible aplicar las rotaciones progresivamente en cada paso del algoritmo, de esta forma se puede obtener la norma del residual en cada paso virtualmente sin costo adicional [24].

Variante: GMRES con reinicio

GMRES se vuelven poco práctico cuando m es muy grande, debido al crecimiento de los requerimientos de memoria y de cómputo del algoritmo a medida que el m aumenta puede llegar a ser al menos de $O(m^2 n)$. *GMRES* con reinicio se aplica para dar solución a esta situación [27]. Esta técnica consiste en reinicializar el algoritmo periódicamente, es decir, calcular una solución aproximada para un m pequeño. Si el error de dicha solución es suficientemente pequeño, entonces el proceso se detiene, en caso contrario, se toma ésta solución como el nuevo iterado inicial y se comienza el proceso nuevamente. Implementaciones de ésta estrategia se pueden ver en el Algoritmo 1.3.

1.1.3. Residual Cuasi-Mínimo Libre de Transpuesta (TFQMR)

El algoritmo del Residual Cuasi-Mínimo Libre de Transpuesta[2] TFQMR, fue propuesto por Roland W. Freund [18] en el año 1993 como un derivado del algoritmo CGS [24]. Es un método que resuelve sistemas de ecuaciones lineales no simétricos, calculando una aproximación en el subespacio $\mathcal{K}_m$.

TFQMR puede implementarse de manera sencilla cambiando unas pocas líneas del algoritmo estándar CGS, para ello es necesario observar que la actualización de x_j en CGS [24] puede realizarse en dos etapas, es decir,

$$\begin{aligned} x_{j+\frac{1}{2}} &= x_j + \alpha_j u_j \\ x_{j+1} &= x_{j+\frac{1}{2}} + \alpha_j q_j \end{aligned}$$

Esta división es natural ya que dicha actualización involucra dos productos matriz vector para pasar de un iterado al siguiente. Un excelente estudio de este algoritmo es mostrado en [24]. El método iterativo propuesto por Freund es mostrado en el Algoritmo 1.4

1.2. Métodos Precondicionados

Un precondicionador es simplemente un medio para transformar el sistema original en otro sistema equivalente con propiedades espectrales más favorables, por lo que probablemente será más fácil de resolver con un método iterativo. En la práctica, la fidelidad de las técnicas iterativas depende mucho más de la calidad del precondicionador que del iterador del subespacio de Krylov utilizado [24].

Frecuentemente, conseguir un buen precondicionador para un sistema lineal es visto como una combinación de arte y ciencia [17], ya que se debe hallar una matriz M tal que [15]:

1. M sea una buena aproximación de A en algún sentido

[2]Residual Cuasi-Mínimo Libre de Transpuesta(Transpose Free QMR)

1. $w_0 := u_0 := r_0 := b - Ax_0$
2. $v_0 := Au_0$
3. $d_0 := 0$
4. $\tau_0 := \|r_0\|_2$
5. $\theta_0 := \eta_0 = 0$
6. Escoger un vector r_0^* arbitrario, tal que $\rho_0 \equiv \left\langle r_0^*, r_0 \right\rangle \neq 0$
7. Para $m = 0, 1, 2 \ldots$, hasta alcanzar la convergencia Hacer:
8. | Si m es par entonces:
9. | | $\alpha_{m+1} = \alpha_m = \frac{\rho_m}{\left\langle v_m, r_0^* \right\rangle}$
10. | | $u_{m+1} = u_m - \alpha_m v_m$
11. | FinSi
12. $w_{m+1} = w_m - \alpha_m A u_m$
13. $d_{m+1} = u_m + \left(\frac{\theta_m^2}{\alpha_m}\right) \eta_m d_m$
14. $\theta_{m+1} = \frac{\|w_{m+1}\|_2}{\tau_m}; c_{m+1} = (1 + \theta_{m+1}^2)^{-\frac{1}{2}}$
15. $\tau_{m+1} = \tau_m \theta_{m+1} c_{m+1}; \eta_{m+1} = c_{m+1}^2 \alpha_m$
16. $x_{m+1} = x_m + \eta_{m+1} d_{m+1}$
17. $r_{m+1} = r_m - \eta_{m+1} A d_{m+1}$
18. | Si m es impar entonces:
19. | | $\rho_{m+1} = \left\langle w_{m+1}, r_0^* \right\rangle$
20. | | $\beta_{m-1} = \frac{\rho_{m+1}}{\rho_{m-1}}$
21. | | $u_{m+1} = w_{m+1} + \beta_{m-1} u_m$
22. | | $v_{m+1} = Au_{m+1} + \beta_{m-1}(Au_m + \beta_{m-1} v_{m-1})$
23. | FinSi
24. FinPara

Algoritmo 1.4: Residual Quasi-Mínimo Libre de Transpuesta (TFQMR)

2. El costo de construir M no sea prohibitivo

3. Resolver sistemas de la forma $Mx = b$ sea económico

Una vez que se tenga la matriz precondicionadora M, esta puede ser aplicada por la izquierda, lo que lleva al sistema precondicionado $M^{-1}Ax = M^{-1}b$, también puede ser aplicado por la derecha y para ello, se realiza un cambio de variables $u = Mx$ y se resuelve el sistema $AM^{-1}u = b$, con, $x = M^{-1}u$ con respecto al vector de incógnitas u. Y finalmente, es común disponer de un precondicionador factorizado de la forma $M = M_L M_R$ donde, típicamente M_L y M_R son matrices triangulares. E n esta situación, el precondicionador puede ser aplicado por ambos lados de la siguiente manera $M_L^{-1} A M_R^{-1} u = M_L^{-1} b$, con, $x = M_R^{-1} u$.

En este trabajo únicamente se hará énfasis en la forma de precondicionamiento por la derecha, para un análisis en detalle de como es el funcionamiento de un precondicionador por la izquierda o por ambos lados puede verse en [24].

Seguidamente se muestran las formas de aplicar un precondicionador por la derecha para los método GMRES y TFQMR mencionando sus propiedades.

1.2.1. GMRES Precondicionado por la Derecha

El algoritmo de GMRES precondicionado por la derecha se basa en resolver

$$AM^{-1}u = b,\ u = Mx.$$

A continuación se muestra que la nueva variable u no necesita ser invocada explícitamente. En efecto, todos los vectores del subespacio de Krylov pueden ser obtenidos sin ninguna referencia para las variables u. Únicamente las instrucciones que utilizan el vector de incógnitas han de ser analizadas, es decir, el calculo del residual inicial y la actualización de la solución. Comenzando con el calculo del residual inicial se tiene que

$$\begin{aligned} r_0 &= b - AM^{-1}u_0 \\ &= b - AM^{-1}Mx_0 \\ &= b - Ax_0 \end{aligned} \tag{1.14}$$

Como se puede ver en (1.14) el residual inicial no cambia para el sistema precondicionado. Para la actualización de la solución del sistema precondicionado se utiliza

$$\begin{aligned} u_m &= u_0 + V_m y_m \\ Mx_m &= Mx_0 + V_m y_m \\ M^{-1}Mx_m &= M^{-1}(Mx_0 + V_m y_m) \\ x_m &= x_0 + M^{-1}V_m y_m \end{aligned} \tag{1.15}$$

Utilizando las ecuaciones (1.14) y (1.15) en el algoritmo de GMRES se obtiene el método de GMRES precondicionado por la derecha como se muestra a continuación

1. Sea $\bar{H}_m = \{h_{ij}\}_{1 \leq i \leq m+1, 1 \leq j \leq m}$, una matriz de $(m+1) \times m$
2. $r_0 := b - Ax_0$
3. $\beta := \|r_0\|_2$
4. $v_1 := \frac{r_0}{\beta}$
5. $\bar{H}_m := 0$
6. Para $j = 1, 2, \ldots, m$ Hacer:
7. | $w_j := AM^{-1}v_j$
8. | Para $i = 1, 2, \ldots, j$ Hacer:
9. | | $h_{ij} := \langle w_j, v_i \rangle$
10. | | $w_j := w_j - h_{ij}v_i$
11. | FinPara
12. | $h_{j+1,j} := \|w_j\|_2$
13. | Si $h_{j+1,j} = 0$ entonces:
14. | | $m := j$
15. | | Ir al paso 19
16. | FinSi
17. | $v_{j+1} = \frac{1}{h_{j+1,j}} w_j$
18. FinPara
19. $y_m := \text{mín}_y \left\| \beta e_1 - \bar{H}_m y \right\|_2$
20. $x_m := x_0 + M^{-1}V_m y_m$

Algoritmo 1.5: Residual Mínimo Generalizado (GMRES) con Gram-Schmidt Modificado, Precondicionado por la Derecha

En este caso, el método de Arnoldi construye una base ortogonal del subespacio de Krylov precondicionado por la derecha

$$\mathcal{K}_m(AM^{-1}) = span\{r_0, AM^{-1}r_0, A^2M^{-1}r_0, \ldots, A^{m-1}M^{-1}r_0\}$$

Además, la norma del residual es ahora relativa al sistema inicial, es decir, $r_m = b - Ax_m$. Esto se debe a que el algoritmo obtiene dicho residual de manera implícita a partir de $b - Ax_m = b - AM^{-1}u_m$. Esta es una diferencia esencial con el enfoque del algoritmo GMRES precondicionado por la izquierda, para detalles de este enfoque ver [24] .

1.2.2. TFQMR Precondicionado por la Derecha

El algoritmo TFQMR precondicionado por la derecha, al igual que GMRES precondicionado por la derecha, no necesita cambios en el calculo del residual inicial.

Además, en los pasos donde esta la matriz A se reemplaza por AM^{-1} y en la actualización del vector solución se multiplica por M^{-1} el paso. Tomando todas estas consideraciones en cuenta, el algoritmo resultante es el siguiente:

1. $w_0 := u_0 := r_0 := b - Ax_0$
2. $v_0 := AM^{-1}u_0$
3. $d_0 := 0$
4. $\tau_0 := \|r_0\|_2$
5. $\theta_0 := \eta_0 = 0$
6. Escoger un vector r_0^* arbitrario, tal que $\rho_0 \equiv \left\langle r_0^*, r_0 \right\rangle \neq 0$
7. Para $m = 0, 1, 2 \ldots$, hasta alcanzar la convergencia Hacer:
8. | Si m es par entonces:
9. | | $\alpha_{m+1} = \alpha_m = \frac{\rho_m}{\langle v_m, r_0^* \rangle}$
10. | | $u_{m+1} = u_m - \alpha_m v_m$
11. | FinSi
12. $w_{m+1} = w_m - \alpha_m AM^{-1}u_m$
13. $d_{m+1} = u_m + \left(\frac{\theta_m^2}{\alpha_m}\right)\eta_m d_m$
14. $\theta_{m+1} = \frac{\|w_{m+1}\|_2}{\tau_m}; c_{m+1} = (1+\theta_{m+1}^2)^{-\frac{1}{2}}$
15. $\tau_{m+1} = \tau_m \theta_{m+1} c_{m+1}; \eta_{m+1} = c_{m+1}^2 \alpha_m$
16. $x_{m+1} = x_m + \eta_{m+1} M^{-1} d_{m+1}$
17. $r_{m+1} = r_m - \eta_{m+1} AM^{-1} d_{m+1}$
18. | Si m es impar entonces:
19. | | $\rho_{m+1} = \left\langle w_{m+1}, r_0^* \right\rangle; \beta_{m-1} = \frac{\rho_{m+1}}{\rho_{m-1}}$
20. | | $u_{m+1} = w_{m+1} + \beta_{m-1} u_m$
21. | | $v_{m+1} = AM^{-1}u_{m+1} + \beta_{m-1}(AM^{-1}u_m + \beta_{m-1}v_{m-1})$
22. | FinSi
23. FinPara

Algoritmo 1.6: Residual Quasi-Mínimo Libre de Transpuesta (TFQMR) Precondicionado por la Derecha

1.3. Inversas Aproximadas Dispersas

En los últimos años se ha incrementado el interés en precondicionadores que aproximan la inversa de la matriz de coeficientes A directamente, es decir, se busca una matriz dispersa $M \approx A^{-1}$. La principal ventaja de éste enfoque es que la operación de precondicionamiento puede implementarse fácilmente en paralelo, ya que consiste únicamente en productos matriz-vector. Más aún, la construcción y aplicación de

precondicionadores de éste tipo tienden a ser inmunes a ciertas dificultades numéricas como pivotes nulos e inestabilidad [6].

La técnicas de inversas aproximadas dispersas confían en la asunción que dada una matriz dispersa A, es posible conseguir una matriz dispersa M que es una buena aproximación, en algún sentido, de A^{-1}. Sin embargo, la inversa de una matriz dispersa no es necesariamente dispersa. Más precisamente, para cualquier patrón de dispersión dado es posible asignar valores numéricos a los elementos distintos de cero de tal manera que todas las entradas de la inversa sean diferentes de cero [16]. Aún así, es frecuente que muchas de las entradas de la inversa de una matriz dispersa sean pequeñas en valor absoluto, haciendo posible la aproximación de la misma por medio de una matriz dispersa [6].

Existen varios algoritmos completamente diferentes para el cálculo de una inversa aproximada dispersa, teniendo cada método sus propias fortalezas y limitaciones [7]. Por otro lado, se acostumbra a distinguir entre dos tipos básicos de inversas aproximadas, dependiendo de si el precondicionador $M \approx A^{-1}$ se expresa como una sola matriz o como producto de dos o más matrices. Estos últimos tipos de precondicionadores son conocidos como inversas aproximadas dispersas factorizadas y son de la forma

$$M = M_U M_L, \quad \text{donde} \quad M_U \approx U_1, \quad y \quad M_L \approx L_1 \tag{1.16}$$

L y U son los factores triangulares inferior y superior de A, respectivamente [6].

Dentro de cada clase hay varias técnicas diferentes, dependiendo del algoritmo utilizado para calcular la inversa aproximada o los factores inversos aproximados. En la actualidad, existen dos enfoques principales: la minimización de la norma de Frobenius, y la (bi-)conjugación incompleta.

La primera clase de técnicas de inversas aproximadas en ser propuesta e investigada, se basa en la minimización de la norma de Frobenius [5]. La idea básica consiste en calcular una matriz dispersa $M \approx A^{-1}$ como la solución del problema de minimización restringido

$$\min_{M \in \mathcal{S}} \|I - AM\|_F ,$$

donde $\mathcal{S}$ es un conjunto de matrices dispersas y $\|.\|_F$ denota la norma de Frobenius de una matriz. Ya que

$$\|I - AM\|_F^2 = \sum_{j=1}^{n} \|e_j - Am_j\|_2^2 ,$$

donde e_j denota $j-$ésima columna de la matriz identidad, el cálculo de M se reduce a resolver n problemas de mínimos cuadrados lineales e independientes, sujetos a ciertas restricciones de dispersión [6].

Hay que tener en cuenta que el enfoque anterior genera una inversa aproximada por la derecha. Un inversa aproximada por la izquierda se puede calcular mediante la resolución de un problema de minimización restringida para $\|I - MA\|_F = \left\|I - A^T M^T\right\|$. Esto equivale a calcular una inversa aproximada por la derecha para A^T y tomar la transpuesta de la matriz resultante. En el caso de matrices no simétricas, la distinción entre inversas aproximadas por la izquierda y por la derecha puede ser importante. De hecho, hay situaciones en las que es difícil calcular una buena inversa aproximada por la derecha pero fácil de encontrar una buena inversa aproximada por la izquierda. Por otra parte, cuando A es no simétrica y mal condicionada, la matriz $M \approx A^{-1}$ puede ser una pobre inversa aproximada por la derecha, pero una buena inversa aproximada por la izquierda. En la discusión siguiente, se supone que se está calculando una inversa aproximada por la derecha.

1.3.1. SPAI para un Patrón de Dispersión Dado

En los primeros trabajos, el conjunto de restricción $\mathcal{S}$, consistía en un conjunto de matrices con un patrón de dispersión prescrito de antemano. Una vez que el conjunto de restricción $\mathcal{S}$ está dado, el cálculo de M es sencillo, y es posible llevar a cabo dicho cálculo de manera eficiente en un computador paralelo [6].

Dado el patrón no cero $\mathcal{G} \subseteq \{(i,j)|1 \leq i,j \leq n\}$ tal que $m_{i,j} = 0$ si $(i,j) \notin \mathcal{G}$. Por lo tanto, el conjunto de restricción $\mathcal{S}$ es simplemente el conjunto de todas las matrices reales de orden $n \times n$ con el patrón no cero contenido en $\mathcal{G}$. Denotando por m_j la $j-$ésima columna de M $(1 \leq j \leq n)$. Para un indice j fijo, considere el conjunto

$\mathcal{J} = \{i|(i,j) \in \mathcal{G}\}$, que especifica el patrón no cero de m_j. Es evidente que las únicas columnas de A que entran en la definición de m_j son aquellas cuyos índices están en $\mathcal{J}$. Sea $A(:, \mathcal{J})$ la submatriz de A formada por tales columnas, y sea $\mathcal{I}$ el conjunto de índices de las filas distintas de cero de $A(:, \mathcal{J})$. Entonces, se puede restringir nuestra atención a la matriz $\hat{A} = A(\mathcal{I}, \mathcal{J})$, al vector incógnita $\hat{m}_j = m_j(\mathcal{J})$, y al vector del lado derecho $\hat{e}_j = e_j(\mathcal{I})$. Las entradas no cero en m_j se pueden calcular resolviendo el (pequeño) problema de mínimos cuadrados sin restricciones

$$\min_{\hat{m}_j} \left\| \hat{e}_j - \hat{A}\hat{m}_j \right\|_2 .$$

Este problema de mínimos cuadrados se puede resolver, por ejemplo, por medio de la factorización QR de $\hat{A}$. Es evidente que cada columna m_j se puede calcular, al menos en principio, independientemente de las otras columnas de M. Hay que tener en cuenta que debido a la dispersión de A, la submatriz $\hat{A}$ contendrá sólo unas pocas filas y columnas no cero, de manera que, el problema de mínimos cuadrados tiene tamaño pequeño y se puede resolver de manera eficiente por las técnicas de matrices densas [6].

La principal dificultad de este enfoque es la escogencia de $\mathcal{S}$, es decir, cómo elegir un patrón de dispersión para M que se traduzca en un buen precondicionador. Para problemas simples, es común imponerle a M el patrón de dispersión de la matriz original A. Otra idea es tomar el patrón de dispersión de A^k, donde $k \geq 2$ es un entero [6].

Capítulo 2

Programación en GPU

Con la llegada de las CPUs de múltiples núcleos y las GPUs de muchos núcleos, significa que la mayoría de los procesadores de hoy en día son sistemas paralelos. Más aún, su paralelismo continúa creciendo según la Ley de Moore. Es por ello, que en noviembre de 2006 la compañia NVIDIA introduce CUDA[1] como una arquitectura de propósito general de cálculo paralelo. La cual, a través de un nuevo modelo de programación y conjunto de instrucciones, permite un incremento importante en el rendimiento del sistema [10].

El modelo de programación de la arquitectura CUDA está diseñado para aprovechar al máximo, de forma transparente para el desarrollador, el creciente número de núcleos del procesador. Para ello, CUDA provee como base tres abstracciones claves: una jerarquía de bloques de hilos, memorias compartidas y barreras de sincronización. Las cuales guían al programador a dividir el problema en sub-problemas grandes que pueden ser resueltos en paralelo por bloques de hilos independientes, y cada sub-problema en piezas más pequeñas que se pueden resolver cooperativamente en paralelo por los hilos de un mismo bloque [10].

Esta descomposición permite la escalabilidad automática, ya que cada bloque de hilos se puede planificar en cualquiera de los núcleos de los procesadores disponibles, en cualquier orden, simultánea o secuencialmente. Por lo tanto, un programa compilado CUDA se puede ejecutar en cualquier número de núcleos de procesador, como se ilustra

[1] Arquitectura de dispositivos de cómputo unificado (Compute Unified Device Architecture)

en la Figura 2.1, y sólo el sistema de ejecución necesita saber el número de procesadores físicos [10].

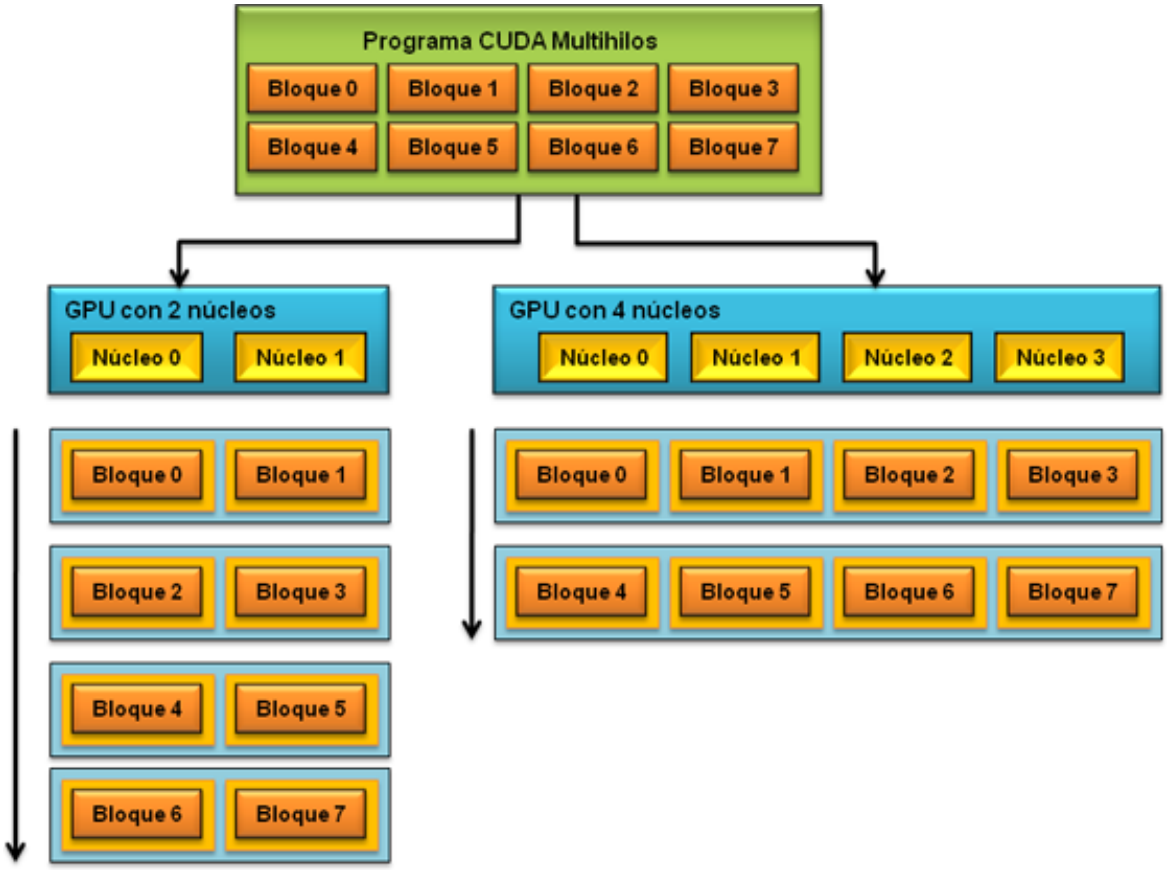


Figura 2.1: Escalabilidad en CUDA: un programa multihilos es particionado en bloques de hilos independientes, de forma que un GPU con más núcleos de ejecución automáticamente ejecuta el mismo programa en menos tiempo

Las aplicaciones en el modelo de programación CUDA consisten de un programa *anfitrión* secuencial, el cual invoca funciones paralelas llamadas *kernels*[2]. Estas funciones, al ser invocadas, se ejecutan N veces en paralelo por N hilos en un *dispositivo* paralelo físicamente separado [10]. Este es el caso cuando los kernels se ejecutan en la GPU y el resto del programa se ejecuta en la CPU, la Figura 2.2 muestra la ejecución común de un programa bajo éste modelo.

[2]El significado de Kernel en español es núcleo, para evitar ambigüedades con los núcleos de ejecución (execution core) se decidió no hacer la traducción y así mantener la consistencia con la documentación existente.

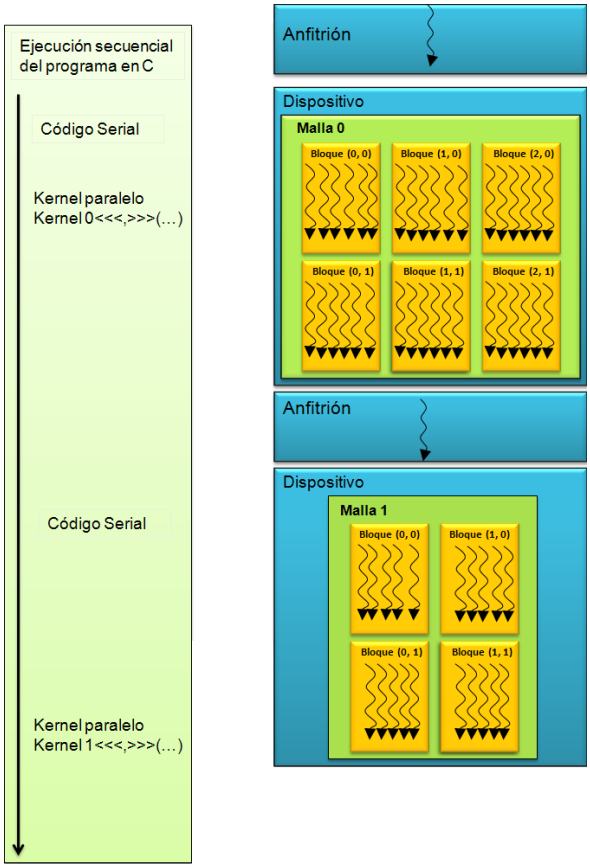


Figura 2.2: Programación Heterogénea

2.1. Arquitectura

Intrínsecamente, una GPU se compone de varios multiprocesadores que se comportan como grandes máquinas SIMD[3]. Más exactamente, cada procesador utiliza una arquitectura SIMT[4] diseñada para ejecutar cientos de hilos al mismo tiempo. En dicha arquitectura, la unidad básica de programación es el hilo, es decir, cada hilo ejecuta de forma secuencial la misma función escalar. El programador organiza los hilos que han de ejecutar un kernel en bloques de hilos, los cuales se planifican en la GPU para ser ejecutados en los multiprocesadores, sin garantía de orden o posibilidad de sincronización entre ellos [10]. Esto implica que, el programador debe tener cuidado al

[3]Simple instrucción, múltiples datos (single instruction, multiple data)
[4]Una instrucción, múltiples hilos (Single Instruction, Multiple Thread)

construir los bloques sin dependencia de ejecución y de no escribir datos en la misma dirección [12].

Sin embargo, ya que todos los hilos de un bloque deben compartir los recursos de memoria limitada de un mismo núcleo de ejecución, existe un límite para el número de hilos por bloque[5]. Por lo cual, un kernel es ejecutado por una *malla* de bloques de hilos homogéneos, como se muestra en la Figura 2.3, de modo que el número total de hilos es igual al número de hilos por bloque multiplicado por el número de bloques de hilos [10].

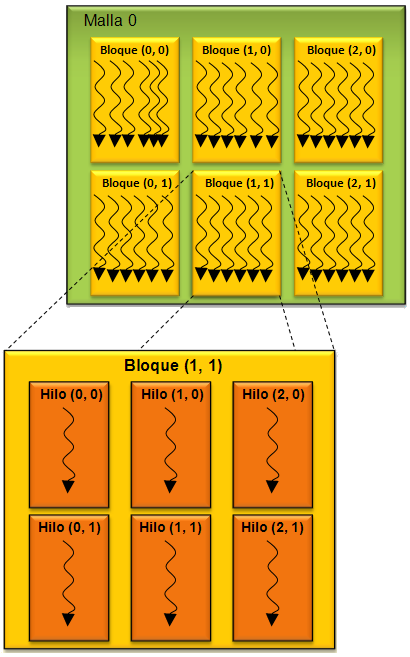


Figura 2.3: Malla de Bloques de Hilos

Los hilos dentro de un bloque pueden cooperar utilizando memoria compartida y sincronizando su ejecución para coordinar los accesos a memoria. Más precisamente, se pueden especificar puntos de sincronización dentro del kernel llamando a la función $\_\_syncthreads()$; los cuales actúan como una barrera que todos los hilos del bloque deben alcanzar antes de que alguno pueda proseguir. Por motivos de eficiencia la memoria compartida debe ser una memoria de latencia baja dentro del chip (como una cache de primer nivel) y la función $\_\_syncthreads()$ debe ser ligera [10].

[5]En las GPUs modernas, un bloque de hilo puede contener hasta 1024 hilos.

En la arquitectura SIMT, el multiprocesador, crea, maneja, planifica y ejecuta los hilos en grupos de 32 hilos paralelos llamados *warps*. Cuando un multiprocesador recibe un bloque de hilos, este lo particiona en warps de forma que cada uno contenga hilos con identificadores consecutivos y crecientes, donde el primer warp recibe el hilo con identificador cero. Finalmente, en el procesador la unidad de planificación es el warp, por lo tanto, todos los hilos de un warp ejecutan una instrucción común a la vez [10].

Los hilos que componen un warp comienzan su ejecución al mismo tiempo y en la misma dirección de programa, pero tienen su propio contador de programa y estado de registros, por lo que son libres de divergir y seguir ramas de ejecución independientes. Sin embargo, si los hilos de un warp divergen por medio de ramificaciones condicionales, el warp ejecuta cada ruta de forma secuencial, deshabilitando los hilos que no pertenecen a la ruta actual. Cuando todos los caminos terminan, los hilos convergen de nuevo a la misma ruta de ejecución, por lo que la mayor eficiencia se consigue cuando todos los hilos de un warp siguen el mismo camino de ejecución. La divergencia de ramas ocurre únicamente dentro de un warp, ya que, los diferentes warps de un mismo bloque se ejecutan de forma independiente sin importar sus rutas de ejecución [10].

Finalmente, los hilos de CUDA tienen acceso a una variedad de espacios de memoria durante su ejecución, como se muestra en la Figura 2.4. Estas memorias difieren en: alcance, tiempo de vida, tamaño y latencia. En el multiprocesador hay una cantidad de registros de 32 bits que son distribuidos entre todos los hilos en ejecución. Estos registros poseen un tiempo de vida igual a la del hilo y su latencia es despreciable. Cada hilo tiene una memoria local privada que esta fuera del chip, específicamente en DRAM. Esta memoria local tiene latencia alta y posee el mismo tiempo de vida del hilo, en esta es donde se manejan las variables que no caben en los registros. Todos los hilos de un bloque tienen acceso a una memoria compartida, la cual perdura durante la ejecución del bloque completo; esta es una memoria pequeña, con baja latencia y alto ancho de banda; todos los hilos del kernel tienen acceso a la memoria global, que es una memoria grande y con alta latencia. Además, todos los hilos tienen acceso a dos espacios de memoria de solo lectura: los espacios de memoria constante y de texturas.

Los datos en memoria global, constante y de textura persisten a lo largo del tiempo de vida de la aplicación [10].

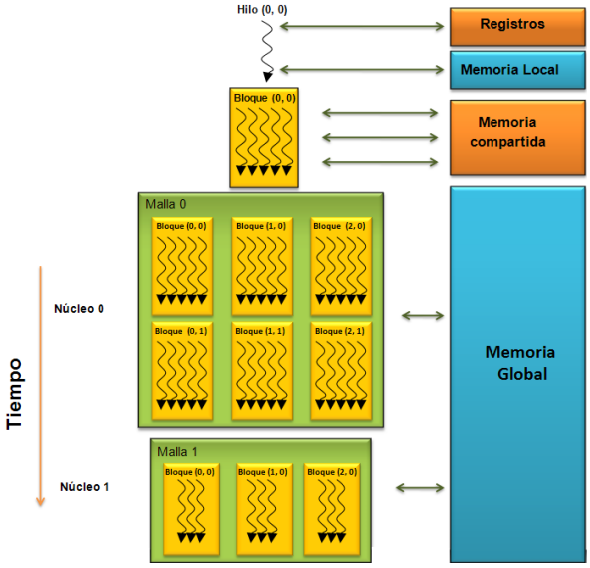


Figura 2.4: Jerarquía de memoria CUDA

2.2. Codificación en CUDA

CUDA viene con un ambiente de software que permite a los desarrolladores utilizar C como lenguaje de alto nivel. Sin embargo, como se muestra en la figura 2.5 es posible utilizar otros lenguajes e interfaces de programación de aplicaciones. Indistintamente del lenguaje utilizado por el programador, se deben utilizar las extensiones especiales definidas por CUDA.

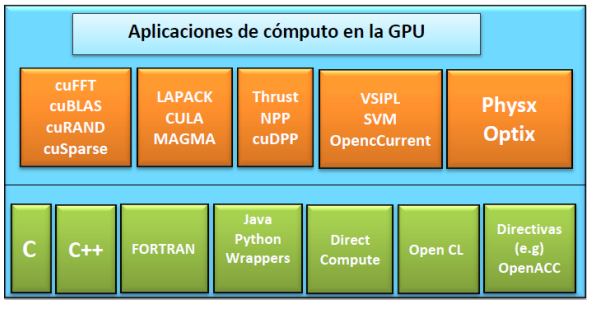


Figura 2.5: Aplicaciones de cómputo en la GPU

Con el fin de ilustrar el diseño e implementación de soluciones en el ambiente CUDA, incluyendo detalles, trucos y errores comunes; a continuación se detalla el proceso para dos operaciones importantes en el álgebra lineal.

Ejemplo: Operación AXPY

Recordando, las actualizaciones de vectores se conocen como operación (*axpy*) y son de la forma $y := \alpha * x + y$, dónde x e y son vectores reales de tamaño N y α es un escalar. En la Figura 2.6 se muestra su implementación en un ambiente tradicional.

```
void axpy_normal(const int N, const float alpha, const float *x, float *y)
{
  for (int i = 0; i < N; ++i)
    y[i] += alpha * x[i];
}
```

Figura 2.6: Axpy secuencial

Allí se observa que se realizan N operaciones iguales sobre diferentes elementos de datos, por lo tanto, en un ambiente CUDA, estas operaciones pueden ser realizadas por N hilos concurrentemente sin necesidad de coordinación, como se muestra en la Figura 2.7.

```
__global__
void axpy_kernel(const int N, const float alpha, const float *x, float *y)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N)
    y[i] += alpha * x[i];
}
```

Figura 2.7: Axpy kernel

Los kernels han de ser etiquetados con el modificador **__global__**, que permite declarar la función que será ejecutada por el dispositivo y convocada por el anfitrión. La primera línea del kernel calcula el índice único de cada hilo, el cual se utiliza para acceder tanto a x como y. De esta manera, cada hilo actualiza una componente del vector. Como los datos son vectores, sólo necesitan bloques y mallas de una dimensión. El tamaño del bloque es elegido por el valor máximo permitido por la arquitectura, mientras que el tamaño de la malla se determina directamente por el tamaño del

problema dividido entre el tamaño del bloque. El cálculo del tamaño de la malla y la invocación del kernel se puede ver en la Figura 2.8.

```
void axpy_driver(const int N, const float alpha, const float *x, float *y)
{
  int hilosPorBloque = 512;
  int bloquesDeHilos = iDivTecho(N, hilosPorBloque);
  axpy_kernel<<<bloquesDeHilos, hilosPorBloque>>>(N, alpha, x, y);
}
```

Figura 2.8: Manejador axpy

El tamaño de la malla viene dado por $f(N, B) = \left\lceil \frac{N}{B} \right\rceil$, donde N es el tamaño del problema y B es la cantidad de hilos por bloque. Dicha operación se puede ver implementada en la Figura 2.9. Como se puede ver, la función ha sido marcada con los modificadores __**device**__ y __**host**__, lo que significa que puede ser invocada tanto por hilos de CUDA como por hilos de CPU.

```
__device__ __host__
int iDivTecho(int a, int b)
{
  return (a + b - 1) / b;
}
```

Figura 2.9: División techo

Ejemplo: Producto interno

Al igual que con el ejemplo de la sección anterior, se comienza con la definición matemática del producto interno, $\langle x, y \rangle := \sum_{i=1}^{N} x_i y_i$. Dicha definición conduce naturalmente a la función mostrada en la Figura 2.10 en un ambiente secuencial.

```
float dot_normal(const int N, const float *x, const float *y)
{
  float resultado = 0;
  for (int i = 0; i < N; ++i)
    resultado += x[i] * y[i];
  return resultado;
}
```

Figura 2.10: Producto punto secuencial

Transformando el ciclo de manera similar a como se hizo en la operación axpy, se obtiene el kernel de CUDA de la Figura 2.11. Sin embargo, analizando en detalle el comportamiento de dicho kernel se puede observar que todos los hilos modifican la misma posición de memoria. Cuando esto sucede los accesos son serializados en algún orden arbitrario, más aún, como la operación no es atómica, ésta podría fallar y causar resultados inconsistentes e impredecibles [12].

```
__global__
void dot_kernel_malo(const int N, const float *x, const float *y, float *res)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N)
    *res += x[i] * y[i];
}
```

Figura 2.11: Producto punto kernel malo

Este ejemplo muestra, que la dificultad de programar en la arquitectura CUDA radica principalmente en la descomposición del problema en subproblemas que pueden ser resueltos independientemente por los bloques de hilos. Además, la cooperación sólo puede existir entre hilos de un mismo bloque. Por lo tanto, existen dos opciones, la primera es realizar la operación completa con un sólo bloque de hilos. Sin embargo, para aprovechar el paralelismo al máximo se puede dividir el problema en dos subproblemas, de manera que uno puede ser realizado por un kernel que no requiere cooperación. De esta forma, sólo la sección que requiere cooperación es realizada por un único bloque de hilos cooperativos.

Específicamente, definiendo z como un vector tal que, $z_i = x_i y_i$, entonces se tiene $\langle x, y \rangle = \sum_{i=1}^{N} x_i y_i = \sum_{i=1}^{N} z_i$. Por lo tanto, se puede utilizar un kernel no cooperativo como el mostrado en la Figura 2.12 para calcular los valores de z y un kernel cooperativo para realizar la suma de los valores del vector, el cual se puede ver en la Figura 2.13.

El primer kernel no introduce novedad con respecto a la operación axpy, por lo tanto, se explicará únicamente en detalle el segundo kernel. Para realizar la suma de los valores se utiliza una malla compuesta por un único bloque de M hilos cooperativos. Cada uno de estos hilos, se encargará de acumular un subconjunto de los valores

```
__global__
void dot_kernel_parte1(const int N, const float *x, const float *y, float *z)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N)
    z[i] = x[i] * y[i];
}
```

Figura 2.12: Producto punto kernel parte1

de z en una posición de memoria compartida, la cual se declara con el modificador **__shared__**. Cuando todos los hilos han terminado dicha acumulación, se procede a reducir las M sumas parciales a un sólo valor. La idea es aprovechar el paralelismo durante la reducción, específicamente, realizar aproximadamente la mitad de las sumas restantes en cada paso.

Una forma intuitiva de realizar la acumulación parcial en memoria compartida es que cada hilo sume sobre un rango contiguo del vector, esta asignación se encuentra ilustrada en la Figura 2.14. Sin embargo, cuando un medio warp accede a posiciones diferentes pero contiguas de memoria, es posible que los 16 accesos se realicen simultáneamente. Por lo tanto, una forma mucho más eficiente es hacer que el hilo i acumule las posiciones i, $i + M$, $i + 2M$, ...; de esta manera, todos los hilos de un medio warp acceden a espacios diferentes y contiguos de memoria en cada iteración del ciclo, como puede verse en la Figura 2.15.

Finalmente, para acumular las sumas parciales en un único resultado, se tiene un ciclo que suma dos posiciones de la memoria compartida, reduciendo así la cantidad de sumas restantes a la mitad en cada paso. Lógicamente dicha reducción se puede ver en la Figura 2.16.

Para ocultar los detalles de implementación, se puede utilizar un manejador como el de la Figura 2.17. El cual se encarga de solicitar memoria auxiliar de dispositivo para almacenar el vector z, invocar los dos kernels, copiar el resultado de la memoria del dispositivo a la memoria del anfitrión y finalmente liberar la memoria auxiliar. Es importante mencionar que, a pesar de que la invocación de un kernel es una operación asíncrona, la invocación de un kernel se bloquea si el dispositivo ya esta ejecutando un kernel o realizando una transferencia. Dicha sincronización implícita entre las diferentes

```
#define M 512

__shared__
float compartida[M];

__global__
void dot_kernel_parte2(const int N, float *x)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  // Se llena la memoria compartida
  compartida[i] = 0;
  for (int j = i; j < N; j += M)
    compartida[i] += x[j];

  // Sincronizacion para asegurar que la memoria
  // compartida ha sido llenada por completo
  __syncthreads();

  // Se realiza la reduccion a un valor
  for (int faltan = M; faltan > 1; faltan -= faltan / 2)
  {
    int otro = i + iDivTecho(faltan, 2);
    if (otro < faltan)
      compartida[i] += compartida[otro];

    // Sincronizacion para asegurar que la memoria
    // compartida ha sido llenada por completo
    __syncthreads();
  }
  if (i == 0)
    x[0] = compartida[0];
}
```

Figura 2.13: Producto punto parte2

invocaciones y transferencias garantiza que el primer kernel calcula el vector z en su plenitud antes de comenzar a realizar la reducción.

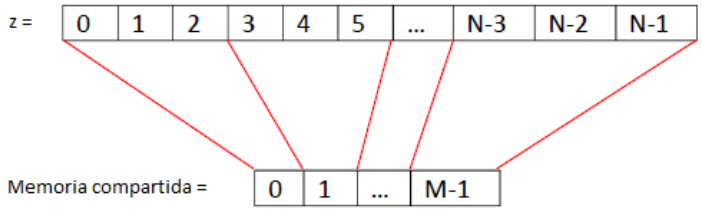


Figura 2.14: Cada hilo suma sobre un rango contiguo del vector

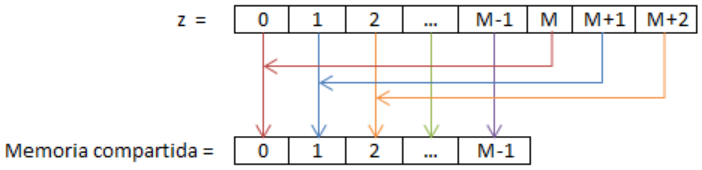


Figura 2.15: El hilo i acumula las posiciones i, $i+M$, $i+2M$, ...; de manera que, todos los hilos de medio warp acceden a espacios diferentes y contiguos de memoria por cada iteración del ciclo

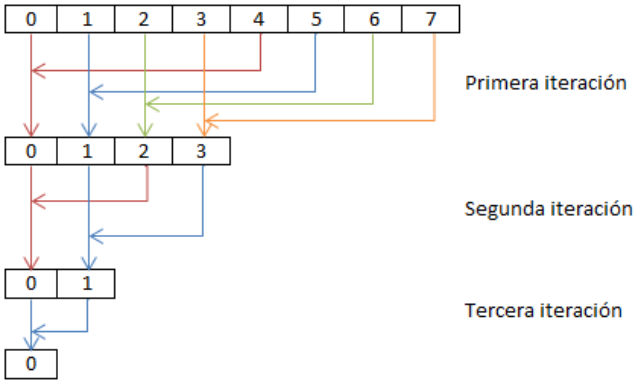


Figura 2.16: Reducción a través de sumas parciales acumuladas en un único resultado

```
float dot_driver(const int N, const float *x, const float *y)
{
  int hilosPorBloque = M;
  int bloquesDeHilos = iDivTecho(N, hilosPorBloque);
  float resultado;

  // Se crea suficiente memoria para almacenar el resultado de los dos
  // kernels
  size_t tamano = N * sizeof(float);
  float* aux;
  cudaMalloc((void**)&aux, tamano);
  // Se inicializa la memoria auxiliar en 0
  cudaMemset(aux, 0, tamano);
  dot_kernel_parte1<<<bloquesDeHilos, hilosPorBloque>>>(N, x, y, aux);
  dot_kernel_parte2<<<1, hilosPorBloque>>>(N, aux);
  // Copiamos el resultado de la memoria global del dispositivo a la
  // memoria del anfitrion
  cudaMemcpy(&resultado, aux, sizeof(float), cudaMemcpyDeviceToHost);

  // Liberamos la memoria auxiliar
  cudaFree(aux);
  return resultado;
}
```

Figura 2.17: Producto punto driver

Capítulo 3

Diseño e Implementación

En éste capítulo se presentan las decisiones y los aspectos más relevantes que fueron considerados para el diseño y su posterior implementación del precondicionador y métodos propuestos. A continuación se describen los diferentes componentes realizados en este trabajo, cuya interacción puede ser vista en la Figura 3.1. Para cumplir con la propuesta del trabajo especial de grado se desarrolló una biblioteca de plantillas llamada *CuspUCV*, la cual contiene los métodos iterativos y precondicionadores implementados. Dicha biblioteca utiliza los algoritmos y estructuras de datos provistos en la biblioteca Thrust que son similares a los que provee la biblioteca estándar de C++ para un ambiente paralelo, así como también se utilizan las operaciones de álgebra lineal altamente optimizadas implementadas en la biblioteca Cusp.

Tanto Thrust como Cusp ofrecen gran flexibilidad debido a su diseño, el cual utiliza plantillas de C++ (templates) para abstraer a la implementación de la plataforma subyacente. De esta forma, al implementar un método iterativo se obtiene la posibilidad de ejecutarlo en una variedad de plataformas diferentes. Específicamente, las estructuras y algoritmos son etiquetadas como anfitrión o dispositivo, donde el anfitrión puede ser C++, OpenMP o TBB; y el dispositivo puede ser CUDA, OpenMP o TBB. Más aún, Cusp utiliza el poder de las plantillas para ofrecer mayores niveles de abstracción, por ejemplo, los métodos iterativos se abstraen del formato de compresión de la matriz y el tipo de precondicionador utilizado.

Al ofrecer dichas abstracciones por medio de plantillas en lugar de métodos virtuales se garantiza el mejor rendimiento posible, ya que el trabajo fuerte se realiza en

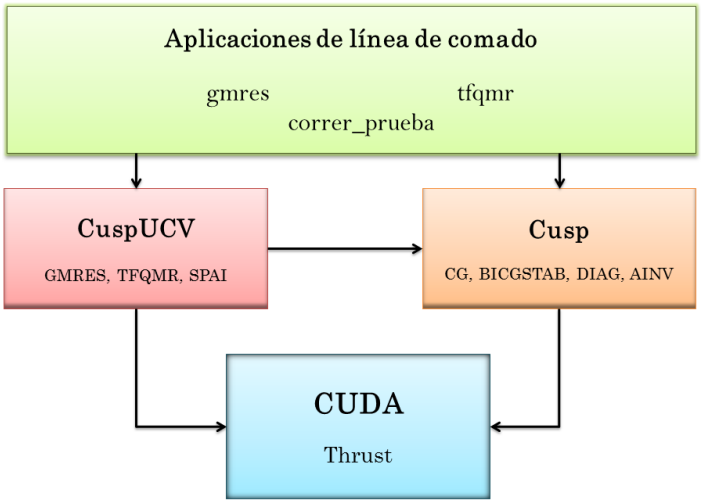


Figura 3.1: Componentes desarrollados

la etapa de compilación. Sin embargo, el uso de plantillas imponen ciertas restricciones, siendo la más relevante el no poder tomar decisiones en tiempo de ejecución. Por lo tanto, todas las combinaciones de plataformas, formatos de compresión, precondicionadores, etcétera; que se deseen ofrecer han de ser compiladas de antemano.

Para obtener un rendimiento competitivo e integración completa entre los algoritmos desarrollados y los ofrecidos por Cusp, fue necesario regir el diseño de CuspUCV por el que utiliza Cusp. Por lo tanto, CuspUCV es una biblioteca de plantillas que cuenta con las mismas ventajas y desventajas que Cusp.

Para facilitar la resolución de sistemas de ecuaciones lineales utilizando los métodos implementados, se desarrollaron dos aplicaciones de línea de comando: *tfqmr* y *gmres*. Estas aplicaciones permiten que el usuario especifique diferentes parámetros que se dividen en tres categorías. Primero, la Tabla 3.1 muestra los parámetros propios de los métodos; en la Tabla 3.2 se especifican los parámetros que controlan los resultados obtenidos, finalmente, los parámetros que definen el ambiente de ejecución se encuentran en la Tabla 3.3.

Por último, para realizar las pruebas de tiempo y estabilidad de los algoritmos propuestos en el Capítulo 4, se desarrolló un programa llamado correr_prueba, el cual recibe todos los parámetros necesarios para ejecutar una prueba y genera un archivo

Parámetro	Descripción	Por Defecto
A	Ruta del archivo con la matriz de coeficientes en formato MatrixMarket. Este parámetro es obligatorio.	
b	Ruta del archivo con la matriz de coeficientes en formato MatrixMarket o la palabra "ones" para todos los valores en uno.	ones
restart	Parámetro que indica cada cuantas iteraciones el método es reinicializado. NOTA: Este parámetro sólo aplica para el método gmres.	mín$(50, N)$
tol	Indica la tolerancia relativa permitida como criterio de convergencia.	1e-6
maxit	Representa el máximo número de iteraciones permitidas para el método.	500
precond	Ruta del archivo con la matriz precondionadora en formato MatrixMarket o uno de los siguientes valores: • **identity**: utiliza la matriz identidad como precondicionador. • **spai**: construye un precondicionador de tipo SPAI. • **ainv**: construye un precondicionador de tipo AINV. • **diag**: construye un precondicionador tipo DIAGONAL.	identity
x0	Ruta del archivo en formato MatrixMarket con el vector inicial o la palabra "zeros" para indicar como iterado inicial al vector nulo.	zeros

Tabla 3.1: Parámetros propios de los métodos aceptados por las aplicaciones

Parámetro	Descripción	Por Defecto
sol	Ruta del archivo donde se colocará la solución aproximada obtenida o la palabra "stdout" para mostrar la aproximación por la pantalla.	stdout

Tabla 3.2: Parámetros de los resultados aceptados por las aplicaciones

con todos los resultados necesarios para su posterior análisis. Esto fue necesario debido a que para tomar los tiempos de forma acertada en CUDA es necesario sincronizar los hilos de la CPU y la GPU lo cual disminuye su paralelismo.

Parámetro	Descripción	Por Defecto
proc	Procesador a utilizar, debe ser uno de los siguientes valores: • **gpu**: utiliza la arquitectura CUDA para ejecutar el método en la GPU en paralelo. • **cpu**: utiliza la CPU para ejecutar el método secuencialmente.	gpu
monitor	Indica el tipo de monitor que observa el progreso de los métodos y verifica convergencia„ debe ser uno de los siguientes valores: • **default**: detiene el método cuando la norma del residual satisface $\|b - Ax\|_2 \leq \|b\|_2 * tol$. • **verbose**: detiene el método cuando la norma del residual satisface $\|b - Ax\|_2 \leq \|b\|_2 * tol$. Muestra la norma del residual en cada iteración. • **convergence**: detiene el método cuando la norma del residual satisface $\|b - Ax\|_2 \leq \|b\|_2 * tol$. Muestra la norma del residual en cada iteración. Muestra información de la tasa de convergencia al finalizar.	default

Tabla 3.3: Parámetros del ambiente de ejecución aceptados por las aplicaciones

3.1. Detalles de Implementación

En esta sección se explica en detalle la implementación de cada uno de los algoritmos desarrollados. Específicamente se hace énfasis en las estrategia tomadas para que las implementaciones sean lo más robusta y eficientes posibles, sin embargo, algunos de los detalles son consecuencias de la plataforma utilizada.

Varios autores han propuesto que los métodos iterativos obtienen grandes ganancias en rendimiento en ambientes paralelos sin ser modificados directamente, simplemente paralelizando los productos matriz vector, productos internos y actualizaciones de vectores [26][23]. Este es el enfoque tomado en este trabajo, por lo tanto, para implementar los métodos no se requieren grandes modificaciones con respecto a las versiones teóricas mostradas en el capítulo 1. Sin embargo, ciertos detalles han de ser tomados en cuenta para garantizar que se tienen implementaciones robustas y eficientes.

3.1.1. GMRES

Espacio de Trabajo del Anfitrión

Observando los Pasos 9 y 12 del Algoritmo 1.2 queda claro que la matriz de Hessenberg requiere que sus componentes sean accedidos eficientemente de forma aleatoria. Sin embargo, en CUDA, este tipo de accesos en la memoria global del dispositivo son costosos. Además, el proceso de sustitución hacia atrás que debe ser realizado para actualizar la solución x_m, es secuencial por naturaleza. Por lo tanto, se decidió mantener dicha matriz en la memoria del anfitrión, sin importar donde se estén ejecutando los productos matriz-vector.

Para aplicar las rotaciones de Givens progresivamente, es necesario mantener en memoria todos los coeficientes c_i y s_i asociados a dichas rotaciones. Además, es necesario aplicar las rotaciones tanto a la matriz de Hessenberg como al vector del lado derecho g_m. Tomando en cuenta que la matriz se mantiene en el anfitrión, es necesario mantener dichos coeficientes y vector en la memoria del anfitrión.

Estabilidad Numérica en el Cálculo de las Rotaciones de Givens

Las ecuaciones mostradas en (1.12) para calcular c_i y s_i no son numéricamente estables, ya que el denominador puede sufrir de desbordamiento [19]. Para mostrar el desarrollo de una alternativa más estable, por motivos de legibilidad, se reescriben las ecuaciones sustituyendo $h_{i,i}^{(i-1)}$ por a y $h_{i+1,i}$ por b; adicionalmente, se omiten los subíndices de c y s intencionalmente. Por lo tanto, la ecuación (1.12) se reescribe como

$$s = \frac{b}{\sqrt{a^2 + b^2}},\ c = \frac{a}{\sqrt{a^2 + b^2}}. \tag{3.1}$$

Definiendo

$$t = \frac{a}{b} \tag{3.2}$$

se tiene que

$$s = \frac{1}{\sqrt{t^2 + 1}} \tag{3.3}$$

y

$$c = ts. \tag{3.4}$$

De forma análoga, definiendo

$$u = \frac{b}{a} \tag{3.5}$$

se obtiene

$$c = \frac{1}{\sqrt{u^2+1}} \tag{3.6}$$

y

$$s = uc. \tag{3.7}$$

Finalmente, para obtener los coeficientes de rotación con la mayor estabilidad numérica posible se utilizan las ecuaciones (3.2), (3.3) y (3.4) cuando $|a| < |b|$; y las ecuaciones (3.5), (3.6) y (3.7) en caso contrario, es decir, si $|a| \geq |b|$.

Aplicación Progresiva de las Rotaciones de Givens

La aplicación progresiva de las rotaciones de Givens requiere que, al obtener una nueva columna h_j se le aplique las rotaciones anteriores. Luego, se calculan los coeficientes de la nueva rotación como se menciona anteriormente, y finalmente, la nueva rotación es aplicada tanto a la columna h_j como al vector del lado derecho g. Por lo tanto, cada vez que una rotación es aplicada, se hace únicamente sobre un vector. Afortunadamente, dicha aplicación puede realizarse de forma eficiente analizando la estructura de la matriz mostrada en (1.11), específicamente

$$\begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & s_i & & & \\ & & & -s_i & c_i & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_{i-1} \\ x_i \\ x_{i+1} \\ x_{i+2} \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} x_1 \\ \vdots \\ x_{i-1} \\ c_i x_i + s_i x_{i+1} \\ -s_i x_i + c_i x_{i+1} \\ x_{i+2} \\ \vdots \\ x_n \end{pmatrix} \tag{3.8}$$

La ecuación (3.8) muestra claramente que, aplicar una rotación de Givens a un vector requiere únicamente modificar dos de sus componentes. Además, no es necesario crear la matriz Ω_i explícitamente, ya que únicamente los coeficientes y el índice i son necesarios. El Algoritmo 3.1 muestra como puede implementarse un procedimiento para aplicar una rotación de Givens a un vector. La variable temporal del paso 2 es necesaria porque en el paso 3 se utiliza el valor de x_i original.

1. Procedimiento AplicarRotación(i, c, s, x)
2. | temp $:= c_i x_i + s_i x_{i+1}$
3. | $x_{i+1} := -s_i x_i + c_i x_{i+1}$
4. | $x_i :=$ temp
5. FinProcedimiento

Algoritmo 3.1: Aplicación de una rotación de Givens a un vector

Monitor de Cusp y la Norma del Residual

Los métodos iterativos de la biblioteca Cusp utilizan un monitor para determinar si se ha alcanzado algún criterio de parada, en realidad, el monitor es cualquier clase que implemente la interfaz adecuada. Con la intención de permitir que los monitores implementen diferentes criterios de parada basados en el residual, la interfaz requiere que en lugar de la norma del residual, el vector completo le sea suministrado. Sin embargo, el método de GMRES con rotaciones de Givens progresivas es capaz de obtener únicamente la norma del residual en cada paso sin costo adicional. Por lo tanto, fue necesario circunvenir la interfaz generando un vector residual falso, el cual debía tener la misma norma que el residual real.

Dado que

$$\|b - Ax\|_2 = |g_{j+1}|\,,$$

es claro que el vector más sencillo que cumple con la característica deseada es el "vector" de una dimensión cuya única componente es g_{j+1}. Desde un práctico, ya que

dicho vector sólo posee una componente y se mantiene en la memoria del anfitrión, su uso no tiene un costo mayor que el de un escalar corriente. Sin embargo, el cálculo de la norma del residual, realizado automáticamente por el monitor, requiere de un producto y una raíz cuadrada en lugar de un único cálculo de valor absoluto.

Costo de Cómputo y Memoria

El costo de cómputo del método GMRES precondicionado por la derecha se puede obtener contando las operaciones más relevantes del método y tomando en cuenta el orden de ejecución de cada una de ellas. La Tabla 3.4 muestra la cantidad y el orden de ejecución acumulado de las operaciones más importantes. Además, muestra que el orden de ejecución del método es $O(m[mn + nnz(A) + nnz(M)])$. Lo cual muestra porque el método se vuelve prohibitivo a medida que m crece, por lo tanto, la versión implementada en la biblioteca CuspUCV incluye reinicialización de acuerdo al parámetro de reinicio (restart).

Operación	**Cantidad**	$O_{\mathbf{FLOPS}}$
Productos matriz-vector con A	$m+1$	$O(m\ nnz(A))$
Productos matriz-vector con M	$m+1$	$O(m\ nnz(M))$
Actualizaciones de vectores	$\frac{m(m+1)}{2} + m + 2$	$O(m^2 n)$
Productos internos	$\frac{m(m+1)}{2}$	$O(m^2 n)$
Cálculos de norma	$m+1$	$O(mn)$
Rotaciones de Givens	$\frac{m(m+1)}{2} + m$	$O(m^2)$
Sustitución hacia atrás	1	$O(m^2)$
		$O(m[mn + nnz(A) + nnz(M)])$

Tabla 3.4: Costo computacional de GMRES precondicionado por la derecha.

Como se mencionó anteriormente, parte del espacio de trabajo es almacenado en el anfitrión sin importar donde se estén realizando las operaciones paralelizables. Por lo tanto, el costo en memoria se divide en dos tablas, la Tabla 3.5 muestra el espacio de

trabajo del anfitrión; mientras que la Tabla 3.6 indica el espacio de trabajo requerido en el dispositivo [1].

Nombre	**Descripción**	**Costo**
$\bar{H}_m$	Matriz de Hessenberg	$(m+1) \times m$
c, s	Vectores de longitud m	$2 \times m$
g	Vector de longitud $m+1$	$m+1$
		$m \times (m+4)+1$

Tabla 3.5: Costo en memoria de anfitrión de GMRES.

Nombre	**Descripción**	**Costo**
V_{m+1}	Matriz	$n \times (m+1)$
w, y	Vectores de longitud n	$2 \times n$
		$n \times (m+3)$

Tabla 3.6: Costo en memoria de dispositivo de GMRES.

3.1.2. TFQMR

Eliminación de los Subíndices

Para implementar el Algoritmo 1.4 de forma eficiente es necesario eliminar los subíndices teóricos, es decir, se deben reemplazar los valores de los vectores y escalares sobre la misma variable en la medida de lo posible. Analizando cada instrucción, se puede observar que únicamente los pasos 10, 19 y 21 causan problemas y han de ser tratados con cuidado.

El primer problema se debe a que en el paso 12 es necesario el valor de u_m, lo que impide actualizar el valor del vector u en el paso 10. Sin embargo, es fácil ver que el valor de u_m no se necesita en ningún otro momento de las iteraciones pares.

[1] Cuando se está utilizando el anfitrión para todas las operaciones, ambos espacios de trabajo residen en él.

Desafortunadamente, no es posible realizar la actualización de w antes del condicional porque el paso 12 requiere el valor de α_m que se calcula en el paso 9. Por lo tanto, la única opción que no desperdicia memoria adicional es mover la actualización del paso 10 a cualquier punto después de la actualización realizada en el paso 12. Es importante que dicha actualización se realice únicamente en las iteraciones pares, una forma eficiente de lograrlo es agregar una cláusula "sino" al condicional impar.

Es trivial ver que el segundo problema no puede ser eliminado cambiando el orden de las instrucciones, esto se debe a que el paso 20 requiere tanto de ρ_{m-1} como de ρ_{m+1}. Afortunadamente, dado que ρ es un escalar, la cantidad de memoria requerida para mantener ambas copias es despreciable. Por lo tanto, la solución empleada es utilizar un escalar ρ_{viejo} al que se le asigna el valor de ρ antes del paso 19, luego, el paso 20 es reemplazado por $\beta := \frac{\rho}{\rho_{\text{viejo}}}$.

El tercer caso es un poco más complejo, ya que en principio parece que el paso 22 requiere de los valores de u_m y u_{m+1} simultáneamente. Sin embargo, es posible dividir la actualización de v en dos, una que utiliza u_m y una que utilza u_{m+1}. De esa forma, es posible realizar la actualización de u entre las dos actualizaciones de v. Finalmente, lo que se debe hacer es reemplazar los pasos 21-22 del Algoritmo 1.4 por

$$\begin{aligned} v &:= AM^{-1}u + \beta v \\ u &:= w + \beta u \\ v &:= AM^{-1}u + \beta v \end{aligned} \tag{3.9}$$

Reutilización de los Productos Matriz Vector

Tal y como se muestra el Algoritmo 1.6, se realizan siete productos matriz-vector por iteración en promedio (cinco en las iteraciones pares y nueve en las iteraciones impares). Sin embargo, algunos de ellos son claramente repetidos mientras que otros pueden ser reescritos en forma de actualizaciones de vectores.

Desglosando el paso 12 en los siguientes tres pasos

$$\begin{aligned} y &:= M^{-1}u \\ z &:= Ay \\ w &:= w - \alpha z, \end{aligned}$$

y notando que los pasos 2 y 21 realizan los mismos productos matriz vector, se desea reutilizar los valores de y y z de alguna manera. La forma más simple es actualizar dichos vectores inmediatamente después de actualizar el vector u, por lo tanto, después de actualizar u se realizan dos productos matriz vector. De esta manera, se garantiza que a lo largo del método se cumpla que

$$y \equiv M^{-1}u_m,\ z \equiv AM^{-1}u_m. \tag{3.10}$$

Dado que el vector u se actualiza exactamente una vez por iteración, se tienen que realizar dos productos matriz vector por iteración para mantener a los vectores y y z actualizados. Por otra parte, el uso de los mismos en los pasos 12 y 21 elimina dos productos en las iteraciones pares y cuatro en las iteraciones impares. Por lo tanto, ahora, en lugar de siete en promedio, se realizan exactamente cinco productos matriz vector por iteración.

Las actualizaciones que aún realizan productos matriz vector son las de los pasos 16 y 17 del Algoritmo 1.6. A simple vista se observa que ambos realizan el producto $M^{-1}d_m$, de forma que, creando un vector $p \equiv p_m = M^{-1}d_m$ queda claro que se puede eliminar un producto matriz vector por iteración. Sin embargo, en lugar de actualizar p con un producto matriz vector cada vez que se actualice d, se puede mantener el valor utilizando únicamente operaciones de actualización de vectores. Para ello es necesario inicializar p con $p_0 = M^{-1}d_0 = M^{-1}0 = 0$, luego, se debe mostrar como obtener p_{m+1} sin realizar ninguna multiplicación matriz vector. Utilizando la definición de p_m, la

actualización del paso 13 del Algoritmo 1.6 y las equivalencias de (3.10) se tiene que

$$\begin{aligned} p_{m+1} &= M^{-1}d_{m+1} \\ &= M^{-1}\left[u_m + \left(\frac{\theta_m^2}{\alpha_m}\right)\eta_m d_m\right] \\ &= M^{-1}u_m + \left(\frac{\theta_m^2}{\alpha_m}\right)\eta_m M^{-1}d_m \\ &= y + \left(\frac{\theta_m^2}{\alpha_m}\right)\eta_m p_m. \end{aligned}$$

Por lo tanto, es posible mantener el vector p actualizado sin realizar productos matriz vector. Además, de forma análoga, es posible mantener un vector $q \equiv q_m = AM^{-1}d_m$ actualizado en cada paso sin realizar productos matriz vector. Al igual que antes, se inicia con el valor de $q_0 = AM^{-1}d_0 = 0$ y se actualiza como se muestra a continuación

$$\begin{aligned} q_{m+1} &= AM^{-1}d_{m+1} \\ &= AM^{-1}\left[u_m + \left(\frac{\theta_m^2}{\alpha_m}\right)\eta_m d_m\right] \\ &= AM^{-1}u_m + \left(\frac{\theta_m^2}{\alpha_m}\right)\eta_m AM^{-1}d_m \\ &= z + \left(\frac{\theta_m^2}{\alpha_m}\right)\eta_m q_m. \end{aligned}$$

Mantener los vectores p y q actualizados, requiere dos actualizaciones de vectores por iteración. Más aún, ahora el vector d no se necesita en ningún momento del algoritmo, por lo que el costo agregado es únicamente una actualización de vector. En cambio, ahora los pasos 16 y 17 pueden ser escritos sin multiplicaciones matriz vector, lo que genera un ahorro de tres productos por iteración. Por lo tanto, la versión final realiza dos multiplicaciones matriz vector por iteración, las necesarias para mantener los vectores y y z actualizados.

Costo de Cómputo y Memoria

De forma análoga al cálculo que se realizó con el método GMRES, es posible calcular el orden de ejecución de algoritmo TFQMR precondicionado por la derecha

contando las operaciones más relevantes del método y tomando en cuenta el orden de ejecución de cada una de ellas. La Tabla 3.7 muestra la cantidad y el orden de ejecución acumulado de dichas operaciones.

Operación	**Cantidad**	$O_{\mathbf{FLOPS}}$
Productos matriz-vector con A	$m+2$	$O(m\ nnz(A))$
Productos matriz-vector con M	$m+1$	$O(m\ nnz(M))$
Actualizaciones de vectores	$7m+1$	$O(mn)$
Productos internos	$m+1$	$O(mn)$
Cálculos de norma	$2m+1$	$O(mn)$
		$O(m[n + nnz(A) + nnz(M)])$

Tabla 3.7: Costo computacional de TFQMR precondicionado por la derecha.

A diferencia de GMRES, la implementación de TFQMR no requiere que parte del espacio de trabajo sea almacenado en el anfitrión. Además, el espacio de trabajo es sencillo de analizar ya que consta únicamente de los siguientes vectores de dimensión n: w, u, r, r^*, v, p, q, y y z; lo que significa que el espacio de trabajo se compone de $9n$ elementos en total.

3.1.3. SPAI

La construcción del precondicionador SPAI M dado un patrón de dispersión $\mathcal{S}$ se puede llevar a cabo eficientemente en paralelo recordando que cada columna de M puede ser calculada independientemente. Una estrategia sencilla para su cómputo paralelo es utilizar N hilos donde cada uno se encarga de calcular una columna m_j. Por lo tanto, cada hilo debe realizar los siguientes pasos:

1. Calcular el conjunto de índices $\mathcal{J}$
2. Calcular el conjunto de indices $\mathcal{I}$
3. Reservar espacio de memoria para las matrices Q y R

4. Obtener la factorización QR de $A(\mathcal{I}, \mathcal{J})$

5. Resolver el sistema triangular $R\hat{m}_j = Q^T \hat{e}_j$

Sin embargo, en el modelo de programación de CUDA, el paso 3 es problemático. Esto se debe a que los hilos de dispositivo no pueden reservar memoria una vez que han comenzado su ejecución, es decir, toda la memoria que requieran ha de ser asignada por el anfitrión antes de la invocación al kernel.

La cantidad de memoria requerida por las matrices Q y R, depende únicamente de los tamaños de los conjuntos $\mathcal{I}$ y $\mathcal{J}$ y no de sus elementos. Por lo tanto, la implementación propuesta se puede resumir en los siguientes pasos:

1. preprocesamiento: Calcular todos los conjuntos $\mathcal{J}$
2. kernel: Calcular los tamaños de todos los conjuntos $\mathcal{I}$
3. host: Calcular la memoria requerida por todos los conjuntos $\mathcal{I}$ y las matrices Q y R
4. host: Crear una partición $1 = x_0 < x_1 < x_2 < \ldots < x_m = N + 1$ del intervalo $[1, N + 1]$
5. Para cada intervalo de columnas $[x_{i-1}, x_i)$, con $i = 1, 2, \ldots, m$; Hacer:
6. | host: Reservar la memoria para almacenar $\mathcal{I}$, Q y R necesaria
7. | kernel: Calcular los índices $\mathcal{I}$
8. | kernel: Almacenar las matrices $\hat{A} = A(\mathcal{I}, \mathcal{J})$ en el espacio reservado para Q
9. | kernel: Resolver los problemas de mínimos cuadrados para obtener $\hat{m}_{x_{i-1}}, \ldots, \hat{m}_{x_i} - 1$
10. FinPara

Algoritmo 3.2: Pasos del Algoritmo SPAI Propuesto

Una de las decisiones más importantes es la elección del formato de compresión que se debe utilizar para el precondicionador resultante, ya que, claramente, se desea utilizar un formato que provea un buen rendimiento durante la aplicación del mismo. Es decir, se desea utilizar un formato que brinde un buen rendimiento de producto matriz vector disperso y, de acuerdo a los resultados mostrados en [2], el formato que en promedio ofrece mejor rendimiento en la GPU es el formato híbrido. Es por ello que, se ha decidido que el resultado del precondicionador sea almacenado en dicho formato.

Sin embargo, durante la construcción del precondicionador, es más importante utilizar un formato que permita un buen acceso a la información requerida por el algoritmo SPAI. Particularmente, es necesario acceder a la matriz M por columnas. El formato comprimido por columnas probablemente ofrece el acceso más simple a las columnas de una matriz, lamentablemente, dicho formato no está implementado en la biblioteca Cusp. Sin embargo, si se representa M^T en formato comprimido por filas, el cual si se encuentra implementado, se obtienen los mismos beneficios a un costo relativamente bajo. Por lo tanto, durante la construcción del precondicionador se trabaja con su traspuesta, con la salvedad de que al terminar todas las columnas (filas de M^T) el resultado se transpone y se convierte en formato híbrido para su aplicación.

En el paso 8 del Algoritmo 3.2 es necesario construir la matriz $\hat{A} = A(\mathcal{I}, \mathcal{J})$, para ello se necesita acceder eficientemente a los elementos de A, ya sea por fila o por columna. Tomando en cuenta que la biblioteca Cusp provee el formato comprimido por filas, se utiliza este para representar a la matriz A durante la construcción del precondicionador SPAI. No obstante, la construcción del precondicionador puede ser invocada con una matriz en cualquier formato existente en la biblioteca Cusp, cuando ésta no se encuentre en el formato deseado, se realiza una copia en formato comprimido por filas antes de empezar la construcción. Por lo tanto, el usuario del precondicionador no necesita esta restringido a un formato particular.

Dado que, durante la construcción del precondicionador se tiene la matriz M^T en formato comprimido por filas, todos los conjuntos $\mathcal{J}$ se encuentran explícitamente representados por el arreglo de indices de columnas. Más aún, el arreglo de desplazamiento de filas provee la información necesaria para calcular el tamaño de cada uno de esto conjuntos y una forma directa de acceder al mismo.

La biblioteca Thrust provee un procedimiento, llamado *for_each*, cuya firma se muestra en la Figura 3.2. Los iteradores *first* y *last* representan el rango semi abierto [*first*,*last*) al cual se le aplica la función *f*.

En teoría, el procedimiento se podría implementar como se muestra en el Algoritmo 3.3, sin embargo, la implementación de la biblioteca Thrust no garantiza ningún orden particular en el que la función f es invocada. En particular, cuando los iteradores

```
template<typename InputIterator,
         typename UnaryFunction>
void for_each(InputIterator first,
              InputIterator last,
              UnaryFunction f);
```

Figura 3.2: Firma del Procedimiento *for_each* de Thrust

representan un rango en memoria de dispositivo, cada invocación de f es realizada por un hilo de dispositivo diferente. Por lo tanto, se debe asumir que todas las invocaciones podrían ser ejecutadas en paralelo.

1. *current* := *first*
2. Mientras *current* $\neq$ *last* Hacer:
3. | f(*current*.elemento)
4. | Avanzar *current*
5. FinMientras

Algoritmo 3.3: Implementación Teórica del Procedimiento *for_each*

Es importante mencionar que el parámetro f no tiene que ser una función tradicional de $C/C++$, es común utilizar objetos cuya clase sobrecarga el operador (). De esta forma, la función puede operar sobre datos provistos durante la creación del objeto sin que el método *for_each* necesite ser modificado.

Adicionalmente, la biblioteca ofrece un método similar cuya firma se puede ver en la Figura 3.3. La diferencia radica en que en este caso, en lugar de una función f se tiene una operación *op* que genera un resultado, el cual es almacenado en el rango que comienza en el iterador *result* y que tiene el mismo tamaño que el rango de entrada.

```
template<typename InputIterator,
         typename OutputIterator,
         typename UnaryFunction>
  OutputIterator transform(InputIterator first, InputIterator last,
                           OutputIterator result,
                           UnaryFunction op);
```

Figura 3.3: Firma del Procedimiento *transform* de Thrust

El Algoritmo 3.4 muestra una posible implementación secuencial del procedimiento *transform*. Sin embargo, la implementación interna de dicha función resulta en la invocación del método *for_each*, con una función que se encarga de asignar el resultado de la operación. Por lo tanto, *transform* posee las mismas ventajas y desventajas que el método *for_each*.

```
1. current := first
2. Mientras current ≠ last Hacer:
3. |   result.elemento := op(current.elemento)
4. |   Avanzar current
5. |   Avanzar result
6. FinMientras
```

Algoritmo 3.4: Implementación Teórica del Procedimiento *transform*

Finalmente, el iterador *counting_iterator*, representa una posición en una secuencia de valores. Dicho iterador es útil para crear una secuencia sin almacenarla explícitamente en memoria.De esta forma, se pueden invocar los métodos *for_each* y *transform* sobre una secuencia de números.

Para calcular los tamaños de los conjuntos $\mathcal{I}$ se utilizó el método *transform* sobre el rango $[0, N)$. Por lo tanto, se crean N hilos de CUDA que reciben como parámetro la columna que les corresponde. Luego, cada uno de ellos calcula el tamaño del conjunto $\mathcal{I}$ correspondiente a su columna. Para ello, se utilizó un objeto que tiene la información estructural de las matrices A y M^T en formato comprimido por filas, es decir, los arreglos de indices de columnas y los desplazamientos de filas únicamente. El Algoritmo 3.5 muestra a grandes rasgos el proceso para el hilo correspondiente a la columna k.

Una vez obtenidos los tamaños de los conjuntos $\mathcal{I}$ y $\mathcal{J}$, estos se copian a memoria de anfitrión. En éste se calculan los requerimientos de memoria necesarios para construir el precondicionador, es decir, el espacio requerido para almacenar todos los conjuntos $\mathcal{I}$ y todas las matrices Q y R.

1. Sea $|\mathcal{I}_k|$ el tamaño del conjunto $\mathcal{I}$, inicializado en 0
2. Sea $\mathcal{J}_{c(k)}$ un iterador al comienzo de los índices de columnas para la k-ésima fila de M^T
3. Sea $\mathcal{J}_{f(k)}$ un iterador al final de los índices de columnas para la k-ésima fila de M^T
4. Para $i = 1, 2, \ldots, N$ Hacer:
5. | Sea $\mathcal{I}_{c(i)}$ un iterador al comienzo de los índices de columnas para la i-ésima fila de A
6. | Sea $\mathcal{I}_{f(i)}$ un iterador al final de los índices de columnas para la i-ésima fila de A
7. | *es_zero* := verdadero
8. | $\mathcal{J}_j := \mathcal{J}_{c(k)}$
9. | $\mathcal{I}_j := \mathcal{I}_{c(i)}$
10. | Mientras *es_zero* y $\mathcal{J}_j \neq \mathcal{J}_{f(k)}$ y $\mathcal{I}_j \neq \mathcal{I}_{f(k)}$ Hacer:
11. | | Si $\mathcal{J}_j$.elemento $= \mathcal{I}_j$.elemento Entonces:
12. | | | *es_zero* = falso
13. | | Sino, Si $\mathcal{J}_j$.elemento $< \mathcal{I}_j$.elemento Entonces:
14. | | | Avanzar $\mathcal{J}_j$
15. | | Sino
16. | | | Avanzar $\mathcal{I}_j$
17. | | FinSi
18. | FinMientras
19. | Si no es_zero Entonces:
20. | | Incrementar $|\mathcal{I}_k|$
21. | FinSi
22. FinPara
23. Retornar $|\mathcal{I}_k|$

Algoritmo 3.5: Cálculo de los tamaños de los conjuntos $\mathcal{I}$

Sin embargo, es posible que estos requerimientos excedan la memoria disponible en el dispositivo. El Algoritmo 3.6 muestra la estrategia propuesta, la cual reduce el rango de columnas restantes a la mitad iterativamente, hasta que se obtiene un rango cuyos requerimientos de memoria se puedan satisfacer. Es decir, si se han calculado las columnas $1, 2, \ldots, c-1$; entonces se verifican los requerimientos de memoria del rango de columnas restantes $[c, N+1)$. En el caso que no se puedan satisfacer dicho requerimientos, se prueba con el rango $[c, \frac{c+N+1}{2})$, luego con el rango $[c, \frac{c+N+1}{4})$, y así sucesivamente. Cuando se consigue un rango $[c, f)$ cuyos requerimientos de memoria no exceden la disponible, se reserva la memoria requerida para ello y se invocan los kernels necesarios para construir las columnas de M. Finalmente, se repite el proceso con el rango de columnas restantes $[f, N+1)$ hasta que dicho rango sea vacío.

1. Sea D la memoria disponible en el dispositivo
2. $c := 1$
3. $f := N + 1$
4. Mientras $c < f$ Hacer:
5. | Sea R la cantidad de memoria requerida para el rango $[c, f)$
6. | Si $R > D$ Entonces:
7. | | $f := \frac{c+f}{2}$
8. | Sino
9. | | Reservar la memoria necesaria para los conjuntos $\mathcal{I}$ y las matrices Q y R del rango $[c, f)$
10. | | Calcular las columnas $m_c, m_{c+1}, \ldots, m_{f-1}$ en paralelo
11. | | Liberar la memoria reservada en el paso 9
12. | | $c := f$
13. | | $f := N$
14. | FinSi
15. FinMientras

Algoritmo 3.6: Búsqueda de un Rango de Columnas que se Pueda Calcular

Una vez que se tiene un rango de columnas que pueden ser calculadas en paralelo sin exceder la memoria disponible, se utilizó la función *for_each* sobre el rango $[c, f)$ para generar los conjuntos $\mathcal{I}$. De esta forma, se crean $f - c$ hilos de dispositivo y cada uno recibe como parámetro la columna que ha de calcular, el proceso para calcular los índices es similar al utilizado en el Algoritmo 3.5 para calcular sus tamaños. Una de las diferencias es que ahora es necesario obtener un iterador a la memoria reservada para almacenar el conjunto $\mathcal{I}$ de la k-ésima columna, por lo que el objeto que implementa la función, debe ser provisto de los tamaños de lo conjuntos $\mathcal{I}$ y cuantos de ellos han sido calculados previamente. Finalmente, el paso 20 del Algoritmo 3.5, en lugar de incrementar el tamaño del conjunto debe agregar la fila i en la posición correspondiente y avanzar el iteredador.

Generados los conjuntos $\mathcal{I}$ y $\mathcal{J}$, se deben construir todas las matrices $\hat{A}$. Para ello, nuevamente se utilizó la función *for_each* sobre el rango $[c, f)$. La primera observación es que las matrices $\hat{A}$ son almacenadas en el espacio de memoria reservados para las matrices Q. Esto se debe a que el proceso de factorización QR se puede llevar a cabo sobre el mismo espacio de memoria, es decir, al final del algoritmo dicho espacio contiene la matriz Q. El Algoritmo 3.7 muestra el proceso realizado por el kernel para

generar la k-ésima matriz $\hat{A}$.

1. Sea $\mathcal{I}_{c(k)}$ un iterador al comienzo de los índices de columnas para la k-ésima fila de A
2. Sea $\mathcal{I}_{f(k)}$ un iterador al final de los índices de columnas para la k-ésima fila de A
3. Sea $\mathcal{J}_{c(k)}$ un iterador al comienzo de los índices de columnas para la k-ésima fila de M^T
4. Sea $\mathcal{J}_{f(k)}$ un iterador al final de los índices de columnas para la k-ésima fila de M^T
5. $\mathcal{I}_k := \mathcal{I}_{c(k)}$
6. Mientras $\mathcal{I}_k \neq \mathcal{I}_{f(k)}$ Hacer:
7. | Sea $\mathcal{I}_{c(\mathcal{I}_k)}$ un iterador al comienzo de los índices de columnas para la fila representada por $\mathcal{I}_k$
8. | Sea $\mathcal{I}_{f(\mathcal{I}_k)}$ un iterador al final de los índices de columnas para la fila representada por $\mathcal{I}_k$
9. | $\mathcal{I}_i := \mathcal{I}_{c(\mathcal{I}_k)}$
10. | $\mathcal{J}_i := \mathcal{J}_{c(k)}$
11. | Mientras $\mathcal{I}_i \neq \mathcal{I}_{f(\mathcal{I}_k)}$ y $\mathcal{J}_i \neq \mathcal{J}_{f(k)}$ Hacer:
12. | | Si $\mathcal{I}_i$.elemento $= \mathcal{J}_i$.elemento Entoces:
13. | | | $Q(\mathcal{I}_i, \mathcal{J}_i) := A$.valores$[\mathcal{I}_i]$
14. | | | Avanzar $\mathcal{I}_i$
15. | | | Avanzar $\mathcal{J}_i$
16. | | Sino, Si $\mathcal{I}_i$.elemento $< \mathcal{J}_i$.elemento Entoces:
17. | | | Avanzar $\mathcal{I}_i$
18. | | Sino
19. | | | Avanzar $\mathcal{J}_i$
20. | | FinSi
21. | FinMientras
22. | Avanzar $\mathcal{I}_k$
23. FinMientras

Algoritmo 3.7: Construcción de las matrices $\hat{A}$

Finalmente, construidas las matrices $\hat{A}$ es necesario resolver el problema de minimización $\min_{\hat{m}_j} \left\| \hat{e}_j - \hat{A}_j \hat{m}_j \right\|$ utilizando factorización QR, para lo cual se podría utilizar la función *for_each* y crear un hilo para cada columna. Sin embargo, la factorización QR se compone únicamente de actualizaciones de vectores, productos internos, escalamiento de vectores y cálculos de normas; todas son operaciones que pueden, al menos hasta cierto punto, ser ejecutadas en paralelo. Más aún, el proceso de sustitución hacia atrás puede ser parcialmente paralelizado, de forma que, se puede desarrollar un kernel más eficiente utilizando más de un hilo CUDA por columna a calcular. La implementación desarrollada utiliza un warp (32 hilos) por columna, el uso de un warp es una elección común y bien fundamentada al programar en CUDA, ya que es el

máximo número de hilos que se puede utilizar sin necesidad de introducir barreras de sincronización.

1. Sean $Q_k = q_{ij}$ y $R_k = r_{ij}$ las matrices asociadas a la k-ésima columna de M
2. Sean M_k y N_k las dimensiones de la matriz Q_k
3. Para $j = 1, 2, \ldots N_k$ Hacer:
4. | Para $i = 1, 2, \ldots j$ Hacer:
5. | | compartida[p] := 0
6. | | Para $p^* = p, 2p, \ldots, \lfloor \frac{M_k}{32} \rfloor p$ Hacer:
7. | | | compartida[p] := compartida[p] + $q_{p^*,j} q_{p^*,i}$
8. | | FinPara
9. | | Reducir compartida[$0 - 32$] a compartida[0]
10. | | Si pista $= 1$
11. | | | r_{ij} := compartida[0]
12. | | FinSi
13. | | Para $p^* = p, 2p, \ldots, \lfloor \frac{M_k}{32} \rfloor p$ Hacer:
14. | | | $q_{p^*,j} := q_{p^*,j} - r_{i,j} q_{p^*,i)}$
15. | | FinPara
16. | FinPara
17. | compartida[p] := 0
18. | Para $p^* = p, 2p, \ldots, \lfloor \frac{M_k}{32} \rfloor p$ Hacer:
19. | | compartida[p] := compartida[p] + $q^2_{p^*,j}$
20. | FinPara
21. | Reducir compartida[$0 - 32$] a compartida[0]
22. | Si pista $= 1$
23. | | $r_{jj} := \sqrt{\text{compartida[0]}}$
24. | FinSi
25. | Para $p^* = p, 2p, \ldots, \lfloor \frac{M_k}{32} \rfloor p$ Hacer:
26. | | $q_{p^*,j} := \frac{q_{p^*,j}}{r_{jj}}$
27. | FinPara
28. FinPara
29. Para $i = N_k, N_k - 1, \ldots 1$ Hacer:
30. | $m_{ki} := \frac{m_{ki}}{r_{ii}}$
31. | Para $p^* = p, 2p, \ldots, \lfloor \frac{M_k}{32} \rfloor p$ Hacer:
32. | | $m_{k,p^*} := m_{k,p^*} - m_{ki} r_{p^*,j}$
33. | FinPara
34. FinPara

Algoritmo 3.8: Kernel para calcular las columnas de M

La función *for_each* facilita la aplicación de una función a un rango, abstrayendo al usuario de la interacción directa con la invocación de un kernel, sin embargo, como

es usual, dicha abstracción conlleva a una reducción en control. En particular, no se puede garantizar que la cantidad de hilos por bloque sea múltiplo de 32, por otro lado, si se desea utilizar hilos cooperantes es necesario que éstos pertenezcan al mismo bloque. Por lo tanto, el kernel que resuelve el problemas de minimización es invocado directamente utilizando bloques de exactamente 32 hilos.

El producto interno y el cálculo de normas requiere una reducción paralela, para mayor eficiencia es necesario utilizar la memoria compartida para realizar dicha reducción. Por consiguiente, el kernel utiliza un arreglo de 32 valores de punto flotante de doble precisión en memoria compartida, cada hilo del warp acumula información en una de esas posiciones y luego el valor es reducido en paralelo sobre la misma memoria. Finalmente, el primer hilo del warp copia el resultado de la reducción en la posición de memoria global correspondiente. El Algoritmo 3.8 muestra de manera simplificada el kernel utilizado, en este caso los hilos no se identifican únicamente por la columna k que les corresponde, sino que, también poseen un valor entero p en el rango $[0, 31]$ que indica la pista dentro del warp a la que pertenece.

```
1. Para d = 16, 8, 4, 2, 1
2. |   Si p < d
3. |   |   compartida[p]:= compartida[p] + compartida[p + d]
4. |   FinSi
5. FinPara
```

Algoritmo 3.9: Reducción paralela en memoria compartida

El bucle 3-20 del Algoritmo 3.8 realiza la factorización QR de la matriz $\hat{A}$ que se encuentra en el espacio de memoria correspondiente a Q, el paralelismo se consigue en los pasos 6, 13 y 18 que calculan un producto interno, una actualización de vector y una norma, respectivamente. Gracias a la arquitectuta SIMT de CUDA los 32 hilos del warp realizan cada una de esas operaciones en simultáneo, reduciendo así el tiempo de ejecución total. Sin embargo, los pasos 9 y 21 requieren una reducción paralela similar a la mostrada el Algoritmo 3.9, cuya naturaleza obliga a que algunos hilos se mantengas ociosos, afortunadamente reducir en paralelo 32 valores requiere únicamente 5

instrucciones. Finalmente, el bucle 29-34 del Algoritmo 3.8 realiza la sustitución hacia atrás, la cual también incluye un bucle paralelizado para la actualización del vector del lado derecho. Vale mencionar que, durante la factorización QR se almacena en m_j la fila de Q que resultaría del producto $Q^T\hat{e}_j$, este paso no es mostrado en el pseudocódigo ya que introduce una complejidad innecesaria que no ayudaría a entender el paralelismo del kernel.

Capítulo 4

Pruebas

En este capítulo se muestran los resultados de los experimentos realizados para evaluar el rendimiento de los algoritmos propuestos. En la primera sección se muestran los experimentos relacionados a resolver sistemas lineales, dichos experimentos tienen como finalidad mostrar que es posible realizar cómputo de alto rendimiento en la GPU sin perder precisión con respecto a la CPU. En la siguiente sección se analizan las pruebas realizadas con el precondicionador SPAI.

Las pruebas se realizaron utilizando el SDK de CUDA versión 4.2, la cual incluye la versión 1.4.0 de la biblioteca Thrust; además, se utilizó la versión 0.3.0 de la biblioteca Cusp. Se realizaron pruebas en diferentes GPUs y CPUs, los resultados que se muestran en éste capítulo se refieren al mejor tiempo obtenido en cada caso. Específicamente, los resultados de la GPU se refieren a una tarjeta Tesla C2070 que posee 448 núcleos de CUDA y 6GB de memoria GDDR5. Mientras que los resultados de la CPU fueron generados con un Intel Core i5-2450M de segunda generación, el cual utiliza un reloj de 2,5GHz en una computadora con 8GB de memoria DDR3.

Todos los resultados que se muestran en las siguientes secciones utilizan números punto flotante de doble precisión. Por un lado, esto presenta un inconveniente en el acceso a memoria que podría perjudicar el rendimiento, sin embargo, es un requerimiento obligatorio si se desea obtener alguna precisión razonable. Más aún, las pocas pruebas realizadas en precisión simple reflejaron que los resultados son poco fiables y que el uso de doble precisión es tan solo aproximadamente 1,2 veces más lento.

Para las pruebas se utilizó una tolerancia relativa de $1e-6$ sobre la norma del residual, más aún, siempre se especificó un máximo de iteraciones suficientemente alto para alcanzar la convergencia. Además, el parámetro de reinicio para GMRES se ajustó caso por caso hasta lograr alcanzar la convergencia en la CPU. Una vez obtenidos los parámetros adecuados, se realizaron pruebas idénticas en ambas arquitecturas sin considerar los experimentos que llevaron a conseguir dichos parámetros.

4.1. Resolviendo Sistemas Lineales en la GPU

Los sistemas $Ax = b$ con A simétrica positiva definida ocurren con frecuencia en aplicaciones prácticas. Por lo tanto, se decidió realizar pruebas con el método del gradiente conjugado (CG) que ofrece la biblioteca Cusp. La intención de dichas pruebas es observar la posibilidad de resolver sistemas grandes en la GPU, además, experimentos con métodos ya implementados eran necesarios para evaluar la factibilidad de la propuesta del presente Trabajo Especial de Grado. Más aún, ésto hace posible comparar, en cierta medida, el comportamiento de los métodos implementados en éste trabajo con aquellos ya existentes en la biblioteca.

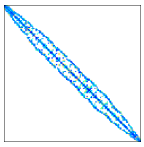
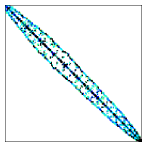
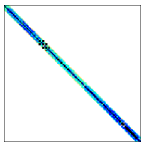
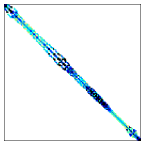
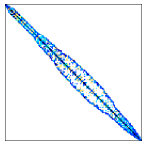
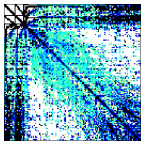
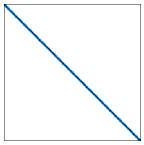
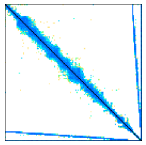
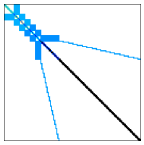
 **bcsstk14** *N*: 1.806 *NNZ*: 63.454	 **bcsstk15** *N*: 3.948 *NNZ*: 117.816	 **bcsstk16** *N*: 4.884 *NNZ*: 290.378
 **bcsstk17** *N*: 10.947 *NNZ*: 428.650	 **bcsstk18** *N*: 11.948 *NNZ*: 149.090	 **hood** *N*: 220.542 *NNZ*: 9.895.422
 **ecology2** *N*: 999.999 *NNZ*: 4.995.991	 **thermal2** *N*: 1.228.045 *NNZ*: 8.580.313	 **G3_circuit** *N*: 1.585.478 *NNZ*: 7.660.826

Tabla 4.1: Matrices simétricas definidas positivas

La Tabla 4.1 muestra las matrices utilizadas con el método del gradiente conjugado, los sistemas pequeños se probaron con los precondicionadores diagonal y AINV de

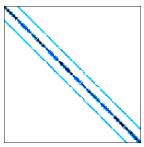
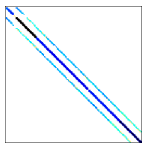
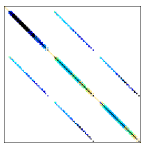
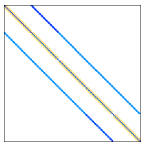
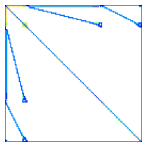
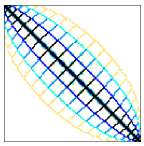
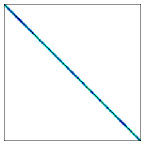
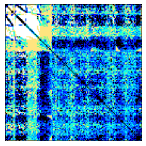
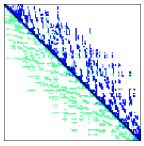
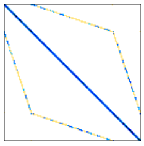
 **sherman1** *N*: 1.000 *NNZ*: 3.750	 **sherman3** *N*: 5.005 *NNZ*: 20.033	 **sherman4** *N*: 1.104 *NNZ*: 3.786
 **orsreg_1** *N*: 2.205 *NNZ*: 14.133	 **memplus** *N*: 17.758 *NNZ*: 99.147	 **SiH4** *N*: 5.041 *NNZ*: 171.903
 **BenElechi1** *N*: 245.874 *NNZ*: 16.150.496	 **F1** *N*: 343.741 *NNZ*: 26.837.113	 **cage14** *N*: 1.505.785 *NNZ*: 27.130.349
 **FEM_3D_thermal2** *N*: 147.900 *NNZ*: 3.489.300		

Tabla 4.2: Matrices generales

la biblioteca Cusp. Sin embargo, para las matrices grandes la construcción del precondicionador AINV consumía grandes recursos y ofrecía poca información. Por lo tanto, únicamente se utilizó el precondicionador diagonal en las mismas.

Por otro lado, para sistemas generales se probaron los métodos GMRES y TFQMR implementados en éste trabajo, así como el método BICGSTAB de la biblioteca Cusp. En éste caso, los sistemas pequeños se probaron sin precondicionar y con los precondicionadores AINV y diagonal; mientras que los sistemas grandes únicamente con el precondicionador diagonal. Más aún, la mayoría de estos se probaron solamente con el método GMRES. En la Tabla 4.2 se listan las matrices utilizadas para dichas pruebas.

Las Tablas 4.3, 4.4, 4.5 y 4.6 muestran los resultados de las pruebas realizadas utilizando los métodos CG, GMRES, BICGSTAB y TFQMR, respectivamente. En ellas se muestran para cada matriz el precondicionador utilizado, los tiempos e iteraciones tanto para CPU como GPU; y finalmente, se muestran las tasas de tiempo de CPU con respecto al tiempo de GPU. Es importante mencionar que los tiempos mostrados se refieren únicamente a la resolución del problema, es decir, no se toma en cuenta transferencias de memoria ni construcción del precondicionador.

Matriz	**Precondicionador**	**Tiempo**		**N° de Iteraciones**		**Tasa**
		CPU	**GPU**	**CPU**	**GPU**	
bcsstk14	Diagonal	37,68 ms	664,19 ms	410	409	0,057
	AINV	24,95 ms	251,44 ms	106	106	0,099
bcsstk15	Diagonal	100,82 ms	1063,86 ms	576	574	0,095
	AINV	132,76 ms	1180,78 ms	362	360	0,112
bcsstk16	Diagonal	67,29 ms	345,17 ms	172	172	0,194
	AINV	49,36 ms	266,58 ms	100	100	0,185
bcsstk17	Diagonal	1,83 s	4,17 s	2840	2841	0,439
	AINV	4,24 s	6,83 s	2409	2408	0,621
bcsstk18	Diagonal	0,55 s	2,56 s	1722	1705	0,216
	AINV	0,72 s	1,81 s	656	638	0,398
ecology2	Diagonal	99,88 s	16,76 s	5567	5567	5,960
G3_circuit	Diagonal	88,17 s	16,09 s	2727	2727	5,481
thermal2	Diagonal	126,55 s	27,63 s	3462	3460	4,581
hood	Diagonal	81,22 s	21,44 s	4799	4769	3,789

Tabla 4.3: Resultados de las pruebas realizadas con CG

Se puede observar que, en general, la tasa es mejor con el precondicionador AINV que con el precondicionador diagonal. Dicho resultado se puede explicar debido a los requerimientos de aplicación de cada uno, ya que aplicar AINV cuesta dos productos matriz vector disperso más que la aplicación de diagonal. Tomando en consideración los resultados mostrados en [3], no sorprende que dichos productos matriz vector se comporten mejor en la GPU.

Matriz	**Precondicionador**	**Tiempo**		**N° de Iteraciones**		**Tasa**
		CPU	**GPU**	**CPU**	**GPU**	
sherman1	Identidad	98,92 ms	15021,30 ms	1232	1233	0,007
	AINV	3,98 ms	3179,37 ms	34	34	0,001
sherman3	AINV	72,50 ms	3061,87 ms	189	189	0,024
sherman4	Identidad	31,89 ms	3221,18 ms	369	369	0,010
	AINV	6,25 ms	429,65 ms	47	47	0,015
orsreg_1	Identidad	35,75 ms	1995,01 ms	225	225	0,018
	Diagonal	38,97 ms	2136,75 ms	240	240	0,018
	AINV	10,29 ms	507,46 ms	48	48	0,020
memplus	Identidad	1,12 s	84,07 s	888	888	0,133
	Diagonal	113,72 ms	862,93 ms	91	91	0,132
SiH4	Identidad	0,32 s	3,95 s	865	865	0,081
FEM_3D_thermal2	Identidad	6,16 s	3,27 s	672	672	1,883
BenElechi1	Diagonal	4,69 min	2,28 min	4206	4228	2,058
F1	Diagonal	8,10 min	2,64 min	5052	4985	3,073
cage14	Diagonal	845,41 ms	185,67 ms	10	10	4,553

Tabla 4.4: Resultados de las pruebas realizadas con GMRES

Finalmente, queda claro que para matrices relativamente pequeñas la CPU tie-

ne un comportamiento mucho mejor, mientras que, la GPU mejora su rendimiento a medida que crece el tamaño del sistema. Para buscar explicación a dicho comportamiento, se analizan las curvas de crecimiento de los tiempos de ejecución con respecto al tamaño del vector para las operaciones de cálculo de norma, producto escalar, actualización y escalamiento. Además, se analiza el creciemiento para el producto matriz vector disperso con respecto a la cantidad de entradas no cero de la matriz.

Matriz	**Precondicionador**	**Tiempo**		**N° de Iteraciones**		**Tasa**
		CPU	**GPU**	**CPU**	**GPU**	
sherman1	Identidad	12,18 ms	837,53 ms	295	311	0,015
	AINV	3,27 ms	123,59 ms	24	24	0,026
sherman3	AINV	19,58 ms	432,36 ms	77	76	0,045
sherman4	Identidad	4,30 ms	235,82 ms	80	80	0,018
	AINV	3,29 ms	161,66 ms	31	31	0,020
orsreg_1	Diagonal	16,73 ms	639,67 ms	218	193	0,026
	AINV	4,85 ms	162,37 ms	27	27	0,030
memplus	Identidad	338,33 ms	1931,71 ms	558	559	0,175
	Diagonal	113,28 ms	957,62 ms	179	282	0,118
FEM_3D_thermal2	Diagonal	206,02 ms	95,17 ms	17	17	2,165
G3_circuit	Diagonal	12,85 s	2,47 s	205	184	5,206
hood	Diagonal	30,96 s	9,90 s	918	900	3,126

Tabla 4.5: Resultados de las pruebas realizadas con BICGSTAB

La Figura 4.1 muestra las curvas de crecimiento para las operaciones de actualización y escalamiento de vectores, tanto para CPU como para GPU. Dado que dichas operaciones son $O(n)$ el crecimiento lineal en la CPU es previsible, sin embargo, en la GPU el tiempo no parece crecer con el tamaño del vector. Esto se debe al alto grado de paralelismo que se puede lograr para éstas operaciones, ya que teóricamente es posible calcular todas las componentes al mismo tiempo en paralelo, por lo tanto, el tamaño del vector no afecta significativamente el tiempo de ejecución.

Las operaciones de cálculo de normas y productos escalares son, al igual que las actualizaciones de vectores, $O(n)$. Por lo tanto, se espera un comportamiento similar en la CPU al obtenido anteriormente. Dicho comportamiento se puede corroborar en la Figura 4.2, la cual muestra las curvas de crecimiento para dichas operaciones. Sin embargo, ésta vez el crecimiento en la GPU también es lineal, aunque posee una pendiente menor que la observada para el crecimiento en la CPU. La diferencia de estas operaciones con respeto a las anteriores en la GPU se debe al proceso de re-

Matriz	Precondicionador	Tiempo		N° de Iteraciones		Tasa
		CPU	GPU	CPU	GPU	
sherman1	Identidad	24,63 ms	1386,61 ms	1035	980	0,018
	AINV	3,17 ms	133,13 ms	48	48	0,024
sherman3	AINV	28,52 ms	567,44 ms	189	190	0,050
sherman4	Identidad	6,02 ms	293,68 ms	208	206	0,020
	AINV	3,90 ms	184,93 ms	79	79	0,021
orsreg_1	Identidad	20,88 ms	571,38 ms	413	390	0,037
	Diagonal	20,34 ms	569,82 ms	390	380	0,036
	AINV	6,19 ms	320,16 ms	66	66	0,019
memplus	Diagonal	55,71 ms	316,98 ms	124	124	0,176
FEM_3D_thermal2	Diagonal	310,49 ms	134,99 ms	41	41	2,300
G3_circuit	Diagonal	23,30 s	4,96 s	530	530	4,702
hood	Diagonal	33,89 s	10,83 s	1772	1778	3,128

Tabla 4.6: Resultados de las pruebas realizadas con TFQMR

ducción paralela, ya que este proceso debe ser realizado por un único bloque de hilos cooperativos.

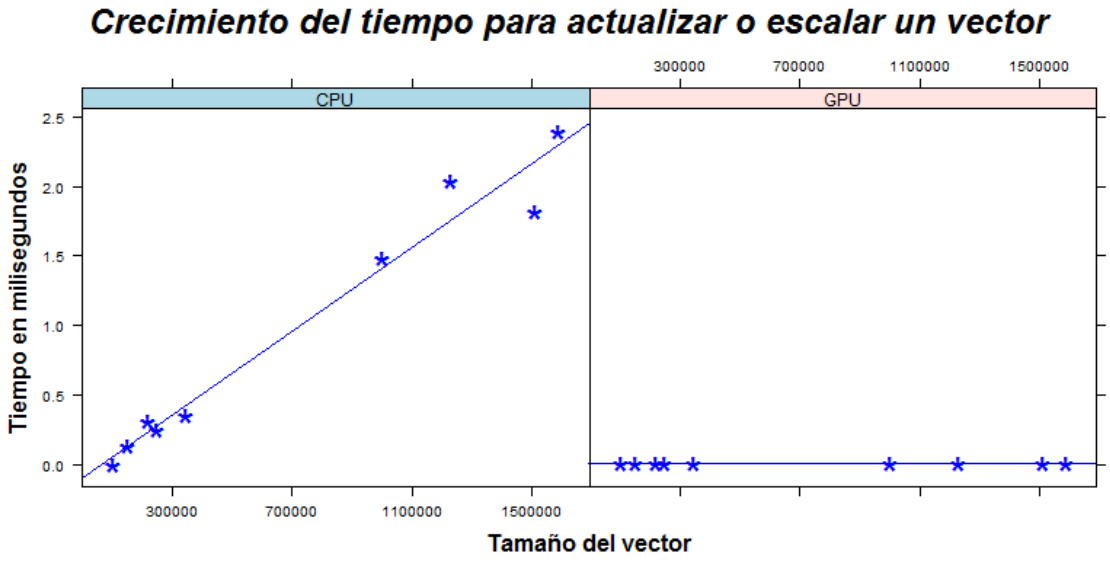


Figura 4.1: Crecimiento del tiempo de ejecución para axpy y escalamiento

El proceso de reducción se compone de dos partes, primero cada hilo acumula ciertos valores del vector en su banco de memoria compartida y luego, se realiza la reducción final sobre dicha memoria. La segunda parte es $O(\lg n)$ y opera sobre una pequeña cantidad de elementos. Sin embargo, la primera acumulación crece linealmente con respecto al tamaño del vector, ya que cada hilo debe iterar aproximadamente $\frac{N}{H}$

veces, donde, N es el tamaño del vector y H representa la cantidad de hilos del bloque. Por lo tanto, el orden de crecimiento esperado es lineal, aunque, dado que existe cierto grado de paralelismo en las operaciones, se puede explicar la reducción en la pendiente de la curva de crecimiento.

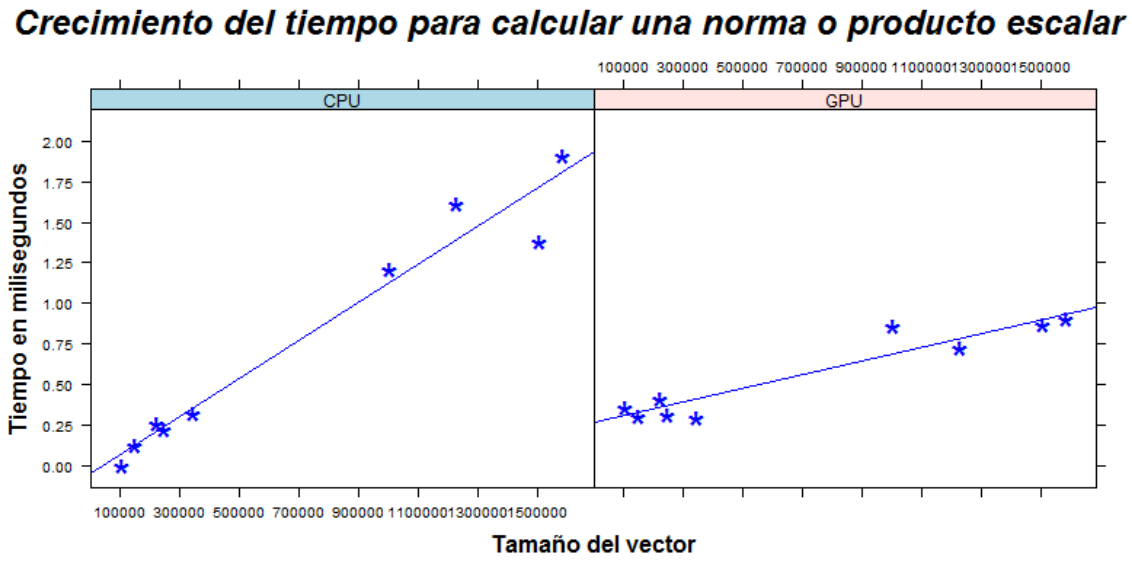


Figura 4.2: Crecimiento del tiempo de ejecución para norma y producto escalar

Finalmente, la Figura 4.3 muestra el creciemiento en tiempo de ejecución para CPU y GPU con respecto al número de entradas no ceros de la matriz para el SpMV. Se puede observar que al igual que en el caso anterior, el crecimiento es lineal en ambas arquitecturas con mayor pendiente de crecimiento para la CPU. Esto se debe a que el producto matriz vector, al igual que las reducciones, no se puede paralelizar por completo. Por una parte, existe cierta dependencia entre las operaciones involucradas, y por otra, la cantidad de hilos necesarios rápidamente sobrepasaría los límites permitidos por la arquitectura.

Los kernels propuestos en [3] para el producto matriz vector utilizan un hilo (un warp en el caso de CSR vector) por fila de la matriz. Por lo tanto, cada hilo calcula una componente del vector resultante y el tiempo de ejecución crece con el promedio de cantidad de elementos no cero por fila. Sin embargo, cada fila se procesa de forma independiente, es por ello que en este caso se observa una gran diferencia entre las

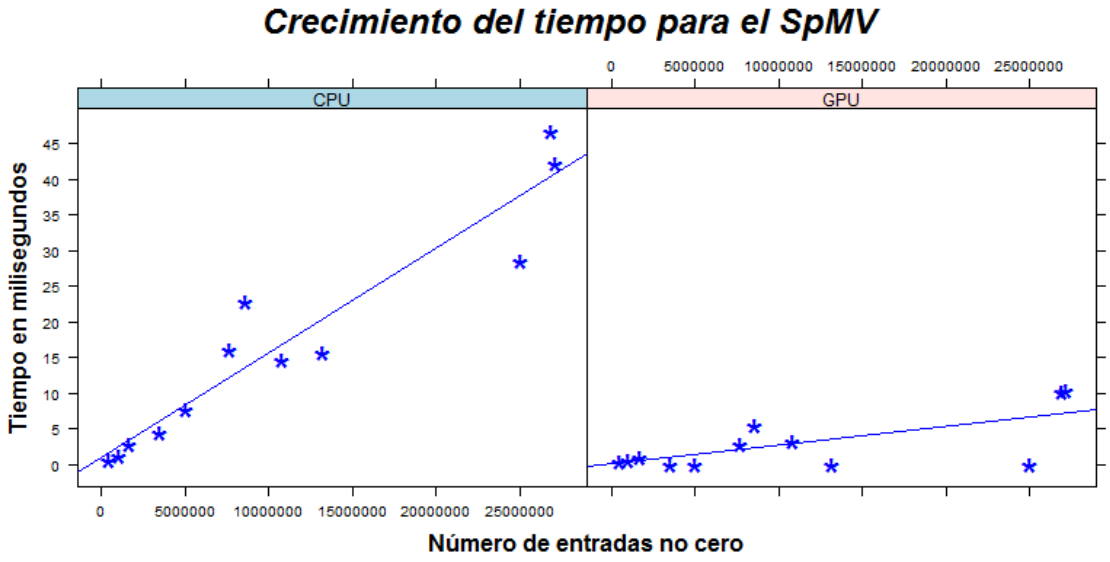


Figura 4.3: Crecimiento del tiempo de ejecución para el SpMV

pendientes de crecimiento de las diferentes arquitecturas.

4.2. Precondicionando con SPAI

Debido al costo de construcción del precondicionador SPAI, éste se utilizó únicamente en aquellos casos donde no se pudo resolver el sistema utilizando los precondicionadores facilitados por la biblioteca Cusp. Vale mencionar que para todos los sistemas mostrados en la Tabla 4.7 se logró alcanzar la convergencia utilizando el precondicionador SPAI propuesto en éste trabajo. Sin embargo, hubo casos donde el patrón de dispersión de la matriz A no ofreció resultados satisfactorios, por lo que fue necesario utilizar el patrón de A^2 para los mismos.

La Tabla 4.8 muestra el número de iteraciones que fueron necesarios para resolver el sistema y la norma del residual alcanzada en ambas arquitecturas, además, se muestra la diferencia de dichas normas. Se puede observar que el número de iteraciones es generalmente el mismo en ambas, más aún, las diferencias que existen no parecen diferir en las observadas para las pruebas de la sección anterior. Por lo tanto, se puede

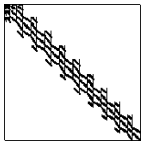
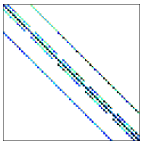
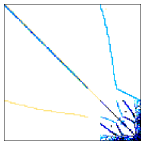
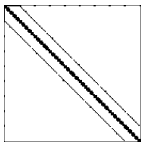
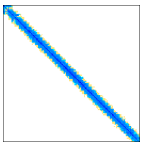
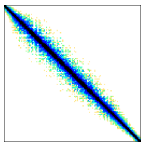
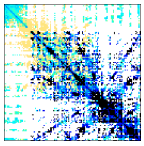
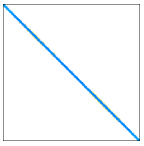
 **fidap027** *N*: 974 *NNZ*: 37.602	 **sherman2** *N*: 1.080 *NNZ*: 23.094	 **powersim** *N*: 15.838 *NNZ*: 64.424
 **rajat09** *N*: 24.482 *NNZ*: 105.573	 **chipcool1** *N*: 20.082 *NNZ*: 281.150	 **airfoil_2d** *N*: 14.214 *NNZ*: 259.688
 **2cubes_sphere** *N*: 101.492 *NNZ*: 1.647.264	 **ABACUS_shell_ud** *N*: 23.412 *NNZ*: 218.484	

Tabla 4.7: Matrices precondicionadas con SPAI

Matriz	**Patrón**	**N° de Iteraciones**		**Relres**		**Diferencia de Relres**
		CPU	**GPU**	**CPU**	**GPU**	
sherman2	A	366	366	$9{,}99e-07$	$9{,}99e-07$	$1{,}00e-12$
fidap027	A	433	433	$9{,}98e-07$	$9{,}98e-07$	0
	A^2	34	34	$8{,}09e-07$	$8{,}09e-07$	0
rajat09	A	2862	2817	$9{,}96e-07$	$9{,}92e-07$	$4{,}41e-09$
powersim	A^2	1921	1921	$9{,}94e-07$	$9{,}94e-07$	0
ABACUS_shell_ud	A	1080	1027	$3{,}99e-07$	$2{,}90e-07$	$1{,}09e-07$
chipcool1	A	78	78	$5{,}97e-07$	$5{,}28e-07$	$6{,}96e-08$
airfoil_2d	A^2	65789	64232	$9{,}99e-07$	$9{,}99e-07$	$1{,}57e-10$
2cubes_sphere	A	3	3	$8{,}049e-07$	$8{,}04e-07$	0

Tabla 4.8: Precisión de la construcción del SPAI

deducir que la construcción del SPAI en la GPU no genera mayores problemas de precisión que la construcción del mismo en la CPU.

Matriz	**Patrón**	**N**	**NNZ**	$\|\mathcal{I}\|$	$\|\mathcal{J}\|$	**Construcción**		**Tasa**
						CPU	**GPU**	
sherman2	A	1080	23094	76,80	21,38	744,29 ms	268,14 ms	2,776
fidap027	A	974	40736	161,83	38,80	4,12 s	0,98 s	4,214
	A^2		157618	529,91	161,83	31,26 s	5,75 s	5,439
rajat09	A	24482	105573	11,23	4,32	15,13 s	1,75 s	8,662
powersim	A^2	15838	147422	40,66	9,03	14,23 s	2,75 s	5,172
ABACUS_shell_ud	A	23412	218484	25,35	9,33	47,40 s	2,84 s	16,718
chipcool1	A	20082	281150	62,02	14,00	70,83 s	4,18 s	16,948
airfoil_2d	A^2	14214	832720	206,08	58,58	3,35 min	0,37 min	9,106
2cubes_sphere	A	101492	1647264	88,43	16,23	38,95 min	3,36 min	11,590

Tabla 4.9: Tiempos de construcción del precondicionador SPAI

La Tabla 4.9 muestra los tamaños de los patrones de dispersión y el promedio

de los tamaños de los conjuntos $\mathcal{I}$ y $\mathcal{J}$ junto al tiempo de construcción del precondicionador, además, se indica la tasa de tiempo de CPU con respecto al de GPU. Se puede observar, que el tiempo de construcción de GPU es siempre menor que el tiempo de CPU, variando desde un poco menos de 3 hasta casi 17 veces más rápido. En la Figura 4.4 se muestra un ajuste lineal sobre el crecimiento de tiempo de construcción, sin embargo, es fácil ver que una línea es una pobre aproximación a la nube de puntos del CPU. Por lo tanto, se puede concluir que el orden de crecimiento no es lineal con respecto al número de entradas no cero.

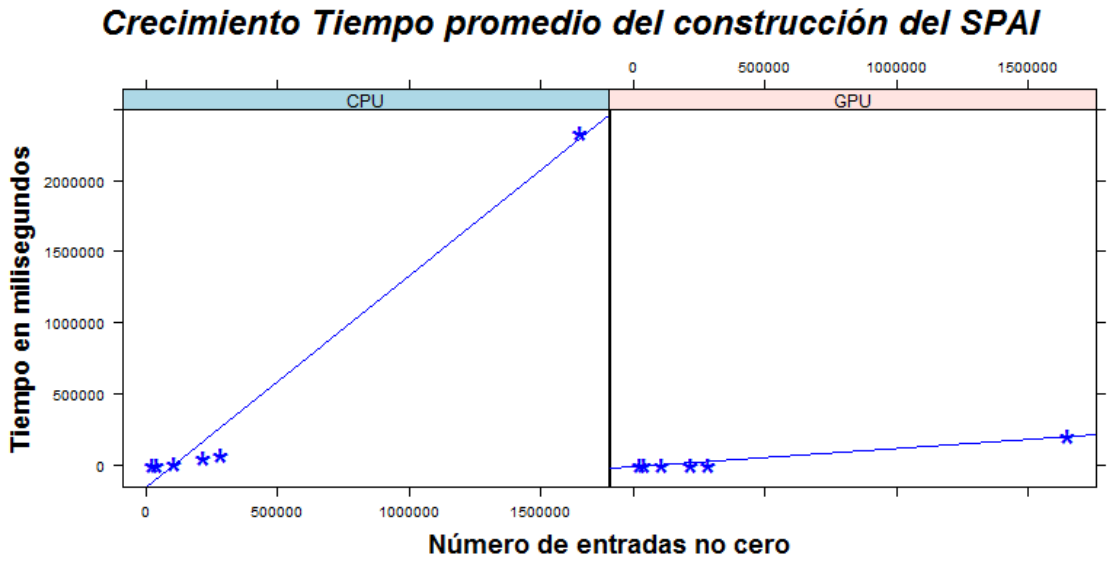


Figura 4.4: Ajuste lineal al crecimiento del tiempo de construcción del SPAI

Luego de un análisis más profundo, se puede observar un crecimiento cuadrático en el tiempo con respecto al número de entradas no cero. En la Figura 4.5 muestra que un ajuste lineal a la raíz cuadrada del tiempo de construcción con respecto al número de entradas no cero es una buena aproximación a los puntos. Dado que ajustar una línea utilizando la raíz de los tiempos es equivalente a utilizar una parábola para aproximar los puntos originales, se puede concluir que el orden de crecimiento es cuadrático.

Por otro lado, el ajuste lineal y el ajuste cuadrático sobre la nube de puntos del GPU son similares, lo que hace difícil determinar su orden de crecimiento. Por lo tanto, en la Figura 4.6 se modela el crecimiento de la tasa de tiempo de CPU con respecto

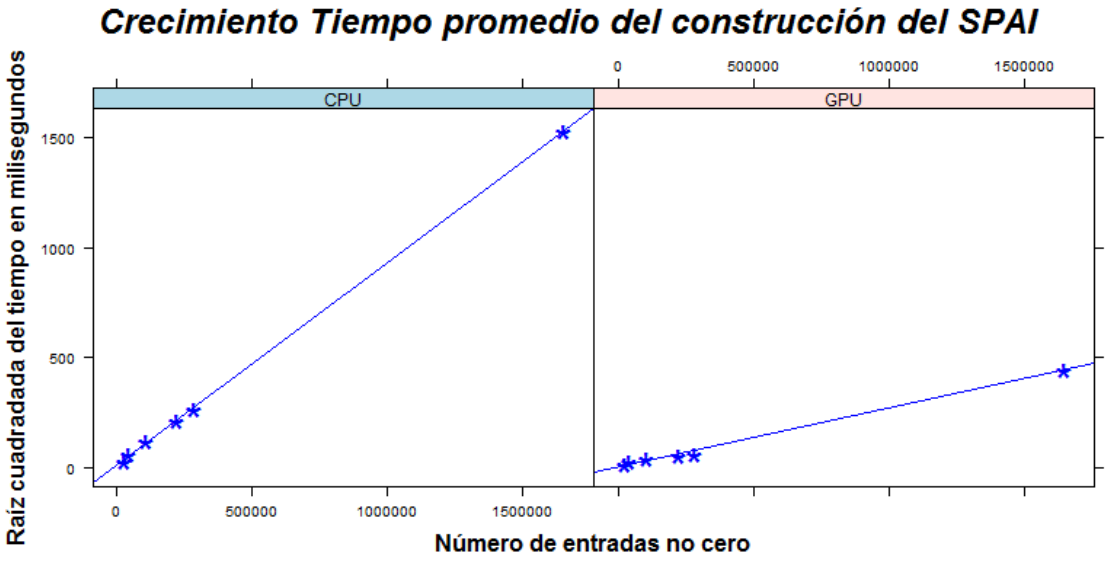


Figura 4.5: Ajuste cuadrático al crecimiento del tiempo de construcción del SPAI

a GPU a medida que aumenta el número de entradas no cero, además, se segmentan los datos según el promedio de los tamaños de los conjuntos $\mathcal{I}$. Se puede observar que mientras mayores sean dichos conjuntos, menor es la ganancia de construcción en el GPU. Lo que muestra que, mientras más densa sea la matriz menor será la ganancia obtenida por el algoritmo propuesto. Esto se debe principalmente a tres razones, la primera de ellas ocurre cuando no caben todas las matrices Q y R en memoria, ya que el algoritmo disminuye el grado de paralelismo al calcular las columnas de la matriz precondicionadora por bloques. Por otro lado, mayores tamaños implican peores patrones de acceso a memoria global durante la ejecución del kernel, ya que los datos necesarios se pueden encontrar alejados unos de otros. Finalmente, cada columna de M es calculada por un warp, donde cada hilo tiene un trabajo proporcional al tamaño del conjunto $\mathcal{I}$.

Finalmente, la Tabla 4.10 muestra los tiempos que tomó el método iterativo y el total utilizado para la resolución de los sistemas probados con SPAI, es decir, la columna "Total" toma en consideración el tiempo de construcción del precondicionador y el utilizado por el método iterativo. Dado que las matrices para las que se construyó el precondicionador SPAI son relativamente pequeñas, se observan tasas bajas para la

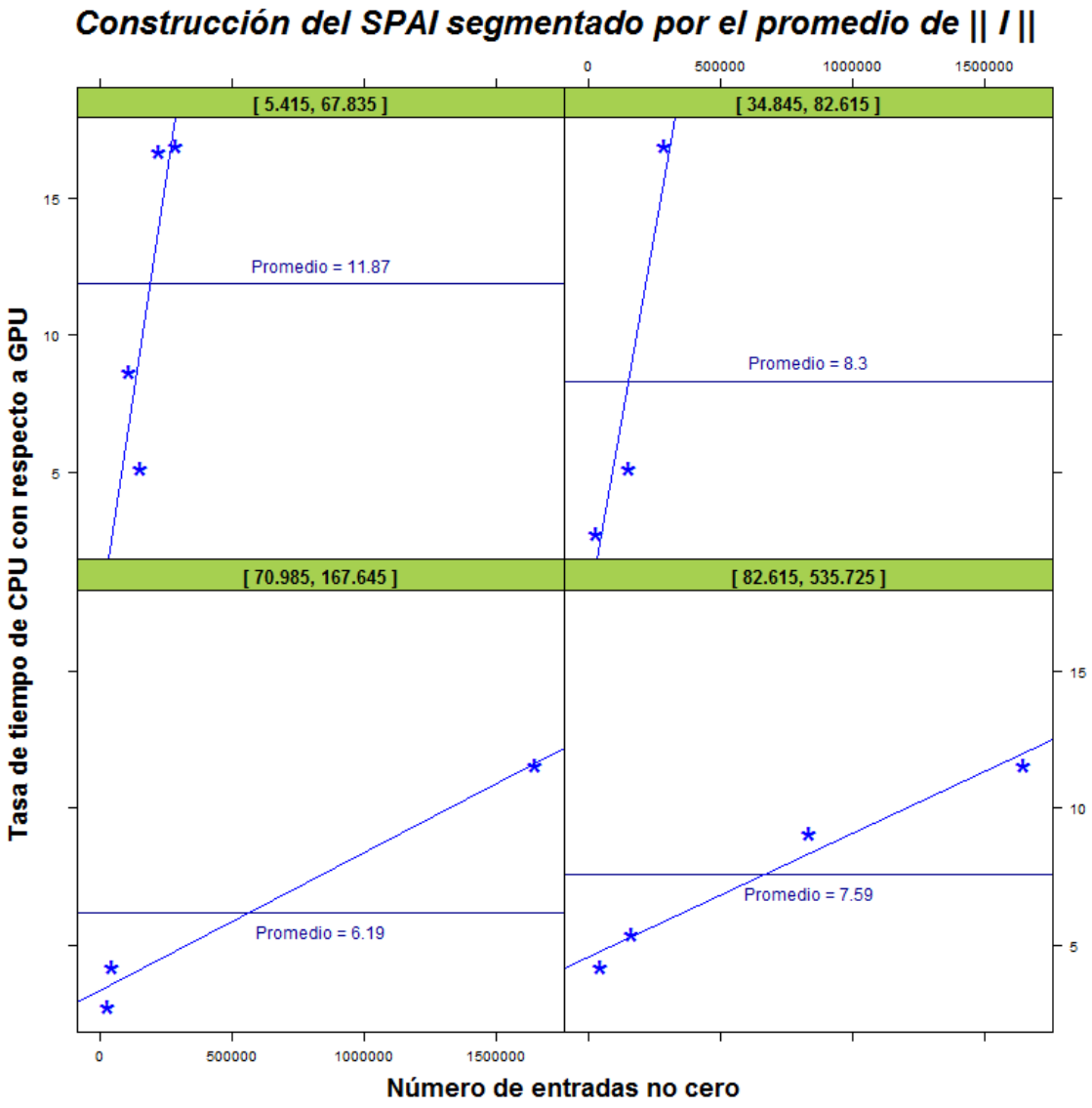


Figura 4.6: Tasa de tiempo de construcción del precondicionador SPAI segmentadada por el promedio de $\|\mathcal{I}\|$

resolución del método. Sin embargo, en un poco más de la mitad de los casos la tasa total sigue favoreciendo al GPU, lo que implica que en muchos casos la ventaja obtenida por la construcción paralela del precondicionador sobrepasa la desventaja de utilizar el GPU para sistemas pequeños. Por otra parte, podría utilizarse un enfoque híbrido para los sistemas pequeños, construyendo el precondicionador en el GPU y luego resolviendo el sistema en el CPU.

Matriz	**Patrón**	**Solución**		**Total**		**Tasa**		
		CPU	**GPU**	**CPU**	**GPU**	**Const**	**Sol**	**Total**
sherman2	A	0,06 s	4,22 s	0,81 s	4,49 s	2,776	0,015	0,180
fidap027	A	0,10 s	3,99 s	4,22 s	4,97 s	4,214	0,025	0,850
	A^2	0,02 s	0,34 s	31,28 s	6,09 s	5,439	0,062	5,137
rajat09	A	27,57 s	25,32 s	42,70 s	27,07 s	8,662	1,089	1,577
powersim	A^2	23,86 s	239,35 s	38,10 s	242,11 s	5,172	0,100	0,157
ABACUS_shell_ud	A	1,96 s	3,76 s	49,36 s	6,60 s	16,718	0,520	7,482
chipcool1	A	0,22 s	0,26 s	71,05 s	4,44 s	16,948	0,863	16,017
airfoil_2d	A^2	8,80 min	68,80 min	12,15 min	69,16 min	9,106	0,128	0,176
2cubes_sphere	A	0,06 s	0,02 s	38,95 min	3,36 min	11,590	2,785	11,589

Tabla 4.10: Tiempo de solución de sistemas utilizando SPAI

Capítulo 5

Conclusiones

En este trabajo, se realizaron implementaciones de los método GMRES y TFQMR para la GPU utilizando la biblioteca Cusp, además, se desarrolló un algoritmo para la construcción paralela del precondicionador SPAI. El uso de la biblioteca Cusp permitió, gracias a operaciones paralelas altamente optimizadas, el desarrollo de métodos iterativos eficientes sin requerir un alto grado de experiencia en la programación de la GPU. Sin embargo, la construcción paralela del precondicionador SPAI presentó una serie de retos y dificultades que requirieron de un nivel relativamente alto de conocimiento sobre la arquitectura CUDA.

Los experimentos realizados muestran que el paralelismo masivo presente en las GPUs modernas, permite la resolución de grandes sistemas lineales de forma eficiente. En las prueba realizadas se resolvieron sistemas hasta 6 veces más rápido en la GPU, además, cuando la cantidad de entradas no cero superaba los 300 mil, la ganancia promedio fue de 3,2. Sin embargo, se observó que la ganancia proviene únicamente de la ejecución altamente paralela, en particular, la ejecución serial ofrece mejores rendimientos en los sistemas pequeños donde el grado de paralelismo es relativamente bajo.

Por otro lado, se pudo observar que los precondicionadores de tipo inversas aproximadas se pueden aplicar eficientemente en este tipo de arquitecturas, esto se debe al paralelismo existente en la operación de producto matriz vector disperso. Más aún, la construcción del precondicionador SPAI es altamente paralelizable, la implementación

propuesta mostró factores de aceleración de hasta 17 veces más rápido en la GPU, con un promedio de 9 y un mínimo de 2,8 en una matriz relativamente pequeña.

En la mayoría de las pruebas realizadas el número de iteraciones coincidió en ambas arquitecturas, además, todos los sistemas se pudieron resolver en la GPU con la misma tolerancia que en la CPU. Esto indica, que las GPUs modernas son capaces de ofrecer cómputo de precisión comparable a las CPUs tradicionales, a diferencia de las generaciones anteriores cuando la precisión de la GPU se encontraba limitada a los números de punto flotante de precisión simple.

Finalmente, se puede concluir que utilizar lenguajes de alto nivel para crear programas paralelos para la arquitectura CUDA permite, a un mayor grupo de desarrolladores, explotar las características de las GPUs actuales. Más aún, la curva de aprendizaje de CUDA es considerablemente menor que la requerida para realizar cómputo de propósito general en la GPU por medio de shaders y texturas. Por lo tanto, se puede esperar un incremento en las investigaciones y aplicaciones de cómputo general en GPUs a medida que se propague su conocimiento en las diferentes áreas de las ciencias de la computación.

Capítulo 6

Trabajos a Futuro

Dado que la biblioteca Cusp permite utilizar OpenMP o TBB en lugar de CUDA como ambiente de ejecución paralelo, únicamente haciendo cambios en archivos de configuración, sería interesante hacer pruebas de los métodos iterativos implementados en este trabajo en estos otros dispositivos ofrecidos por Cusp. Esto permitiría evaluar diferentes arquitecturas con un esfuerzo relativamente bajo.

Los métodos implementados están precondicionados por la derecha, sin embargo, en ciertas ocasiones es más fácil conseguir una buena matriz precondicionadora por la izquierda. Es por ello que se recomienda la implementación de dichos métodos precondicionados por la izquierda para continuar los experimentos con ellos.

El algoritmo del precondicionador SPAI se diseño para trabajar en una sola tarjeta de vídeo, sin embargo, para matrices grandes no es posible calcular todas las columnas al mismo tiempo en una única tarjeta. Por lo tanto, se propone expandir el algoritmo propuesto de manera que en computadoras con múltiples tarjetas gráficas, se distribuya el trabajo de la construcción del precondicionador entre ellas.

Por otro lado, el algoritmo SPAI implementado en este trabajo utiliza un patrón de dispersión prescrito, el mayor problema de este enfoque es que conseguir un buen patrón de dispersión, no es trivial. Esto motiva a proponer la implementación del precondicionador SPAI de forma adaptativa, una manera de hacerlo es como la mostrada en [22].

Bibliografía

[1] W. E. Arnoldi. "The Principle of Minimized Iterations in the Solution of the Matrix Eigenvalue Problem." *Quart. Appl. Math.*, vol. 9, pp. 17–29, 1951.

[2] N. Bell y M. Garland. "Efficient Sparse Matrix-Vector Multiplication on CUDA." NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, dec 2008.

[3] N. Bell y M. Garland. "Implementing sparse matrix-vector multiplication on throughput-oriented processors." En *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–11. ACM, New York, NY, USA, 2009.

[4] N. Bell y M. Garland. "Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations.", 2010. Version 0.3.0.

[5] A. Benson y D. J. Evans. "Algorithm 80: An algorithm for the solution of periodic quindiagonal systems of linear equations." *The Computer Journal*, vol. 16, no. 3, pp. 278–279, agosto 1973.

[6] M. Benzi. "Preconditioning Techniques for Large Linear Systems." En *PDP '10: Journal of Computational Physics, 2002*, pp. 182, 418–477. Emory University, Mathematics and Computer Science Department, Atlanta, Georgia 30322, 2002.

[7] M. Benzi y M. Tuma. "A comparative study of sparse approximate inverse preconditioners." *Applied Numerical Mathematics: Transactions of IMACS*, vol. 30, no. 2–3, pp. 305–340, junio 1999.

[8] R. F. Boisvert, R. Pozo, K. Remington, R. Barrett, y J. J. Dongarra. "Matrix Market: A Web Resource for Test Matrix Collections." En *The Quality of Numerical Software: Assessment and Enhancement*. http://math.nist.gov/MatrixMarket.

[9] R. L. Burden. *Numerical analysis*. Brooks/Cole, pub-BROOKS-COLE: adr, eighth edición, 2005.

[10] N. Corporation. "NVIDIA CUDA C Programming Guide Version 4.2."

[11] N. Corporation. "GPU Computing." http://www.nvidia.es/page/gpu_computing.html, 2010.

[12] R. Couturier y S. Domas. "Sparse systems solving on GPUs with GMRES." *J. Supercomput.*, vol. 59, no. 3, pp. 1504–1516, marzo 2012.

[13] B. N. Datta. *Numerical Linear Algebra and Applications*. Brooks/Cole, Pacific Grove, CA, 1995.

[14] T. A. Davis y Y. Hu. "The University of Florida Sparse Matrix Collection." En *ACM Transactions on Mathematical Software (to appear)*. http://www.cise.ufl.edu/research/sparse/matrices.

[15] J. J. Dongarra, I. S. Duff, D. C. Sorensen, y H. A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[16] I. S. Duff, A. M. Erisman, C. W. Gear, y J. K. Reid. "Sparsity structure and Gaussian elimination." *ACM SIGNUM Newsletter*, vol. 23, no. 2, pp. 2–8, abril 1988.

[17] I. S. Duff, A. M. Erisman, y J. K. Reid. *Direct methods for sparse matrices*. Oxford University Press, Inc., New York, NY, USA, 1986.

[18] R. W. Freund. "A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems." *SIAM Journal on Scientific Computing*, vol. 14, no. 2, pp. 470–482, marzo 1993.

[19] G. H. Golub y C. F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[20] J. Hoberock y N. Bell. "Thrust: A Parallel Template Library.", 2010. Version 1.4.0.

[21] D. Kincaid y W. Cheney. *Numerical analysis: mathematics of scientific computing (2nd ed)*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 1996.

[22] M. Lukash, K. Rupp, y S. Selberherr. "Sparse approximate inverse preconditioners for iterative solvers on GPUs." En *Proceedings of the 2012 Symposium on High Performance Computing*, HPC '12, pp. 13:1–13:8. Society for Computer Simulation International, San Diego, CA, USA, 2012.

[23] G. MARKALL. "Accelerating Unstructured Mesh Computational Fluid Dynamics on the NVidia Tesla GPU Architecture." 2009.

[24] Y. Saad. *Iterative Methods for Sparse Linear Systems, 2nd edition*. SIAM, Philadelpha, PA, 2003.

[25] Y. Saad y M. H. Schultz. "GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems." vol. 7, pp. 856–869, 1986.

[26] F. Veselý. *Iterative GPGPU Linear Solvers for Sparse Matrices*. Tesis de Maestría, Czech Technical University in Prague, mayo 2008.

[27] M. Wang, H. Klie, M. Parashar, y H. Sudan. "Solving Sparse Linear Systems on NVIDIA Tesla GPUs." En *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pp. 864–873. Springer-Verlag, Berlin, Heidelberg, 2009.