UNIVERSIDAD NACIONAL DE CÓRDOBA

Facultad de Matemática,
Astronomía y Física

Paralelización de algoritmos para verificación simbólica de modelos probabilísticos.

Tealdi, Matías D.

Directores: D'Argenio, Pedro R.
Bederián, Carlos

17 de Diciembre de 2013

Agradeciminetos

Quisiera agradecer en primer lugar a los directores de mi trabajo, Doctor Pedro D'Argenio y Licenciado Carlos Bederían, y al Doctor Nicolás Wolovick que me apoyaron y ayudaron a lo largo de este trabajo brindando sus conocimientos y ayudado a desarrollar mi interés en las Ciencias de la Computación.

También quisiera agradecer a mis amigos dentro y fuera de la universidad. Principalmente a mi amigo y compañero Gastón Ingaramo con quién hemos formado un muy buen grupo de trabajo y compartido muchísimas experiencias a lo largo de estos años. Finalmente quiero agradecer a mis padres Raúl e Ida y a mis hermanos Valeria y Lucas que me han alentado y brindado su apoyo constante.

Resumen

La verificación de modelos es una técnica útil para el análisis de modelos complejos. En particular, la verificación de modelos probabilistas se basa en una especificación formal de un sistema que presenta comportamientos probabilistas o estocásticos y una propiedad lógica. Esta propiedad es evaluada automáticamente para detectar posibles fallas, posibilitando el análisis cuantitativo del sistema.

La principales variables que afectan el alcance de esta técnica son los tamaños de los modelos y la complejidad computación de los cálculos numéricos involucrados. La verificación simbólica ataca el tamaño de los modelos con estructuras de datos compactas. Aún así, los cálculos numéricos demandan una gran cantidad de recursos computacionales.

La motivación de este trabajo es mejorar los tiempos de ejecución de los algoritmos numéricos, específicamente el método de Jacobi, a través de la arquitectura de GPU Computing sin perder los beneficios, antes nombrados, de las estructuras simbólicas.

Clasificación: D.2.4 Software/Program Verification, G.1.0 General (Numerical Analysis), D.1.3 Concurrent Programming.

Palabras Claves: model checking, sistemas de ecuaciones lineales, computación de alta performance, modelos probabilistas.

Índice general

Índice general IV

Índice de figuras VII

Nomenclature VII

1. Introducción 1
- 1.1. Motivación . . . 1
- 1.2. Objetivos del Trabajo . . . 2
- 1.3. Estructura de la tesis . . . 3

2. Verificación de modelos probabilistas 5
- 2.1. Modelos Probabilistas . . . 5
  - 2.1.1. Cadenas de Markov de tiempo discreto . . . 5
  - 2.1.2. Cadenas de Markov de tiempo continuo . . . 6
- 2.2. Formalismos para la especificación de propiedades probabilistas . . . 7
  - 2.2.1. PCTL . . . 7
  - 2.2.2. CSL . . . 9
- 2.3. Verificación de modelos probabilistas . . . 10
  - 2.3.1. Verificación de DTMC . . . 10
  - 2.3.2. Verificación de CTMC . . . 12
- 2.4. Métodos iterativos de solución de sistemas de ecuaciones lineales . . . 14
  - 2.4.1. Método de Jacobi . . . 15

3. Modelo Computacional de arquitectura GPGPU 17
- 3.1. ¿Por qué utilizar GPGPU? . . . 17
- 3.2. Desde una estructura fija a una programable . . . 18
  - 3.2.1. CUDA: Historia . . . 19
- 3.3. CUDA como modelo de programación escalable . . . 19
  - 3.3.1. Modelo de programación . . . 20
- 3.4. Implementación del hardware CUDA . . . 23

3.4.1. Arquitectura SIMT 23
3.4.2. Características del Hardware Multi-hilo 24
3.5. Técnicas de Rendimiento 25

4. Paralelización **27**
4.1. Algoritmo genérico del método de Jacobi 27
4.2. Representación de Matrices y vectores en PRISM 27
4.2.1. Representación rala de matrices 29
4.2.2. Diagrama de decisión binaria multi-terminal 29
4.3. Motor Híbrido 32
4.3.1. Primer Algoritmo 33
4.3.2. Método mejorado 34
4.3.3. Una vuelta más de tuerca 35
4.4. Paralelización en arquitectura GPGPU 37
4.4.1. Adaptación de la estructura de representación MTBDD 38
4.4.1.1. Primera Representación 38
4.4.1.2. Segunda Representación 41
4.4.2. Algoritmos en GPGPU 43
4.4.2.1. Algoritmo de Multiplicación Matriz-Vector 44
4.4.2.2. Algoritmo de Reducción 45

5. Resultados **49**
5.1. Casos de estudio 49
5.1.1. Colas Tandem [Tandem] 49
5.1.2. Sistema de Sondeo Cíclico [Polling] 49
5.1.3. Sistema de Manufactura Kanban [Kanban] 50
5.1.4. Sistema de Manufactura Flexible [FMS] 50
5.1.5. Estaciones de Trabajo [Cluster] 51
5.1.6. Protocolo de Retransmisión Acotada [BRP] 51
5.2. Experimentos y Análisis 52
5.2.1. Medición de rendimiento en el procesamiento 52
5.2.2. Entorno de prueba 52
5.2.3. Experimentos 52
5.2.3.1. Medición de memoria para la paralelización 53
5.2.3.2. Resultados de rendimiento 55

6. Conclusiones **61**
6.1. Conclusiones académicas 61
6.2. Conclusión de resultados obtenidos 62

References 64

Índice de figuras

3.1. Escalabilidad Automática 20
3.2. Grilla de bloques de hilos 21
3.3. Jerarquía de memoria 22

4.1. Ejemplo de MTBDD M 30
4.2. Reducción de un MTBDD M 31
4.3. Matriz M y su representación en MTBDD 32
4.4. Matriz M y su representación en Offset-labelled MTBDD 35
4.5. Matriz M y su representación en offset-labelled MTBDD 36
4.6. Producto Matriz - vector 42

5.1. Entorno de prueba 52
5.2. Memoria de representación 54
5.3. Memoria de representación en proporción a memoria base 56
5.4. Tabla de resultados del caso de estudio FMS 57
5.5. Tabla de resultados del caso de estudio Kanban 57
5.6. Tabla de resultados del caso de estudio Polling-before 57
5.7. Tabla de resultados del caso de estudio Polling-s1 58
5.8. Tabla de resultados del caso de estudio Cluster 58
5.9. Tabla de resultados del caso de estudio Tandem 58
5.10. Tabla de resultados del caso de estudio BRP 58

Capítulo 1

Introducción

1.1. Motivación

En la actualidad los sistemas computarizados juegan un rol importante en todos los aspectos de nuestras vidas. Muchas veces no somos concientes de que las computadoras y el software están involucrados en nuestras actividades diarias. Muchos sistemas de control de autos modernos están basados en sistemas computarizados embebidos, como por ejemplo frenos, airbag, control de velocidad crucero. Teléfonos celulares, sistemas de comunicación, instrumentos médicos son otros ejemplos típicos donde podemos encontrar este tipo de sistemas.

Un patrón común es el crecimiento de la complejidad del software. En la mayoría de los casos su correcto funcionamiento es crucial para garantizar la seguridad humana e integridad de los recursos puestos en juego. Algunos casos muy famosos en la falla de sistemas críticos han sido la explosión del cohete Ariane 5 o la muerte de pacientes por exceso de radiación en la máquina de radioterapia Therac-25. Por ello, uno de los desafíos principales de la ciencia de la computación es proveer de formalismos, técnicas y herramientas que permitan garantizar el buen desempeño y funcionamiento correcto de sistemas computacionales.

En las últimas décadas se han tratado de perfeccionar distintas técnicas para tratar de combatir estos problemas. Algunos ejemplos son las técnicas de verificación dinámicas como *Testing*. Estas se llevan a cabo durante la ejecución del programa comprobando dinámicamente el buen funcionamiento. Otras técnicas estáticas comprueban a través de cálculos matemáticos, inspección de código o evaluación lógica que el software cumpla con los requerimientos.

La *verificación de modelos* (Model Checking) es una técnica de verificación formal que permite verificar propiedades sobre el comportamiento de un sistema en base a un modelo del mismo a través de la inspección **estática** y **sistemática** de todos sus estados. Su éxito se basa en que la validación es completamente automática.

Spin [Hol97], NuSMV [CCGR99] o Uppaal [BLL+96] son herramientas de verificación de modelos que han sido utilizadas para verificar algoritmos críticos de control. Ejemplos de sistemas verificados son el sistema de control de las barreras contra inundaciones de Holanda, el firmware de un switch telefónico de la empresa Lucent, protocolos de comunicación de audio en tiempo real, un controlador de caja de cambios, sistemas de control de vuelo e incluso ciertos componentes de distintas sondas espaciales.

El problema principal que enfrenta la verificación de modelos es la explosión combinatorial de estados al incrementar el número de variables en un programa. Esto impacta en la cantidad de memoria necesaria para su representación y en la complejidad computacional de los algoritmos numéricos. Por ello, uno de los objetivos de estos últimos años en este campo de la computación ha sido encontrar técnicas que permitan analizar modelos de cada vez mayor tamaño. Un avance reciente en este ámbito son las técnicas de verificación simbólica. Esta utiliza estructuras de diagramas de decisión binaria (BDD) que permiten una representación más compacta haciendo posible el análisis de sistemas de mayor tamaño.

Es importante destacar que ninguna de las herramientas nombradas permite producir código sin errores. En particular, toda ellas requieren de un grado de asistencia manual a la hora de escribir la especificación formal del sistema y definir las propiedades a verificar. Sin embargo, la combinación conjunta de éstas y otras técnicas permiten reducir considerablemente la probabilidad de errores en un programa.

1.2. Objetivos del Trabajo

PRISM [KNP11] es un verificador de modelos probabilistas desarrollado inicialmente por investigadores de la Universidad de Birmingham y actualmente mantenido por la Universidad de Oxford. Este esta distribuido bajo la licencia GNU General Public License.

PRISM es una herramienta para el modelado y análisis formal de sistemas que exhiben un comportamiento aleatorio o estocástico. Este soporta un gran rango de modelos probabilistas y ha sido utilizado para el análisis de muchos sistemas de diferentes dominios, incluyendo protocolos de comunicación y multimedia, algoritmos distribuidos aleatorios, protocolos de seguridad, sistemas biológicos, entre otros.

Actualmente PRISM se encuentra en la versión 4.1, lanzada el 20 de diciembre del 2012. Esta versión cuenta con tres motores númericos distintos, los cuales implementan diferentes estructuras de datos y algoritmos para la solución de los cálculos numéricos. Estos motores son *Explícito*, *MTBDD* e *híbrido*. En el primer caso los los algoritmos numéricos se realizan sobre matrices ralas. El motor MTBDD, implementa estructuras del tipo Diagrama de decisión binario multi-terminal, uno de los últimos grandes avances en las técnicas de verificación de modelos. Por último el motor híbrido toma algunas de las ventajas de los dos motores anteriores. Las diferentes implementaciones están desarrolladas para arquitecturas

de un solo procesador, esto limita el modelado y análisis a sistemas de tamaño acotado, debido a la explosión de estados y la complejidad computacional de los cálculos.

Por 30 años, uno de los principales métodos para incrementar el rendimiento de los procesadores han sido el incremento del la velocidad del reloj del procesador. Sin embargo, en los últimos años, los fabricantes de procesadores han estado forzados a buscar una alternativa diferente para mejorar el poder computacional. Esto se debe a varias limitaciones físicas en la fabricación de los circuitos integrados. Algunas de las limitaciones son el aumento en el consumo y el calor generado [Sut05]. Entre las arquitecturas alternativas más comunes se encuentran el multiprocesamiento simétrico (SMP), clusters, unidades de procesamiento gráfico de propósito general (GPGPU) entre otras.

Las unidades de procesamiento gráfico de propósito general nacieron como una extensión de las unidades de procesamiento gráfico, destinadas para realizar cálculos intensivos especialmente en el ámbito científico y simulación. Estas consiguen incrementar el poder de cómputo a través del procesamiento vectorial, alto rendimiento en el acceso de memoria con ciertos patrones y ocultamiento de latencia de acceso de memoria por medio del paralelismo masivo. Estas técnicas de paralelismo convirtieron a las GPGPU en una arquitectura de alto desempeño con gran proyección en los próximos años.

Varios trabajos se han propuesto para la paralelización con arquitecturas GPGPU del motor explícito de PRISM [BESW10] [WB12] debido a que los métodos numéricos sobre las estructuras ralas se adaptan muy bien en la arquitectura de GPU. Principalmente se busca mejorar el rendimiento del producto matriz-vector, un problema ya muy tratado dentro del ámbito de programación en GPU. El problema del motor explícito radica en que necesita demasiada memoria para la representación del modelo, llevando a disminuir el tamaño de los problemas a analizar. Recordemos que por ello, la incorporación de las estructuras simbólicas en los verificadores ha sido un gran avance en los últimos años.

El objetivo de este trabajo es aumentar el rendimiento del motor híbrido de PRISM a través de la arquitectura GPU. Mas específicamente, el trabajo se centra en la paralelización del método de Jacobi y Jacobi extendido para la solución de sistemas lineales de ecuaciones. Como destacamos anteriormente, el motor híbrido combina las mejoras del motor MTBDD a través de la compresión de la representación del modelo con estructuras simbólicas y el rendimiento del motor explícito a través de la inyección de pequeñas matrices.

1.3. Estructura de la tesis

La tesis está estructurada de la siguiente manera :

- Capítulo 2 se introduce el formalismo de la verificación de modelos.

- Capítulo 3 se explican los conceptos de las arquitecturas GPGPU y las técnicas de programación para lograr su mejor desempeño.
- Capítulo 4 se desarrollan las distintas variaciones de las estructuras de representación utilizadas por el motor híbrido. Además, se explican las modificaciones hechas sobre las mismas para la paralelización del motor y los algoritmos de GPGPU que las utilizan.
- Capítulo 5 se presentan los resultados obtenidos en la paralelización a través de la medición de casos de estudio propuestos por PRISM.
- Capítulo 6 se presentan las conclusiones del trabajo y futuros campos de investigación.

Capítulo 2

Verificación de modelos probabilistas

En este capítulos, presentaremos todo el marco teórico para el modelo de sistemas probabilistas y para la verificación de propiedades sobre el mismo. Solo nos concentraremos en los formalismos utilizados por la herramienta en la que se centra el trabajo. En la Sección 2.1 analizaremos los formalismos de modelado de sistemas probabilistas. En la Sección 2.2 detallaremos los distintos formalismos para el modelado de propiedades a analizar. Las secciones siguientes cubren los diferentes aspectos prácticos que son necesarios para la implementación del trabajo. En la Sección 2.3 estudiaremos como el formalismo es llevado a un sistema lineal de ecuaciones para el análisis de la propiedad. Finalmente, en la Sección 2.4 repasaremos el método numérico iterativo de solución de sistemas lineales que se adapta de forma más natural a nuestros requerimientos.

2.1. Modelos Probabilistas

La verificación de modelos tradicional involucra la verificación de propiedades sobre sistemas de transición etiquetados. En el contexto de modelos probabilistas, usaremos modelos que además incorporan información sobre la probabilidad de transición entre los diferentes estados que ocurren en el modelo. En este trabajo solo analizaremos dos tipos de modelos probabilistas: cadenas de Markov de tiempo discreto y cadenas de Markov de tiempo continuo. Cabe destacar que PRISM soporta otros tipos de modelado como procesos de decisión de Markov, sin embargo, este trabajo involucra la paralelización de los primeros.

2.1.1. Cadenas de Markov de tiempo discreto

Es el método de modelado de sistemas probabilistas más simple, las *Cadenas de Markov de tiempo discreto* (DTMCs) definen simplemente la probabilidad de la transición entre estados. Este puede ser utilizado para modelar un sistema de transición simple o varios sistemas de transición sincronizados. Este tipo de modelo se basa en dos principios fundamentales.

El primero es que el sistema evoluciona en intervalos discretos de tiempo. El segundo es que las probabilidades de pasar del estado actual al siguiente solo dependen del estado actual y no de lo que ha ocurrido en el pasado, dando noción de independencia entre estados.

Definiremos a un **DTMC** como una tupla $M = (S, s_{init}, P, L)$ donde:

- S es un conjunto de estados
- s_{init} es el estado inicial
- $P : S \times S \to [0, 1]$ es una matriz de probabilidades de transición.
  donde $\sum_{s' \in S} P(s', s) = 1$ para todo $s \in S$
- $L : S \to 2^{AP}$ función de etiquetado de estados

Destaquemos que las restricciones sobre P nos obligan a que cada estado tenga al menos una transición. Por ello definimos los estados terminales como aquellos estados que tienen sólo una transición a si mismo, es decir $P(s, s) = 1$. La función de etiquetado L, asigna cada estado con un conjunto de propiedades interesantes que se cumplen en dicho estado.

Una ejecución del sistema es un camino o traza a través del DTMC. Un *traza* ω es una secuencia no vacía de estados $s_0, s_1, \ldots$ donde $s_i \in S$ y $P(s_i, s_{i+1}) > 0$ para todo $i \geq 0$. Las *trazas* pueden ser finitas o infinitas. Así, la noción de probabilidad puede ser extendida a las trazas. El cálculo de probabilidad de una traza finita es simple, la probabilidad de una traza $s_0, s_1, \ldots, s_n$ es la probabilidad dada por $P(s_0, s_1) \times P(s_1, s_2) \times \cdots \times P(s_{n-1}, s_n)$. Esta definición de probabilidad puede ser extendida a trazas infinitas, pero la definición requiere de herramientas matemáticas más avanzadas por lo que serán omitidas. Para mas detalles consulte [KSK66].

2.1.2. Cadenas de Markov de tiempo continuo

El segundo tipo de modelo que vamos a considerar son las *Cadenas de Markov de tiempo continuo* (CTMC). Estas extienden los modelos DTMC. Mientras que en un DTMC las transiciones corresponden a una distribución de tiempo discreta, en un CTMC, las transiciones corresponden a transiciones en tiempo real. Cada transición es etiquetada con un parámetro r el cual define la demora antes que esta ocurra. Esta demora es producida por una distribución de probabilidad exponencial negativa con parámetro r.

Definiremos a un **CTMC** como una tupla $M = (S, s_{init}, R, L)$ donde:

- S es un conjunto de estados
- s_{init} es el estado inicial
- $P : S \times S \to \mathbb{R}_{\geq 0}$ es una matriz de parámetros r de transición.

- $L : S \to 2^{AP}$ función de etiquetado de estados

La matriz de parámetros r de transición, como su nombre lo indica, nos proporciona los parámetros r de la distribución probabilistica a diferencia de la probabilidad misma en el caso de DTMC. Para estados s y s' la probabilidad de la transición de s a s' transcurridas t unidades de tiempo será $1 - \epsilon^{-R(s,s')\dot{t}}$. Por lo general hay más de un estado s' con $R(s, s') > 0$. La transición tomada será la primera disponible. Así, se puede demostrar que $P(s, s')$ moviéndose en un paso simple es $\frac{R(s,s')}{\sum_{s'' \in S} R(s,s'')}$ (a menos que s no tenga transiciones, en tal caso definimos $P(s, s') = 1$ si $s' = s$ y $P(s, s') = 0$ si $s' \neq s$).

Al igual que en los modelos DTMC, podemos definir las ejecuciones de sistema como caminos o trazas finitas o infinitas. En el caso de **CTMC**, una *traza* es una secuencia $s_0 t_0 s_1 t_1 \ldots$ donde $R(s_i, s_{i+1}) > 0$ y $t_i \in \mathbb{R}_{>0}$ para todo $i \geq 0$. El valor de t_i representa el tiempo transcurrido en el estado s_i. Nuevamente se pueden definir las probabilidades de las trazas finitas e infinitas, pero vamos a obviar debido a su complejidad matemática.

Pareciera ser que en los CTMC estamos en presencia de una distribución de probabilidad discreta. Si bien es cierto que las probabilidades de transición de un estado a otro son discretas (pues el conjunto de estados es finito), los DTMC no toman en cuenta los tiempos asociados a cada transición. Estos tiempos nos permiten describir un sistema que evoluciona con un mismo patrón que otro pero dos veces más lento, o un sistema con transiciones más rápidas y más lentas.

2.2. Formalismos para la especificación de propiedades probabilistas

Ahora describiremos el formalismo utilizado para especificar las propiedades a verificar en nuestro modelo. Analizaremos la lógica temporal PCTL la cual es interpretada sobre modelos DTMC y la lógica CSL interpretada en modelos CTMC.

2.2.1. PCTL

La lógica de árbol computacional probabilista o PCTL (probabilistic computational tree logic) presentado en [HJ94] es una extensión de la lógica CTL a modelos probabilistas, y nos permite en particular trabajar con los modelos de tiempo discreto DTMC. La gramática formal para esta lógica es la siguiente:

$$\phi ::= true \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \mathcal{P}_{\bowtie p}[\psi]$$
$$\psi ::= \mathcal{X}\phi \mid \phi \mathcal{U}^{\leq k} \phi \mid \phi \mathcal{U} \phi$$

donde a denota una proposición atómica, k un número natural y $\bowtie$ uno de los operadores $\{<, \leq, >, \geq\}$.

Las fórmulas de esta lógica se dividen en dos tipos: *fórmulas de estado* ϕ y *fórmulas de camino* ψ . Las fórmulas de estado hacen referencia al comportamiento del sistema en un instante de tiempo. De esta forma, podemos chequear la validez de una proposición atómica a para el estado actual, así como utilizar los conectores $\wedge$ y $\neg$ (y otros derivados como **false**, $\vee$ y $\rightarrow$) para construir sentencias más complejas.

Pero el verdadero trabajo es llevado a cabo por el operador $\mathcal{P}$ que nos permite hablar de la evolución del sistema a partir de un punto dado. Este operador tiene como argumentos una fórmula de camino junto con una cota de probabilidad, y permite establecer si la probabilidad de seguir un camino que satisfaga la fórmula dada está dentro de la cota provista.

Debemos explicar entonces como se interpretan las fórmulas de camino. Podemos pensar este tipo de fórmula como una forma de seleccionar un subconjunto de trazas de ejecución dentro del conjunto de trazas posibles. De esta forma, el operador $\mathcal{X}$ (next) sólo es válido para trazas en las cuales, después de haber realizado una transición del estado inicial, el estado que sigue satisface una fórmula ϕ dada.

Los operadores $\mathcal{U}$ (until) y $\mathcal{U}^{\leq k}$ (until acotado) por otro lado toman como parámetro dos formulas de estado ϕ_1 y ϕ_2, y restringen las trazas válidas a aquellas en donde la primera condición ϕ_1 debe cumplirse hasta llegar a un estado que satisfaga la segunda condición ϕ_2. El segundo operador permite ajustar todavía más la restricción estableciendo un límite máximo de k transiciones para llegar a un estado donde la segunda condición sea verdadera.

Fórmulas PCTL sobre modelos DTMC

Para un DTMC (S, s_{init}, P, L), estado $s \in S$ y fórmula ϕ, escribiremos $s \models \phi$ para indicar que ϕ se satisface en s. Alternativamente diremos que ϕ se mantien en s o que ϕ es verdadero en s. Denotaremos por $Sat(\phi)$ al conjunto $\{s \in S \mid s \models \phi\}$ de todos los estados que satisfacen la fórmula ϕ. De forma similar, para una camino ω que satisface una fórmula de camino ψ escribiremos $\omega \models \psi$. Ahora podemos describir formalmente la semántica de PCTL sobre un modelo DTMC. Para un camino ω:

$$
\begin{aligned}
\omega \models \mathcal{X}\phi \quad &\iff \omega(1) \models \phi \\
\omega \models \phi_1 \mathcal{U} \phi_2 \quad &\iff \exists i \leq k.(\omega(i) \models \phi_2 \wedge \omega(j) \models \phi_1 \forall j < i) \\
\omega \models \phi_1 \mathcal{U}^{\leq k} \phi_2 \quad &\iff \exists k \geq 0.\omega \models \phi_1 \mathcal{U} \phi_2
\end{aligned}
$$

y para un estado $s \in S$:

$$\begin{aligned} s &\models true && \text{para todo } s \in S \\ s &\models a && \iff a \in L(s) \\ s &\models \phi_1 \wedge \phi_2 && \iff s \models \phi_1 \, y \, s \models \phi_2 \\ s &\models \neg\phi && \iff s \not\models \phi \\ s &\models \mathcal{P}_{\bowtie p}[\psi] && \iff p_s(\psi) \bowtie p \end{aligned}$$

donde $p_s(\psi) = Prob_s(\{\omega \in Path_s \mid \omega \models \psi\})$.

2.2.2. CSL

Los modelos continuos tienen la característica adicional de contar con un período variable de transición entre un estado y otro. La lógica utilizada en estos casos se denomina *lógica estocástica continua* o CSL (continuous stochastic logic), y fue presentada en [ASSB96] y ampliada en [BKH99]. Sus operadores son similares a los de PCTL pero sustituyen los valores discretos de tiempo por valores continuos. La gramática para esta lógica es:

$$\begin{aligned} \phi &::= true \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \mathcal{P}_{\bowtie p}[\psi] \mid \mathcal{S}_{\bowtie p}[\phi] \\ \psi &::= \mathcal{X}\phi \mid \phi \mathcal{U}^{\leq t} \phi \mid \phi \mathcal{U} \phi \end{aligned}$$

Los operadores compartidos funcionan igual que antes, salvo en el caso del **until acotado** que en vez de tomar como argumento una cota máxima para el número de transiciones, lleva en cambio un número real que representa el tiempo máximo para que se satisfaga la segunda fórmula de estado.

El único operador nuevo es $\mathcal{S}$, que nos permite especificar lo que se conoce como propiedades de estado estable (steady-state properties). Es decir, a diferencia de $\mathcal{P}$ que calcula probabilidades sobre ejecuciones posibles que satisfagan una fórmula de camino, el operador $\mathcal{S}$ calcula la probabilidad de que a largo plazo (cuando el tiempo tiende a infinito) el modelo se estabilice en un estado que satisfaga la propiedad dada. Esto es una herramienta muy útil cuando tenemos un modelo con una fase de inicialización de tiempo variable, pero lo que verdaderamente nos interesa es el comportamiento final del sistema.

Obviaremos el análisis de la semántica de formulas CSL en CTMC debido a que requiere de muchas definiciones que por simplicidad no dimos. Este puede ser consultado en [ASSB96].

2.3. Verificación de modelos probabilistas

En esta sección vamos a resumir los algoritmos para verificar los dos casos discutidos anteriormente: PCTL sobre DTMC y CSL sobre CTMC. El algoritmo de verificación de modelos toma un propiedad PCTL o CSL, un modelo del tipo apropiado, una fórmula ϕ en la lógica y devuelve el conjunto $Sat(\phi)$ que contiene los estados del modelo en los cuales se satisface la propiedad ϕ.

La estructura general de los algoritmos son similares en ambos casos, y son una extensión del algoritmo de verificación de modelos sobre CTL presentado en [CES86]. Primero construimos el árbol de parseo de la fórmula ϕ. Cada nodo del árbol corresponde a una sub-fórmula de ϕ y el nodo raíz corresponde a la fórmula ϕ propiamente. Las hojas del árbol corresponden a *true* o a una preposición atómica a. Entonces, para resolver una formula ϕ se opera desde las hojas hacia la raíz, recursivamente generando el conjunto de estados que satisfacen dicha sub-fórmula. Así, en la raíz, determinaremos si cada estado del modelo satisface ϕ.

La verificación de operadores no probabilistas de ambas lógicas es realizada de forma idéntica para DTMC y CTMC: es trivial deducir que estados del modelo satisfacen una proposición atómica. Los operadores lógicos como conjunción y negación son también simples. Los casos no triviales son los operadores $\mathcal{P}$ y $\mathcal{S}$. En estos casos es necesario computar las probabilidades relevantes y luego identificar los estados que satisfacen las restricciones dadas en la fórmula. El cálculo de probabilidad de cada operador es descripto en la siguiente sección. La complejidad del algoritmo en el peor caso es polinomial en el tamaño del modelo. Por lo tanto, la complejidad de verificar una formula PCTL sobre un modelo DTMC o una fórmula CSL sobre un modelo CTMC es lineal en con respecto al tamaño de la lógica temporal y polinomial en el tamaño del modelo.

2.3.1. Verificación de DTMC

Para la verificación del operador $\mathcal{P}_{\bowtie p}[\psi]$ sobre un modelo DTMC $(S, s_{init}, \mathbf{P}, L)$, necesitaremos calcular la probabilidad de satisfacer la fórmula de camino ψ partiendo de cada estado s. Cuando ψ es uno de los operadores **next** ($\mathcal{X}\phi$), **until** ($\phi_1 \mathcal{U} \phi_2$), o **until acotado** ($\phi_1 \mathcal{U}^{\leq k} \phi_2$), calcularemos para todos los estados $s \in S$, la probabilidad p_s de dicho operador respectivamente. Luego calcularemos el conjunto $Sat(\mathcal{P}_{\bowtie p}[\psi])$ como $\{s \in S \mid p_s(\psi) \bowtie p\}$. Debido a la naturaleza recursiva del algoritmo de verificación de PCTL, podemos asumir que los conjuntos $Sat(\phi)$, $Sat(\phi_1)$ o $Sat(\phi_2)$ son conocidos. Los algoritmos discutidos en la siguiente sección son analizados en [CY88, HJ94].

Operador PCTL next

Es simple de demostrar que $p_S(\mathcal{X}\phi) = \sum_{s' \in Sat(\phi)} \mathbf{P}(s, s')$. Asumiendo que tenemos un vector $\underline{\phi}$ con $\underline{\phi}(s) = 1$ si $s \models \phi$ y 0 en otro caso, podemos calcular el vector $\underline{p(\mathcal{X}\phi)}$ de la propiedad requerida como: $\underline{p(\mathcal{X}\phi)} = \mathbf{P}.\phi$. Esto solo requiere un operación simple de producto matriz-vector.

Operador PCTL until acotado

Primero dividiremos todos los estados en tres conjuntos disjuntos: S^{no}, S^{yes} y $S^{?}$. El conjunto $S^{no} = S \setminus (Sat(\phi_1) \cup Sat(\phi_2))$ y $S^{yes} = Sat(\phi_2)$ contienen los estados para los cuales $p_s(\phi_1 \mathcal{U}^{\leq k} \phi_2)$ es trivialmente 0 o 1, respectivamente. El conjunto $S^{?} = S \setminus (S^{no} \cup S^{yes})$ el cual contiene el resto de los estados. Para estos estados tenemos :

$$p_s(\phi_1 \mathcal{U}^{\leq k} \phi_2) = \begin{cases} 0 & \text{si } k = 0 \\ \sum_{s' \in S} \mathbf{P}(s, s').p_s(\phi_1 \mathcal{U}^{\leq k-1} \phi_2) & \text{si } k \geq 1 \end{cases}$$

Si definimos la matriz $\mathbf{P}'$ como :

$$\mathbf{P}'(s, s') = \begin{cases} \mathbf{P}(s, s') & \text{si } s \in S^{?} \\ 1 & \text{si } s \in S^{yes} \text{ y } s = s' \\ 0 & \text{otro caso} \end{cases}$$

Abreviando el vector de probabilidades $\underline{p(\phi_1 \mathcal{U}^{\leq k} \phi_2)}$ como p_k, el cálculo es de la siguiente forma. Para $k = 0$, $\underline{p_0(s) = 1}$ si $s \in S^{yes}$ y 0 en otro caso. Para $k > 0$, $p_k = \mathbf{P}'.p_{k-1}$. Resumiendo, en total se requieren k multiplicaciones matriz-vector.

Operador PCTL until

Este operador es ligeramente más complicado, ya que una ejecución podría requerir una cantidad de pasos arbitrariamente grande hasta llegar a un estado que satisfaga ϕ_2. Sin embargo, hay estados para los cuales podemos calcular la probabilidad directamente: aquellos que satisfagan ϕ_2 (para los cuales la probabilidad es 1), llamaremos a este conjunto S^{yes}, y aquellos que no satisfacen ϕ_1 ni ϕ_2 (para los cuales la probabilidad es 0), llamaremos a este conjunto S^{no}.

De hecho, es necesario ampliar estos dos conjuntos. Por ejemplo, si tenemos un es estado que satisface ϕ_1 y no satisface ϕ_2, pero todas las transiciones posibles llevan finalmente a un estado que satisface ϕ_2, entonces el estado considerado tendrá probabilidad 1. La definición más amplia es la siguiente:

- Si no existe una traza finita desde el estado inicial a un estado que satisfaga ϕ_2 , pasando

únicamente por estados intermedios que satisfagan ϕ_1, entonces la probabilidad del estado inicial es 0.

- Si no existe una traza finita desde el estado inicial a un estado con probabilidad 0 (i.e. que cumpla la condición del ítem anterior), sin pasar por ningún estado que satisfaga ϕ_2, entonces la probabilidad del estado inicial es 1.

En la práctica, estos conjuntos se generan con un algoritmo de punto fijo que se puede encontrar en [Par02b].

Una vez analizados estos casos, nos resta calcula $p_s(\phi_1 \mathcal{U} \phi_2)$ para los estados restantes $S^? = S \setminus (S^{no} \cup S^{yes})$. Esto puede ser calculado a través de la solución del siguiente sistema lineal de ecuaciones con variables $\{x_s | s \in S\}$:

$$x_s = \begin{cases} 0 & \text{si } s \in S^{no} \\ 1 & \text{si } s \in S^{yes} \\ \sum_{s' \in S} \mathbf{P}(s, s').x_{s'} & \text{si } s \in S^? \end{cases}$$

luego $p_s(\phi_1 \mathcal{U} \phi_2) = x_s$. Para reescribir este cálculo en la forma tradicional $\mathbf{A}.\underline{x} = \underline{b}$, definiremos $\mathbf{A} = \mathbf{I} - \mathbf{P'}$ donde $\mathbf{I}$ es la matriz identidad y $\mathbf{P'}$ esta dada por :

$$\mathbf{P'}(s, s') = \begin{cases} \mathbf{P}(s, s') & \text{si } s \in S^? \\ 0 & \text{otro caso} \end{cases}$$

y $\underline{b}$ es un vector sobre los estados con $\underline{b}(s)$ igual a 1 si $s \in S^{yes}$ y 0 en otro caso. Luego sistema de ecuaciones lineal es $\mathbf{A}.\underline{x} = \underline{b}$ puede ser resuelto con cualquier método numérico. Estos incluyen métodos directos como eliminación Gaussiana, o métodos iterativos como Jacobi y Gauss-Seidel. Ya que queremos resolver sistemas a partir de modelos muy grandes, nos concentraremos en los métodos iterativos.

2.3.2. Verificación de CTMC

Para las cadenas de Markov de tiempo continuo, la lógica utilizada es CSL que sólo presenta ligeras variaciones sobre la lógica PCTL. Más aún, como comentamos anteriormente, toda CTMC tiene asociada una cadena discreta cuya función de probabilidad de transición $\mathbf{P}$ se puede calcular en términos de la función $\mathbf{R}$ como $\mathbf{P}(x, y) = \frac{\mathbf{R}(x,y)}{\sum_{z \in S} \mathbf{R}(x,z)}$. Luego, para operadores como $\mathcal{X}$ y $\mathcal{U}$ que no dependen de la variable adicional de tiempo que agrega las CTMCs, la verificación se lleva a cabo sobre la DTMC asociada de forma idéntica que para las demás cadenas discretas.

Restan sólo dos operadores nuevos a considerar: $\mathcal{U}^{\leq t}$ y el operador de estado estable $\mathcal{S}_{\bowtie p}$. Para analizar estos operadores, primero definimos la *matriz generadora* Q de una CTMC

como :

$$Q_{x,y} = \begin{cases} \mathbf{R}(x,y) & \text{si } x \neq y \\ -\sum_{z \neq x} \mathbf{R}(x,z) & \text{si } x = y \end{cases}$$

Operador until acotado por tiempo

Recordemos que un camino sólo satisface $\phi_1 \mathcal{U}^{\leq t} \phi_2$ si se alcanza un estado que satisface ϕ_2 antes del tiempo t, y todos los estados anteriores satisfacen ϕ_1. Luego, igual que antes tenemos dos conjuntos S^{no} y S^{yes} donde la probabilidad es trivialmente 0 y 1, respectivamente. Es decir los estados que no satisfacen ϕ_1 ni ϕ_2 para el primer caso y los estados que satisfacen ϕ_2 para el segundo caso.

Una primera idea presentada en [BHHK00], es modificar la CTMC eliminando las transiciones que salen de dichos estados. Luego, un camino que satisface la fórmula $\phi_1 \mathcal{U}^{\leq t} \phi_2$ en un instante anterior al tiempo t debe pasar por un estado que satisfaga ϕ_2, y como en la nueva cadena este estado no tiene transiciones salientes, seguirá estando en el mismo estado en el instante t.

Por otro lado, los caminos que no satisfacen la fórmula son aquellos que no llegan a un estado que satisfaga ϕ_2 antes del instante t, y los que pasan por un estado con probabilidad 0 y quedan allí trabados al no existir transiciones salientes de dichos estados. Es decir que los caminos válidos para la cadena original son exactamente aquellos que en la nueva cadena se hallan en el instante t en un estado con probabilidad 1.

En otras palabras, verificar el operador $\mathcal{U}^{\leq t}$ en la cadena original es equivalente a verificar el operador $\mathcal{U}^{=t}$ en la nueva cadena. Esto último es un ejemplo de *análisis de comportamiento transitorio* (tramsient analysis) en cadenas de Markov de tiempo continuo, donde la palabra transitorio hace referencia al análisis del comportamiento después de un periodo determinado de tiempo.

Para llevar a cabo este análisis, se hace uso de una técnica llamada *uniformización*. En base a la matriz Q se calcula una matriz uniformizada $\mathbf{P}$ dada por $\mathbf{P} = \mathbf{I} + \mathbf{Q}/q$, donde q es cualquier entero positivo mayor o igual en valor absoluto a todos los elementos de la diagonal de $\mathbf{Q}$. Luego, en base a $\mathbf{P}$ podemos obtener la matriz con las probabilidades de transición para una CTMC y un intervalo de tiempo t de la siguiente manera:

$$P_t = \sum_{i=0}^{\text{inf}} \frac{e^{-qt} \cdot (qt)^i}{i!} \cdot p^i$$

En la práctica, la sumatoria se detiene una vez que se ha alcanzado la precisión necesaria. Finalmente, las probabilidades para cada estado se obtienen sumando las probabilidades de transición de dicho estado hacia todos los estados que satisfacen ϕ_2 dadas por P_t. En símbolos,

$P_t \cdot \underline{x}$ donde $x_i = 1$ si el i-ésimo estado satisface ϕ_2 y 0 en caso contrario.

Operador de estado estable

Consideremos ahora el operador de estado estable $\mathcal{S}_{\bowtie p}$. En este caso, queremos caracterizar el comportamiento del sistema a largo plazo. Es decir, nos interesa calcular la probabilidad de que el sistema evolucione hacia un estado determinado cuando la variable tiempo tiende a infinito. Se puede demostrar que para las CTMC que permite especificar PRISM, estas probabilidades no dependen del estado inicial, lo que simplifica la tarea.

Sea $\underline{y}$ el vector de probabilidades estacionarias. Es decir, y_i representa la probabilidad de hallarse en el estado i-ésimo a largo plazo. No es difícil ver que este vector debe satisfacer las condiciones $\underline{y} \cdot \mathbf{Q} = \underline{0}$ y $\sum_i y_i = 1$. Al mismo tiempo, esto sugiere una forma de calcular los valores de $\underline{y}$:

1. Resolver el sistema de ecuaciones $\underline{y} \cdot \mathbf{Q} = \underline{0}$
2. Normalizamos la solución obtenida para que la suma de sus elementos sea 1, obteniendo $\underline{y} = \frac{y_0}{|y_0|}$.

2.4. Métodos iterativos de solución de sistemas de ecuaciones lineales

Como vimos en las secciones anteriores, el análisis de muchos operadores de PCTL y CSL son reducidas a resolver sistemas de ecuaciones lineales. Podemos asumir que este sistema de de la forma $\mathbf{A} \cdot \underline{x} = \underline{b}$, donde $\mathbf{A}$ es una matriz de valores reales, $\underline{b}$ es un vector de valores reales, y $\underline{x}$ es un vector de incógnitas el cual debe ser determinado. Ya que este sistema es derivado de los modelos a analizar, matrices y vectores son indexados por estados de S.

Se conocen muchas técnicas distintas para resolver sistemas de ecuaciones lineales. Por lo general, existen dos variantes para solucionar dichos sistemas : *método directo* o *método iterativo*. Los métodos directos calculan el valor exacto de la solución (dentro de los errores numéricos de representación). Ejemplos de estos algoritmos son *eliminación Gaussiana* o *descomposición* $L\backslash U$. Los métodos iterativos computan aproximaciones sucesivas de la solución, terminado cuando la secuencia de la solución converge a un valor o está dentro de cierto error de aproximación. Ejemplos de estos algoritmos son *método de potenciación*, *método de Jacobi* y *método de Gauss-Seidel*.

En nuestro trabajo solo vamos a considerar los métodos iterativos, esto se debe a que la verificación de modelos genera matrices $\mathbf{A}$ muy grandes y ralas. Los métodos directos no son adecuados para resolver este tipo de sistemas debido a la complejidad de la solución y a que modifican la matriz. Por suerte, los métodos iterativos no modifican la matriz, evitando el

aumento de tamaño de esta, y su complejidad es menor, haciendo que sean los métodos de solución de sistemas de ecuaciones lineales más adecuado en este caso.

Si bien PRISM implementa otros métodos iterativos de solución de sistema de ecuaciones lineales como método de potenciación o método de Gauss-Seidel, en este trabajo solo nos concentramos en la paralelización del método de Jacobi. Por ello no se darán en más detalles de los métodos restantes.

2.4.1. Método de Jacobi

El método de Jacobi se basa en la observación de que la i-ésima ecuación del sistema lineal de ecuaciones $\mathbf{A} \cdot \underline{x} = \underline{b}$:

$$\sum_{j=0}^{|S|-1} \mathbf{A}(i,j) \cdot \underline{x}(j) = \underline{b}(j)$$

puede ser reordenado como

$$\underline{x}(i) = \left(\underline{b}(i) - \sum_{j \neq i} \mathbf{A}(i,j) \cdot \underline{x}(j) \right) / \mathbf{A}(i,i)$$

Basándonos en esto, el i-ésimo elemento del vector de la k-ésima iteración es calculado a partir de los elementos del vector de la (k-1)-ésima iteración.

$$\underline{x}^{(k)}(i) = \left(\underline{b}(i) - \sum_{j \neq i} \mathbf{A}(i,j) \cdot \underline{x}^{(k-1)}(j) \right) / \mathbf{A}(i,i)$$

Es importante notar que los sistemas resultantes del análisis de PCTL o CSL, los elementos de la diagonal $\mathbf{A}(i,i)$ son siempre distintos de cero. Por ello, es útil expresar una iteración del método de Jacobi en términos de matrices y vectores.

$$\underline{x}^{(k)} = \mathbf{D}^{-1} \cdot ((\mathbf{L} + \mathbf{U}) \cdot \underline{x}^{(k-1)} + \underline{b})$$

donde $\mathbf{D} - (\mathbf{L} + \mathbf{U})$ es una partición de la matriz $\mathbf{A}$ en su diagonal, el triangulo superior y el triangulo inferior. Destacaremos que la operación principal en cada iteración es multiplicación de matriz-vector.

En la implementación del método de Jacobi es necesario elegir un valor inicial para el vector $\underline{x}^{(0)}$ y calcular la convergencia de la solución en cada iteración. Para los valores de $\underline{x}^{(0)}$, es común elegir valores aleatorios o algún valor estimado. Luego, cada iteración generará una vector solución más exácto al valor de la solución.

Para calcular la convergencia del sistema, recordemos que el vector solución de la iteración k-ésima es calculado a partir del vector solución de la iteración $(k-1)$-ésima. Así, el proceso

de iteraciones termina cuando el vector solución ha convergido lo suficiente. El criterio de convergencia puede variar, PRISM implementa dos criterios : *absoluto* y *relativo*.

En el criterio absoluto, se considera que el sistema converge cuando la diferencia máxima entre elementos de vectores solución consecutivos es menor a cierto valor ϵ :

$$\max_i |\underline{x}^{(k)}(i) - \underline{x}^{(k-1)}(i)| < \epsilon$$

Un problema de este criterio es que si el vector solución contiene valores menores a ϵ, el sistema puede terminar prematuramente.

En en criterio relativo, se considera que el sistema converge cuando la diferencia máxima relativa entre mismo elementos de vectores solución consecutivos es menor a cierto valor ϵ :

$$\max_i \left(\frac{\underline{x}^{(k)}(i) - \underline{x}^{(k-1)}(i)}{\underline{x}^{(k)}(i)} \right) < \epsilon$$

Es importante notar, además, que el método de Jacobi utiliza dos vectores solución en memoria, a diferencia de otros métodos que solo utilizan uno. Esto tiene algunas ventajas para el trabajo, como que la paralelización es más simple o factible y algunas desventajas como por ejemplo, que la cantidad de memoria necesaria es considerablemente mayor.

Capítulo 3

Modelo Computacional de arquitectura GPGPU

En este capítulo detallaremos porqué la arquitectura GPU se ha popularizado en el ámbito de la programación de alto desempeño. Comenzaremos con un poco de historia y como se fue evolucionando desde una arquitectura de propósito específico a una arquitectura completamente programable. Luego, detallaremos detenidamente la arquitectura y el modelos computacional, para finalizar explicando las técnicas utilizadas para lograr su máximo rendimiento.

3.1. ¿Por qué utilizar GPGPU?

El hardware de procesamiento de gráficos, conocidos generalmente como GPU por sus siglas en inglés (Graphics Processing Unit), es hoy en día probablemente el hardware computacional de mejor relación precio-rendimiento. Investigadores y desarrolladores de software se comenzaron a interesar en utilizar este poder computacional para resolver sus problemas, ámbito conocido como GPGPU (computación de propósito general en GPU).

Las arquitecturas más recientes de GPU proveen mucho ancho de banda y un gran poder computacional. Por ejemplo la Tesla K20X de NVIDIA posee un ancho de banda que ronda en 250 GB/seg comparado a los 50 GB/seg de una arquitectura CPU de múltiples núcleos, o un rendimiento pico teórico en doble precisión de 1,31 TFLOPS comparados a aproximadamente 170 GFLOPS de la arquitectura CPU más reciente (Sandy Bridge).

El hardware gráfico es poderoso y esta mejorando rápidamente. Las mediciones de rendimiento computacional (GFLOPS) hecha sobre las diferentes evoluciones de la arquitectura muestra un factor de crecimiento de 2 aproximadamente cada año. Este rango de crecimiento va más allá de la conocida Ley de Moore que prevee una factor de crecimiento de 1.4X cada año.

La diferencia en el crecimiento de rendimiento se explica por dos factores. En primer lugar la tecnología de fabricación de semiconductores avanza de igual modo para ambas arquitecturas, GPU y CPU. La disparidad puede ser atribuida al segundo factor, la diferencia en la arquitectura. Las CPU están optimizadas para procesamiento de alto desempeño en problemas secuenciales, utilizando gran cantidad de transistores para el paralelismo a nivel de instrucción. Para ello utiliza técnicas como predicción de bifurcaciones y ejecución fuera de orden. Por otro lado, la naturaleza del procesamiento de gráficos, permite utilizar más transistores directamente para computar, alcanzando un mayor rendimiento aritmético con la misma cantidad de transistores.

3.2. Desde una estructura fija a una programable

Las arquitecturas gráficas modernas se han convertido en una arquitectura flexible y poderosa, sin embargo no lo fueron así desde su comienzo. Las primeras arquitecturas GPU surgieron alrededor de los 80-90 con la aparición y popularización de los sistemas operativos con interface gráfica y los juegos en primera persona. Esto, con ayuda de la popularización de la computadora personal y las consolas de video juegos hicieron que empresas como NVIDIA, ATI technologies y 3dfx Interactive se interesaran en el mercado de las GPU.

El dominio de procesamiento iterativo de gráficos en 3D posee varias características que lo diferencian de las aplicaciones generales de CPU. En particular las aplicaciones 3D requieren de alto desempeño computacional y exhiben un paralelismo total. Por ello, las primeras placas constaban de etapas de trabajo (pipeline) fijas las cuales estaban bien definidas, no eran flexibles a su modificación o programación, permitiendo un mayor desempeño que el conseguido por los procesadores convencionales.

Con el paso de los años, la arquitectura GPU fue evolucionando para conseguir mejor rendimiento gráfico y dar más flexibilidad a las aplicaciones. Para ello, algunas de las etapas de procesamiento comenzaron a ser flexibles y programables por el usuario. Las capacidades de las placas eran utilizadas a través de bibliotecas gráficas como DirectX o OpenGL. Sabiendo del poder computacional de las GPU, investigadores y desarrolladores comenzaron a utilizarlas para solucionar sus problemas, los cuales no siempre pertenecían al dominio gráfico. Para lograrlo debían no solo adaptar sus problemas, moldearlos, para que se asemejaran a la renderización 3D sino también aprender la utilización de las bibliotecas que permitían su uso.

Cada nueva generación de GPU incrementaba su funcionalidad y la convertía en una arquitectura más programable. El paso más importante para permitir la programación de propósito general con GPU fue la introducción de un hardware completamente programable y un lenguaje ensamblador para especificar como el código corría en la GPU.

3.2.1. CUDA: Historia

En noviembre de 2006, NVIDIA lanzó su primera placa que seguía los estándares de DirectX 10 y que estaba construida bajo la arquitectura CUDA de NVIDIA. Esta arquitectura incluía nuevos componentes diseñados especialmente para la computación GPU y con el objetivo de eliminar muchas de las limitaciones que hacían más difícil el uso de las GPU para cómputos de propósito general.

Para alcanzar el número máximo posible de desarrolladores, NVIDIA tomó el estándar industrial de C++ y agregó un número relativamente pequeño de directivas para permitir la utilización de las características especiales de la arquitectura CUDA. Así, NVIDIA hizo público un compilador para el lenguaje CUDA C, convirtiéndolo en uno de los primeros lenguaje especialmente diseñado por una compañía para facilitar el cómputo de propósito general con unidades de procesamiento gráfico. NVIDIA desarrolló también drivers especiales para explotar el poder computacional de la arquitectura CUDA. De este modo, los desarrolladores no tuvieron que aprender más la utilización de bibliotecas gráficas como DirectX o OpenGL ni convertir sus problemas en algoritmos gráficos.

3.3. CUDA como modelo de programación escalable

Las arquitecturas de CPU de múltiples núcleos y GPU significaron que los chips de procesadores convencionales fueran sistemas paralelos. Más aún, su paralelismo continua escalando con la ley de Moore. El desafío es lograr construir aplicaciones que utilicen este paralelismo y que de forma transparente escalen para aprovechar el incremento en el número de núcleos. Tal como las aplicaciones de procesamiento gráfico 3D escalan su paralelismo a GPU de múltiples núcleos.

El modelo de programación paralelo de CUDA está diseñado para sobreponerse a este desafío mientras facilita el aprendizaje con la utilización del estándar de C.

Su modelo proporciona tres tipos de abstracciones: una jerarquía de grupos de hilos (threads), memoria compartida y barreras de sincronización. Estas son utilizadas por el programador a través de un número pequeño extensiones del lenguaje C.

Estas abstracciones proveen de paralelismo de datos fino y paralelismo de hilos, mezclado en medio de un paralelismo de datos grueso y paralelismo de tareas. Esto lleva al usuario a dividir el problema en subproblemas que puedan ser solucionados independientemente en paralelo por bloques de hilos, y cada problema en partes más pequeñas que puedan ser resueltos de forma cooperativa en paralelo por todas los hilos de un mismo bloque.

Esta descomposición preserva la expresividad del lenguaje, permitiendo a los hilos cooperar cuando solucionan cada subproblema, y al mismo tiempo permite la escalabilidad automáticamente. De este modo, cada bloque de hilos puede ser asignado a cualquiera de

los multiprocesadores disponibles en la GPU, en cualquier orden, de forma concurrente o secuencial, permitiendo que el código CUDA pueda ejecutarse en cualquier número de multiprocesadores y solo el sistema de planificación debe conocer la cantidad física de multiprocesadores, como se ilustra en la Figura 3.1.

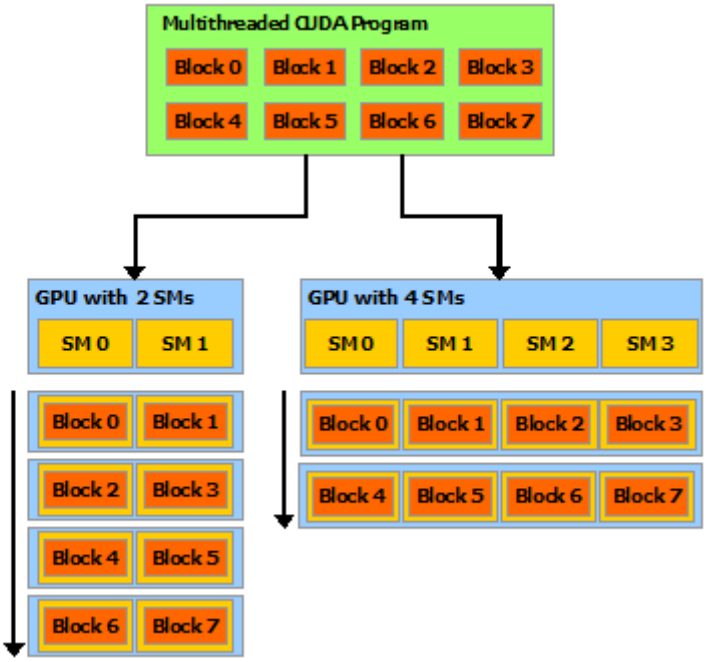


Figura 3.1: Escalabilidad Automática

3.3.1. Modelo de programación

En esta Sección presentaremos los conceptos principales el modelo de programación de CUDA C. CUDA C extiende el lenguaje estándar C, permitiendo al programador definir funciones, llamadas *Kernels*, que cuando son llamadas se ejecutan N veces en paralelo por N hilos de CUDA, a diferencia de sólo un hilo en una función regular de C. El programador es quién decide el valor dinámico o estático del parámetro N en el momento de ejecutar el kernel. A cada hilo que ejecuta un kernel se le asigna un identificador único el cual es accesible por el hilo dentro del kernel. Estos identificadores siguen los lineamientos de la jerarquía de hilos analizada a continuación.

Jerarquía de Hilos

Cada identificador de hilo puede ser visto como una 3-upla, por lo que cada hilo puede ser identificado utilizando un índice de una, dos o tres dimensiones, formando así un bloque

de hilos de una dos o tres dimensiones. Esto provee una forma natural de mapear los identificadores de hilos con el accesos a datos. Hay un límite en el número de hilos por bloque, ya que se espera que cada bloque de hilos resida en un mismo multiprocesador y debe compartir recursos de memoria limitados dentro del procesador. El número máximo de hilos por bloque es de 1024 en la arquitectura utilizada en este trabajo.

Los bloques son organizados en grillas de una, dos o tres dimensiones como se ilustra en la Figura 3.2. El número de bloques de hilos en una grilla está normalmente limitado directamente por le tamaño de los datos a procesar o el número de procesadores en el sistema.

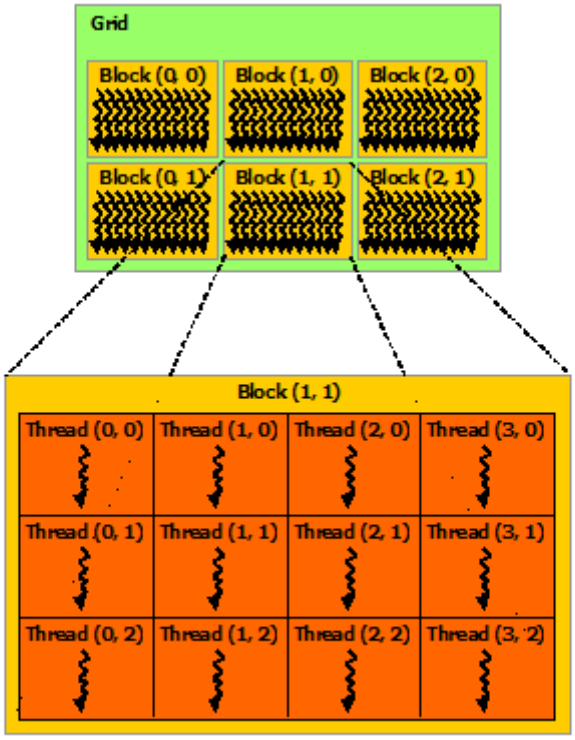


Figura 3.2: Grilla de bloques de hilos

Cada bloque dentro de una grilla puede ser identificado por un índice de una, dos o tres dimensiones según haya sido declarado y accesible dentro del kernel a través de una variable predefinida. Del mismo modo es accesible las dimensiones del bloque de hilos y de la grilla de bloques.

Los hilos dentro de un bloque pueden cooperar entre ellos compartiendo datos a través de memoria compartida y sincronizando su ejecución para coordinar el accesos a esta memoria. Para que la cooperación sea eficiente, se requiere que el acceso a memoria compartida tenga baja latencia y la sincronización no tenga una gran penalización.

Jerarquía de Memoria

Los hilos de CUDA pueden acceder a diferentes espacios de memoria durante su ejecución como se ilustra en la Figura 3.3. Cada hilo dispone de memoria local. Cada bloque de hilos dispone de memoria compartida visible por todos los hilos de un mismo bloque. Todos los hilos tienen acceso a la misma memoria global.

Hay adicionalmente dos memorias de sólo lectura accesible por todos los hilos: memoria constante y memoria de textura. La memoria global, memoria de textura y memoria constante están optimizadas para diferentes usos. La memoria de textura ofrece un modo de acceso y filtrado de datos para formatos de memoria específicos. No cubriremos este tipo de memoria ya que no es utilizada en el trabajo.

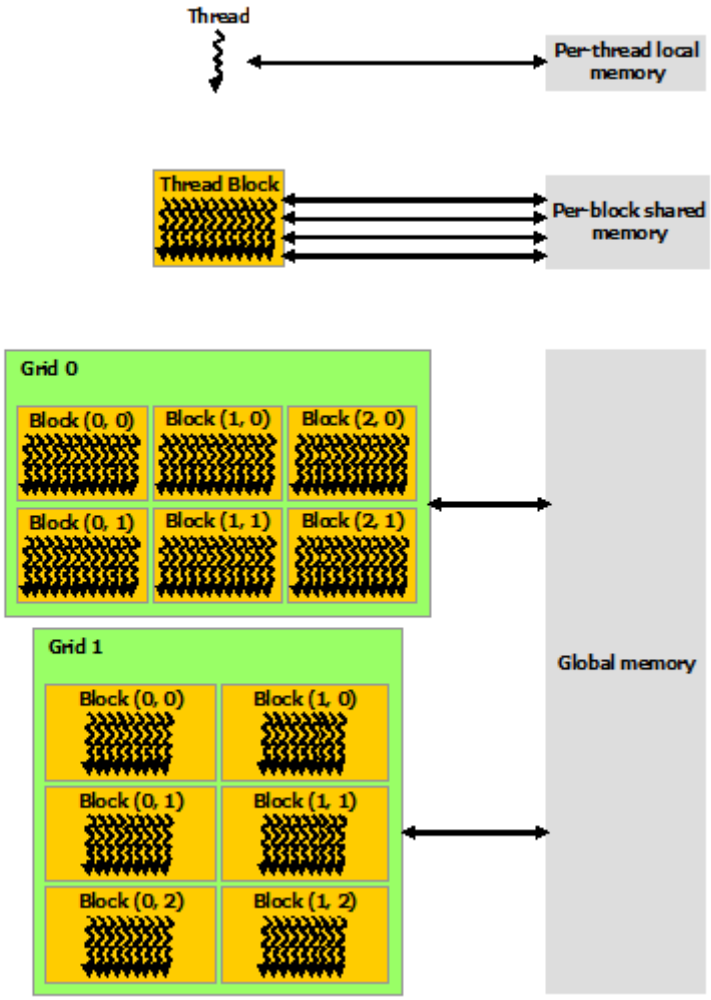


Figura 3.3: Jerarquía de memoria

Programación Heterogénea

El modelo de programación de CUDA asume que los hilos de CUDA ejecutan en un dispositivo físicamente separado que opera como un coprocesador del *host* que esta ejecutando la aplicación, por lo general y para el análisis de nuestro trabajo esta aplicación está escrita en C/C++ utilizando el lenguaje CUDA. El código kernel será ejecutado específicamente en la GPU y el resto del programa se ejecutará en el procesador central o CPU.

Además, el modelo computacional de CUDA asume que tanto el host como el dispositivo manejan distintos espacios de memoria, referidos como *memoria de host* y *memoria de dispositivo* respectivamente. CUDA provee una API completa para manejar la memoria de dispositivo y poder ser reservada, escrita y leída por el host.

3.4. Implementación del hardware CUDA

La arquitectura CUDA está construida alrededor de un arreglo de procesadores de múltiples hilos o *Streaming Multiprocessors (SMs)*. Cuando un programa CUDA está ejecutando en host e invoca la ejecución de una grilla de kernels, los bloques de la grilla son numerados y distribuidos a los multiprocesadores disponibles para su ejecución. Los hilos de un bloque ejecutan concurrentemente en un mismo multiprocesador, y múltiples bloques de hilos pueden ejecutar de forma concurrente en un mismo multiprocesador. A medida que los bloques de hilos terminan, nuevos bloques son asignados a los multiprocesadores vacantes.

Los multiprocesadores están diseñados para ejecutar cientos de hilos de forma concurrente. Para manejar este número de hilos, estos utilizan una arquitectura llamada *SIMT (Single Instruction, Multiple Thread)*

3.4.1. Arquitectura SIMT

Los multiprocesadores crean, manejan, planifican y ejecutan en paralelo grupos de 32 hilos, llamados *warps*. Los hilos de un warp comienzan juntos en el mismo punto del programa, pero cada uno tiene su propio contador de instrucciones y registros de estados y son libres de ejecutar independientemente.

Cuando un multiprocesador posee uno o más bloques de hilos para ejecutar, este parte los bloques en warps y cada warp es planificada por un planificador de warps para ser ejecutada. La forma en que los bloques son divididos en warps es siempre la misma; cada bloque contiene hilos con identificadores numéricos asignados de forma consecutiva. El primer warp contiene los hilos con identificadores 0 a 31, el segundo warp los hilos 32 a 63 y así sucesivamente.

Cada hilo dentro de un mismo warp ejecuta una misma instrucción al mismo tiempo, por lo tanto el rendimiento óptimo se consigue cuando los 32 hilos de un warp siguen el mismo camino de ejecución. Si los hilos de un warp divergen en el flujo de ejecución, las ejecución

de los hilos del warp son serializados, deshabilitando los hilos que no están en el flujo de ejecución y cuando los posibles caminos convergen, todos los hilos vuelven al mismo punto del programa. Esto sólo ocurre entre hilos de un mismo warp. Diferentes warps ejecutan independientemente.

Si analizamos la corrección del programa, el programador puede esencialmente ignorar el comportamiento de la arquitectura SIMT. Sin embargo, se pueden conseguir sustanciales resultados de mejoramiento de rendimiento teniendo en cuenta la forma en que los hilos son agrupados y cómo es el comportamiento de ellos en los warps. En la práctica, esto es análogo a cómo se comporta la cache. El tamaño de cache puede ser ignorado en la corrección del diseño, pero debe ser considerado en la estructura del código para conseguir el rendimiento máximo. La arquitectura SIMT requiere de ciertos cuidados al acceder a la memoria y manejar la divergencia de los hilos, que serán analizados más adelante.

3.4.2. Características del Hardware Multi-hilo

El contexto de ejecución de cada warp (contadores de programa, registros, etc.) es mantenido en la memoria interna de cada multiprocesador lo largo de la vida del warp. Esto implica que cambiar de un contexto de ejecución a otro no tiene costo. Esto es aprovechado para que los multiprocesadores mantengan un conjunto de warps activos para la ejecución y el planificador de ejecución del multiprocesador elije cual es la siguiente warp a ejecutar. El modo de manejar la ejecución de los warps es una gran ventaja en el diseño de la arquitectura, permitiendo ocultar de forma óptima la latencia de lectura y escritura a memoria, siempre y cuando el multiprocesador tenga suficientes warps disponibles para la ejecución.

En particular, cada multiprocesador contiene un conjunto de registros de 32 bits que son divididos a lo largo de los bloques y una cache de datos y memoria compartida que es dividida a lo largo de los bloques de hilos. El número de bloques y warps que pueden residir y ser procesados al mismo tiempo dentro de un multiprocesador para un programa dado, depende de la cantidad de registros y memoria compartida utilizada para el programa y la cantidad de registros y memoria compartida disponibles en el multiprocesador. También existe un número máximo de bloques residentes y de warps residentes en cada multiprocesador.

Para comprender este hecho, veremos un ejemplo concreto en la arquitectura específica utilizada en el trabajo. Hablamos de la arquitectura Kepler de NVIDIA. En esta arquitectura, el tamaño de los warps es 32 y cada multiprocesador posee 256 KB de memoria de registros y memoria compartida programable en 16, 32 o 48 KB. Suponemos que poseemos un kernel que utiliza 25 registros locales de 32 bits y cada bloque lanzado es de 256 hilos. Cada bloque necesita de $256 \times 25 \times 4 = 25KB$ lo cual nos indica que no puede haber más de 10 bloques simultáneamente en el mismo SM, de haberlos el multiprocesador se quedaría sin memoria local. Recordemos que cada hilo necesita que sus valores locales persistan en memoria local

a lo largo de su ejecución para permitir que el planificador los saque y ponga en ejecución rápidamente. Del mismo modo si los SM están configurados para tener 48KB de memoria compartida y cada bloque utiliza 12KB de esta memoria, no puede haber mas de 4 bloques simultáneamente en el mismo SM. De estos dos parámetros analizados para determinar, en tiempo de compilación, cuantos bloques pueden residir el cada SM, el mínimo entre ambos será el valor final.

De lo analizado anteriormente se desprende un valor de utilización de los multiprocesadores o *Occupancy* que es la diferencia entre la cantidad de bloques de un kernel en particular que puede manejar cada cada multiprocesador y la cantidad máxima de bloques determinados por la arquitectura. En el caso de la arquitectura Kepler el número máximo de bloques por SM es 16. Así, Occupancy es un valor entre 0 y 1. Mientras más cerca de 1 se encuentre, no significará que el código será más eficiente ya que esto depende de la combinación de muchos factores, pero determina cuan ocupados estarán los SM, permitiendo así mejorar el ocultamiento de latencia de accesos a memoria entre otras cosas.

3.5. Técnicas de Rendimiento

Para lograr conseguir el máximo de rendimiento de la arquitectura GPU es necesario adaptar el problema para seguir algunos lineamentos de la arquitectura. En nuestro problema trataremos de conseguir :

1. Maximizar la ejecución en paralelo para alcanzar la máxima utilización.

2. Optimizar el uso de la memoria para alcanzar el máximo ancho de banda.

Para lograr la máxima utilización debemos separar el problema en bloques lo más independientes posibles para que estos puedan ser mapeados a diferentes componentes del sistema y mantener estos componentes lo más ocupados posible. A nivel multiprocesador, como ya explicamos, es importante que haya muchas warps activas dispuestas a ejecutar para poder ocultar la latencia de acceso a memoria. Además, es necesario que los threads de un mismo warp minimicen las bifurcaciones y las sincronizaciones como barreras o mutex de escritura de memoria.

En cuanto a utilización de memoria, el primer paso es tratar de maximizar el rendimiento en el accesos a memoria global, es decir, memoria de bajo ancho de banda comparada a la memoria compartida. Las técnicas más utilizadas son diseñar el algoritmo para minimizar el acceso a memoria global y utilizar la memoria compartida como una cache intermedia entre la lectura - operación - escritura. El esquema básico sería :

1. Cargar los datos de memoria global a memoria local.

2. Sincronizar todos los threads del bloque de tal modo que cada thread pueda acceder a la memoria cargada por otro thread de forma segura.

3. Procesar los datos en memoria compartida.

4. Sincronizar nuevamente, si es necesario, para asegurar que todos los threads terminaron de procesar los datos.

5. Escribir los resultados nuevamente a memoria global.

Otro punto que mejora el rendimiento es seguir los patrones de accesos óptimos a memoria. Cada memoria tiene sus propias características.

La memoria global reside en memoria del dispositivo, esta memoria es accedida a través de transacciones de 16, 32 y 64 bytes. Dichas transacciones están alineadas. Cuando una warp ejecuta una instrucción que accede a memoria global, esta genera las cantidad de transacciones necesarias dependiendo del tamaño de dato accedido de tal manera de poder satisfacer cada hilo y luego lo distribuye entre ellos. Por lo general, mientras más transacciones sean necesarias, más datos innecesarios son transferidos al warp y luego desechados, empeorando el rendimiento. Por ello es importante que las instrucciones de acceso a memoria global sean hechas de tal forma que los datos necesarios por los hilos estén lo más juntos posible.

La memoria compartida reside en los multiprocesadores. Por ello el acceso a esta memoria tiene más baja latencia y más alto ancho de banda que la memoria global. Para maximizar el ancho de banda, la memoria compartida es dividida en módulos de igual tamaño, llamados bancos, los cuales pueden ser accedidos simultáneamente. Cualquier requerimiento de lectura o escritura realizado a n direcciones que caen en n bancos de memoria distintos pueden ser servidos simultáneamente. Del mismo modo, accesos simultáneos de varios hilos a posiciones distintas del mismo banco generan la serialización del acceso. Es importante destacar que si varios hilos acceden a la misma posición de memoria, el warp realiza una sola transacción y luego distribuye la información a todos los hilos que la requirieron.

La memoria constante y memoria de textura son memorias que residen en memoria de dispositivo, pero no analizaremos su patrón de acceso ya que este trabajo no hace uso de este tipo de memorias.

Capítulo 4

Paralelización

Este trabajo se enfoca en la paralelización del algoritmo iterativo de resolución de sistemas lineales de ecuaciones de Jacobi, explicado en el Capítulo 1. Como veremos a lo largo de este Capítulo, PRISM implementa el algoritmo de Jacobi con tres representaciones distintas de la matriz de transiciones y el vector de estados. En particular, nos centraremos en la representación híbrida de PRISM, ya que almacena la información de la matriz en una representación compacta. Es importante destacar que esta paralelización con arquitecturas GPGPU no ha sido estudiada anteriormente.

4.1. Algoritmo genérico del método de Jacobi

Como vimos en la sección 2.4, el método de Jacobi o método de relajado de Jacobi puede ser reducido a producto de matriz-vector, operaciones vector-vector y al cálculo de convergencia. Este método se basa en calcular el vector de estados $\underline{x}^{k+1}$ a través del vector de estado $\underline{x}^{k}$ en cada iteración hasta alcanzar la condición de convergencia. El algoritmo genérico de Jacobi para resolución del sistema lineal $\mathbf{A} \cdot \underline{x} = \underline{b}$ es presentado en el algoritmo 1, donde la descomposición de la matriz $\mathbf{A} = \mathbf{D} - (\mathbf{L} + \mathbf{U})$ fue la explicada en la Sección 2.4. Dependiendo de la estructura utilizada para representar las matrices y vectores, el algoritmo variará ligeramente.

4.2. Representación de Matrices y vectores en PRISM

PRISM implementa tres motores distintos de verificación :

- Motor explícito
- Motor simbólico (MTBDD)
- Motor híbrido

Jacobi($\mathbf{A}$, $\underline{x}$, $\underline{b}$)
$\{\mathbf{A} = \mathbf{D} - (\mathbf{L} + \mathbf{U})\}$ $\mathbf{M} := (\mathbf{L} + \mathbf{U})$ $\underline{soln} := \underline{x}$, $\underline{soln2}$ $done := false$ **do** $\neg done \rightarrow$ $\underline{soln2} = \mathbf{M} \times \underline{soln}$ $\underline{soln2} = \mathbf{D}^{-1} \cdot (\underline{soln2} + \underline{b})$ $done = Convergence(\underline{soln}, \underline{soln2})$ $swap(\underline{soln}, \underline{soln2})$ **od**
ret : $\underline{soln}$

Algoritmo 1: Método de Jacobi

Entre ellos se diferencian en la estructura de datos utilizada para representar las matrices y vectores en los diferentes algoritmos numéricos. Principalmente, entre las distintas estructuras de dato varía la cantidad de memoria necesaria para la representación y los tiempos de acceso a los elementos de la matriz o vector. Obviamente, con representaciones más compactas, que necesitan menos memoria, se pueden representar problemas de mayor tamaño y complejidad.

En el motor explícito, las matrices son representadas como matrices ralas y los vectores son representados como un arreglo de memoria consecutiva. Este motor es el que más memoria demanda de los tres, sin embargo, el acceso a los elementos de la matriz se adapta muy bien a los algoritmos iterativos de solución de sistemas lineales.

Por otro lado, el motor simbólico utiliza estructuras MTBDD, las cuales analizaremos a continuación, tanto para representar la matriz como para los vectores. Si bien este motor es el más eficiente en cuanto a utilización de memoria, la complejidad de los algoritmos numéricos es considerablemente mayor. Esto se debe, principalmente, a que el acceso a los elementos es logarítmico en el tamaño del vector o matriz. Es importante destacar que el acceso consecutivo de los elementos de la matriz o vector es lineal. Finalmente, el motor híbrido lleva este nombre porque combina las ventajas de los dos motores anteriores. En este motor, las matrices se representan utilizando estructuras MTBDD haciendo disminuir considerablemente la cantidad de memoria necesaria y los vectores son representados como un vector de memoria consecutiva. Esta representación y el hecho de cambiar el rango de la función booleana que representa el MTBDD a matrices (diremos que inyecta matrices ralas) hacen que este motor sea además muy eficiente en tiempo de ejecución.

4.2.1. Representación rala de matrices

Como ya explicamos, los modelos probabilistas considerados y cubiertos por esta tesis son descriptos a través de matrices de valores reales. Estas matrices son generalmente muy grandes, pero contienen un número relativamente pequeño de valores distintos a ceros. Por ejemplo, una matriz de transición de un modelo con 10^6 estados, puede tener tan poco como 10 entradas con elementos distintos a cero. En esta tesis, no se analizará, ni implementará alguna solución con matrices de transición en representación rala. Este tema, es bien conocido y ya existen implementaciones paralelas estudiadas a fondo. Sin embargo, la representación de la matriz en el motor híbrido, utiliza matrices inyectadas en representación rala como explicaremos en secciones siguientes.

Hay muchas maneras de representar matrices ralas. Todas ellas se basan en la idea de sólo representar de forma explícita las entradas con elementos distintos a cero. PRISM utiliza muchas representaciones distintas como filas comprimidas ralas (compressed sparse row, CSR) y columnas comprimidas ralas (compressed sparse column, CSC). Para llevar a cabo este trabajo, se ha agregado a PRISM la representación lista de coordenadas (COO).

La representación de matrices ralas COO es una lista de tuplas de la forma (fila, columna, valor). Por lo general la lista esta ordenada por fila y luego por columna y, así mejorar el acceso aleatorio. Cabe señalar que el espacio de memoria ocupado por una matriz en COO es proporcional a la cantidad de elementos no cero (nnz), es decir el espacio necesario para una matriz de nnz elementos distintos de cero sería, $((sizeof(int) \cdot 2 + sizeof(double)) \cdot nnz)$.

4.2.2. Diagrama de decisión binaria multi-terminal

La estuctura *Diagramas de decisión binaria multi-terminal* (MTBDD, por sus siglas en inglés), es la estructura de datos en la que se basa la tesis. Esta es una extensión de los Diagramas de decisión binaria (BDD), que fueron creados por Lee [Lee59] y Akers [Ake78] y luego aplicados por primera vez en verificación de modelos por McMillan, Clarke entre otros [BCM+92, McM93].

Un BDD es un grafo dirigido acíclico con raíz el cual representa una función booleana de la forma $f : \mathbb{B}^n \to \mathbb{B}$. Los MTBDD extienden los BDD a través de permitir la representación de funciones cuya imagen se extiende a cualquier conjunto D, es decir $f : \mathbb{B}^n \to D$. En la mayoría de los casos, D es tomado como $\mathbb{R}$ y también lo haremos aquí. Notar que BDD es un caso especial de MTBDD.

Sea $\{x_1, x_2, \ldots, x_n\}$ un conjunto de variables booleanas distintas, totalmente ordenadas de la siguiente forma : $x_1 < x_2 < \ldots < x_n$. Un MTBDD $\mathbf{M}$ sobre $\underline{x} = \{x_1, x_2, \ldots, x_n\}$, es un grafo dirigido acíclico el cual tiene un elemento raíz. Los vértices del grafo son llamados nodos. Cada nodo del grafo corresponde a una de dos categorías : *no-terminal* o *terminal*. Un nodo no-terminal $\mathbf{m}$ es etiquetado con una variable $var(\mathbf{m}) \in \underline{x}$ y tiene exactamente dos

hijos, denotados *then* y *else*. Un nodo terminal $\mathbf{m}$ es etiquetado con un valor real y no tiene hijos.

El orden sobre las variables booleanas es impuesto a los nodos del MTBDD. Para dos nodos no terminales, $\mathbf{m}_1$ y $\mathbf{m}_2$, si $var(\mathbf{m}_1) < var(\mathbf{m}_2)$ entonces $\mathbf{m}_1 < \mathbf{m}_2$. La estructura tiene la invariante que para cualquier nodo no-terminal $\mathbf{m}$ del MTBDD, $\mathbf{m} < then(\mathbf{m})$ y $\mathbf{m} < else(\mathbf{m})$.

La Figura 4.1, muestra un ejemplo de un MTBDD. Los nodos están ordenados en niveles horizontales, uno por cada variable booleana. Los dos hijos del nodo $\mathbf{m}$ están conectados por aristas al nodo $\mathbf{m}$, una línea sólida al hijo $then(\mathbf{m})$ y una línea punteada al hijo $else(\mathbf{m})$. Los nodos terminales son dibujados con cuadrados a diferencia de los nodos no-terminales. Por claridad, se ha omitido el terminal con el valor 0 y cualquier arista que vaya a este.

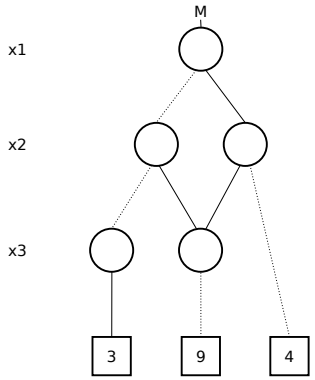


Figura 4.1: Ejemplo de MTBDD M

Como ya explicamos, un MTBDD $\mathbf{M}$ sobre variables booleanas $\underline{x} = (x_1, \ldots, x_n)$ representa una función $f_M(x_1, \ldots, x_n) : \mathbb{B}^n \to \mathbb{R}$. El valor de $f_M(x_1, \ldots, x_n)$ para una cierta asignación de las variables es calculado recorriendo el camino desde la raíz hasta un nodo terminal, por cada nodo no-terminal $\mathbf{m}$ tomando la arista $then(\mathbf{m})$ si $var(\mathbf{m})$ es 1, o tomando la arista $else(\mathbf{m})$ si $var(\mathbf{m})$ es 0. Por ejemplo, en el MTBDD de la Figura 4.1, $f_M(0, 1, 0) = 9$. Notar que por la naturaleza recursiva de la estructura, cada nodo del MTBDD es un MTBDD por si mismo.

La razón por la cual los MTBDD pueden, dependiendo del caso, ahorrar mucho espacio es porque pueden ser representados en una forma reducida. Para ello se utilizan dos reglas.

1. si los nodos $\mathbf{m}$ y $\mathbf{m}'$ son idénticos, es decir $var(\mathbf{m}) = var(\mathbf{m}')$, $then(\mathbf{m}) = then(\mathbf{m}')$ y $else(\mathbf{m}) = else(\mathbf{m}')$, sólo una copia del nodo es guardada.

2. Si un nodo $\mathbf{m}$ satisface $then(\mathbf{m}) = else(\mathbf{m})$, este es eliminado y cada nodo entrante a $\mathbf{m}$ es conectado con el único hijo de $\mathbf{m}$.

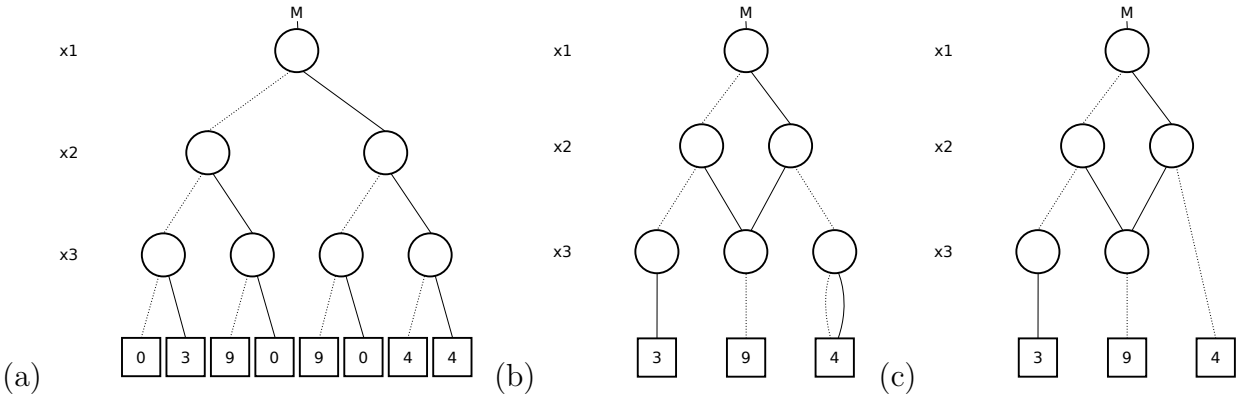


Figura 4.2: Reducción de un MTBDD M

En la Figura 4.2 (a) se presenta un MTBDD completo para una función $f(x_1, x_2, x_3) \to \mathbf{Z}$. En la Figura 4.2 (b), se encuentra el MTBDD para la misma función f pero luego de eliminar las aristas que llegan a 0 y aplicar la Regla 1 en los terminales 4 y en los terminales 9 y los nodos que apuntan a estos. Finalmente, en la Figura 4.2 (c) se representa el MTBDD de la misma función f luego de aplicar la Regla 2 sobre el nodo del nivel x_3 que apunta al terminal 4.

Para un orden fijo de variables booleanas y asumiendo que el MTBDD está completamente reducido se puede demostrar que la estructura de datos es canónica. Es decir, bajo estas asunciones, dada una función f_M hay una sola estructura que la representa. Esto tiene gran impacto en el rendimiento de la implementación de la estructura de datos. La Figura 4.2 (c) representa la estructura canónica del MTBDD para la función f y el orden de variables x_1, x_2, x_3.

Otro aspecto importante a considerar es el orden de las variables $\underline{x}$. El tamaño de un MTBDD (i.e. número de nodos) representado una función específica, es extremadamente dependiente del orden de las variables booleanas. Esto tiene un gran impacto tanto en el espacio necesario para el almacenamiento como el tiempo requerido para los algoritmos que involucran esta estructura de datos.

Matrices y vectores

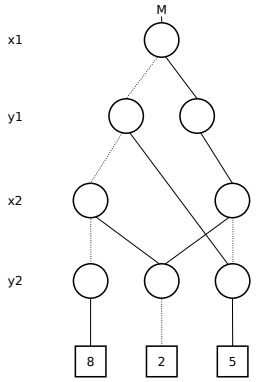
Desde el punto de vista de esta tesis, una de las aplicaciones más interesantes de los MTBDD es la representación de matrices y vectores. Esto fue investigado en el trabajo original que proponía la estructura de datos [FMY97, BFG+97]. Considere un vector de valores reales $\underline{v}$ de largo 2^n. Podemos pensar a $\underline{v}$ como un mapeo entre los índices del vector a los reales, e.i. $\underline{v} : \{0, \ldots, 2^n - 1\} \to \mathbb{R}$. De este modo, si tomamos un mapeo entre los índices $\{0, \ldots, 2^n - 1\}$ a n variables booleanas, podemos representar a $\underline{v}$ como un MTBDD $\mathbf{v}$ sobre las variables $\underline{x} = \{x_1, \ldots, x_n\}$ de tal modo que $f_v(\underline{x} = rep(i)) = \underline{v}(i)$ para todo $0 \leq i < 2^n$.

Esta idea se puede extender fácilmente a matrices. Podemos pensar en una matriz de 2^n por 2^n, $\mathbb{M}$, como un mapeo entre $\{0,\ldots,2^n-1\} \times \{0,\ldots,2^n-1\}$ a los $\mathbb{R}$. Nuevamente asumiendo una representación de los índices a $\mathbb{B}^n$, podemos representar la matriz $\mathbb{M}$ a través de un MTBDD sobre $2n$ variables booleanas, n de las cuales se utilizan para representar las filas y las n restantes para representar a las columnas. Usando las variables de representación de filas $\underline{x} = \{x_1,\ldots,x_n\}$ y las variables de representación de columnas $\underline{y} = \{y_1,\ldots,y_n\}$, diremos que el MTBDD $\mathbf{M}$ representa a una matriz $\mathcal{M}$ si $f_M(\underline{x} = rep(i), \underline{y} = rep(j)) = \mathcal{M}(i,j)$ para $0 \leq i,j < 2^n$. Notar que también podemos representar a vectores y matrices de tamaño diferente a una potencia de 2 simplemente agregando elementos 0 hasta llegar a un tamaño que sea potencia de 2. Esto no influirá en el tamaño final de MTBDD.

La figura 4.3, muestra la representación de una matriz de 4x4 a través de un MTBDD. Se utilizan variables de fila (x_1, x_2) y variables de columnas (y_1, y_2). La representación de los índices es la representación común de números enteros en booleanos.

Como remarcamos anteriormente, el tamaño final de la estructura depende fuertemente de la elección en el orden de las variables. Para el caso de la representación de matrices, una buena heurística es intercalar las variables de fila y columnas, generando un grafo MTBDD compacto según las pruebas y una forma simple de relacionar el recorrido de la estructura con la partición de la matriz. Es importante destacar que no existe un algoritmo polinómico para encontrar el orden de variables que genera la representación más compacta.

$$\mathbf{M} = \begin{pmatrix} 0 & 8 & 0 & 5 \\ 2 & 0 & 0 & 5 \\ 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 \end{pmatrix}$$



x_1	y_1	x_2	y_2	f_M	Entrada en M
0	0	0	1	8	(0,1) = 8
0	0	1	0	2	(1,0) = 2
0	1	0	1	5	(0,3) = 5
0	1	1	1	5	(1,3) = 5
1	1	1	0	2	(3,2) = 2
1	1	0	1	5	(2,3) = 5

Figura 4.3: Matriz M y su representación en MTBDD

4.3. Motor Híbrido

Esta sección explica las modificaciones propuestas en [Par02a] a la representación MTBDD para mejorar el rendimiento de los algoritmos que la manipulan.

En el motor híbrido se representa la matriz como una estructura MTBDD, obteniendo los beneficios de representación compacta explicada anteriormente. El vector de iteración será representado con un arreglo continuo de memoria. Esto permite el acceso a los elementos

de forma directa mejorando el rendimiento del producto matriz vector. Así, el problema al cual se enfrenta este modelo es como implementar la multiplicación matriz-vector utilizando estas dos estructuras fundamentalmente diferentes.

4.3.1. Primer Algoritmo

El primer intento es emular el algoritmo de multiplicación matriz-vector para matrices en representación explícita. Este multiplica una matriz **A**, almacenada en un vector de tuplas (fila, columna, valor), por un arreglo $\underline{soln}$ y guardar el resultado en otro arreglo $\underline{soln2}$. Destacaremos que en este algoritmo descripto, *cada elemento de la matriz es visitado sólo una vez.*

En el algoritmo de multiplicación matriz-vector con representación explícita, los elementos pueden ser obtenidos simplemente leyendo a lo largo del vector de tuplas. Las entradas de la matriz son obtenidas en orden, fila tras fila, aunque no sea requerido en el algoritmo. El mismo resultado puede ser obtenido estableciendo cada elemento de $soln2$ a 0, y luego realizando la operación $soln2[r] := soln2[r] + v \times soln[c]$ para todas las entradas de la matriz (r, c, v) en **cualquier orden**.

Con esto en mente, podemos pensar un algoritmo de multiplicación utilizando la representación MTBDD de la matriz de transiciones. Todas las entradas distintas a cero de la matriz en MTBDD pueden ser extraídas a través de un algoritmo de recorrido de grafos DFS (depth-first search) a lo largo de la estructura. Las entradas no serán extraídas en un orden particular, pero este es un método rápido para extraer todos los elementos. A modo de mayor compresión del algoritmo, podemos aclarar que cada camino distinto desde la raíz del MTBDD hacia un nodo terminal, distinto del nodo cero, es una entrada de la matriz que representa la estructura.

La parte clave del algoritmo es el recorrido de todos los caminos y cálculo de los índices de fila y columna. En un MTBDD, cada índice esta asociado con una codificación booleana, i.e. un elemento de $\mathbb{B}^n$. Asumiremos que las filas y las columnas de la matriz están indexados desde 0 hasta $2^n - 1$ y son codificados usando la representación booleana estándar para valores enteros. Estos valores pueden ser calculados independientemente acumulando valores a lo largo del camino hacia un nodo terminal. Es decir, supongamos que tomamos un camino, correspondiente a una entrada de la matriz, desde la raíz hasta un nodo terminal. El índice *fila* de esta entrada de la matriz se consigue sumando la potencia de dos correspondiente a cada nivel cada vez que se toma un camino *then* de un nodo que representa la variable *fila*. Igualmente con el índice columna y los nodos que representan variables columna.

Pero este algoritmo todavía tiene un problema. Los MTBDD representan matrices que pueden tener filas y columnas completas con ceros. Estos ceros corresponden a estados inalcanzables en el modelo. El número de estas filas y columnas completas con ceros puede

ser potencialmente grande y no aportan significado alguno al cálculo, ya que las entradas del vector resultado de la multiplicación permanecerá en cero. Incluir estas filas y columnas hace que estos vectores de iteración tengan más elementos que los realmente utilizados. Recordemos que el problema de memoria es una de las grandes limitaciones en la verificación de modelos.

4.3.2. Método mejorado

Construiremos un nuevo MTBDD a partir del viejo, en el cual a cada nodo no terminal se le agrega un offset. Para un nodo fila, este offset indicará la cantidad de filas con algún elemento distinto de cero en la submatriz superior, y para un nodo columna, este indicará la cantidad de columnas con un elemento distinto de cero en la submatriz izquierda. Esto significa que, cuando recorremos un camino desde la raíz a un nodo terminal, que representa una entrada de la matriz, los valores de los índices filas y columnas de esta entrada pueden ser determinados sumando los offsets etiquetados con variables filas o variables columnas, respectivamente, desde los cuales se toma el camino *then*.

Describiremos entonces una nueva estructura MTBDD a la que llamaremos *offset-labelled MTBDD*. Esta es esencialmente un MTBDD pero con dos diferencias importantes. Primero, como ya describimos, cada nodo no terminal es etiquetado con un valor entero, offset, el cual será utilizado para calcular los índices fila y columna. Segundo, el MTBDD no necesita estar completamente reducido. En realidad, este requerimiento es un poco más complicado y lo explicaremos a continuación.

Como vimos en la estructura MTBDD, existen dos tipos de reducciones permitidas que minimizan el tamaño del MTBDD : combinar nodos compartidos, i.e. aquellos nodos del mismo nivel y con hijos idénticos; y remover los nodos para los cuales la arista *then* y *else* apuntan hacia el mismo nodo, introduciendo uno o más saltos de nivel. En un offset-labelled MTBDD, el segundo tipo de reducción es sólo permitida cuando tanto la arista *then* como la *else* apuntan al nodo cero. Esto significa que cada arista de cada nodo apunta a un nodo del nivel inmediatamente inferior o directamente al nodo cero. Como consecuencia, cada camino desde la raíz a un nodo terminal distinto de cero, pasa por todos los niveles del offset-labelled MTBDD.

En la Figura 4.4 aparece una matriz con filas y columnas en 0, aquí representada con x, y su representación con un offset-labelled MTBDD. Los números en los nodos no terminales representan el offset, explicado anteriormente, correspondiente a dicho nodo.

Existen varias razones para realizar esto, pero la que más nos interesa es que necesitamos el valor offset que será utilizado para calcular los índices fila y columna en cada nivel de la estructura, y este esta almacenado en los nodos no terminales. Offset-labelled MTBDD no son estructuras canónicas, sin embargo no estamos interesados en este hecho. La estructura

$$\mathbf{M} = \begin{pmatrix} 0 & 3 & x & 6 \\ 0 & 3 & x & 4 \\ x & x & x & x \\ 5 & 0 & x & 4 \end{pmatrix}$$

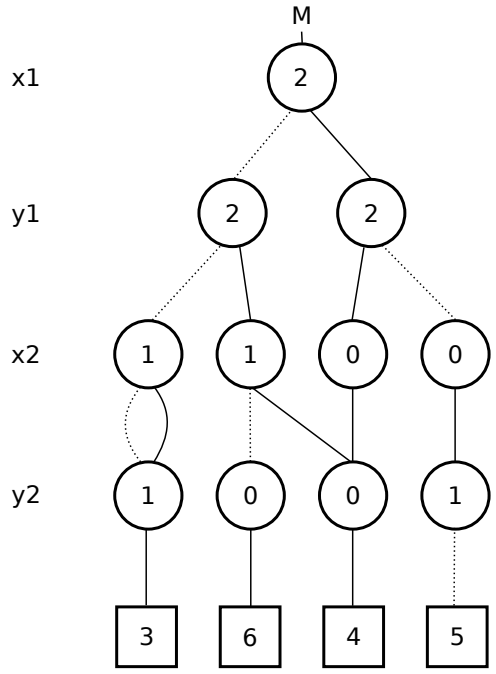


Figura 4.4: Matriz M y su representación en Offset-labelled MTBDD

será construida sólo una vez y utilizada para realizar los cálculos numéricos recorriéndola y luego será desechada, no es necesario manipularla.

En esta sección estamos interesados en comprender la estructura, por lo cual no analizaremos el algoritmo para construirla. Sin embargo vamos a destacar que la complejidad del algoritmo es proporcional a la cantidad de nodos del MTBDD original y la cantidad de nodos creados a lo largo del proceso. Por esta razón, el tiempo para construir el offset-labelled MTBDD debería ser generalmente menos que una iteración del proceso de solución. Además, y lo que más nos concierne, la cantidad de memoria adicional requerida para almacenar la matriz no incrementa significativamente. Estos resultados son analizados detalladamente en [Par02a].

4.3.3. Una vuelta más de tuerca

Ahora presentaremos una nueva mejora propuesta en [Par02a] al intento anterior, la cual resulta en un significante aumento de rendimiento en el algoritmo numérico. Los MTBDD son estructuras recursivas, es decir que cada nodo puede ser pensado como una sub-matriz de la matriz original. Entonces, los nodos compartidos de la estructura representan sub-matrices que se repiten. Por otro lado, cada iteración del algoritmo numérico requiere acceso a cada entrada de la matriz (dos elementos distintos de la matriz pueden estar representado por un mismo nodo del MTBDD) sólo una vez. En algoritmos explicados en las secciones anteriores, estas entradas son alcanzadas atravesando la estructura MTBDD u offset-labelled MTBDD. No es sorprendente que la implementación del motor explícito, en donde las entradas de la

matriz pueden ser leídas directamente del arreglo que las almacenan, es más rápido.

Para mejorar el algoritmo, la estructura offset-labelled MTBDD y el algoritmo para recorrerla son modificados para explotar la regularidad de la estructura en términos de tiempo y espacio. La idea clave es que, aunque los índices filas y columnas de las entradas de cada sub-matriz repetida son distintos en cada ocurrencia de dicha sub-matriz en la matriz original, los índices locales, relativos a la sub-matriz son los mismos. Esto es crucial para que funcione el algoritmo recursivo de recorrido de la estructura que computa los índices.

La optimización propuesta en [Par02a] es, para algún conjunto de nodos en el MTBDD, mantener una copia de la versión explicita de la sub-matriz correspondiente a esos nodos. Luego, reutilizar estas copias cada ves que el nodo es visitado en lugar de seguir con el algoritmo que atraviesa de ese nodo hacia abajo de la estructura. Para calcular los valores reales de los índices de las sub-matrices en cada visita, sólo es necesario agregarle a los índices locales de la sub-matriz los valores de los índices fila y columnas calculados en el recorrido desde la raíz hasta llegar al nodo que guarda la copia de la sub-matriz explícita. Las sub-matrices serán llamadas *matrices inyectadas* y se colocaran en todos los nodos de un nivel específico del MTBDD. Este nivel es calculado según la cantidad de memoria necesaria para guardar la copia de las matrices inyectadas.

Esta mejora tiene un impacto prácticamente despreciable en utilización de memoria, y muestra aumentos de rendimiento sorprendentes. Más resultados de esta mejora pueden ser vistos en [Par02a].

En la Figura 4.5 podemos ver el offset-labelled MTBDD correspondiente a la misma matriz presentada en la Figura 4.4. Las submatrices están inyectadas en el nivel 3, lo que sería equivalente a cambiar la signatura de la función que representa la matriz a $f(x_1, y_1) \rightarrow \mathbb{R}_{2\times 2}$

$$\mathbf{M} = \begin{pmatrix} 0 & 3 & x & 6 \\ 0 & 3 & x & 4 \\ x & x & x & x \\ 5 & 0 & x & 4 \end{pmatrix}$$

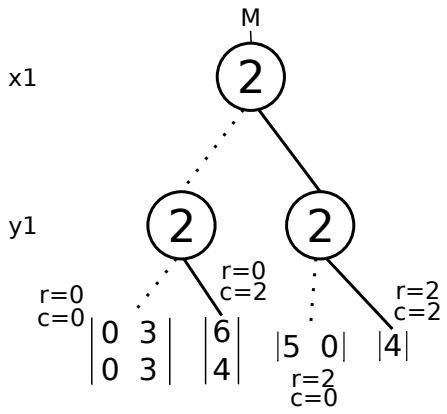


Figura 4.5: Matriz M y su representación en offset-labelled MTBDD

4.4. Paralelización en arquitectura GPGPU

Para lograr conseguir el máximo de rendimiento de la arquitectura GPU es necesario adaptar el problema para seguir algunos lineamentos de la arquitectura. En nuestro problema trataremos de conseguir :

1. Maximizar la ejecución en paralelo para alcanzar la máxima utilización
2. Optimizar el uso de la memoria para alcanzar el máximo ancho de banda

Para lograr la máxima utilización debemos separar el problema en bloques lo más independientes posibles para que estos puedan ser mapeados a diferentes componentes del sistema y mantener estos componentes lo más ocupados posible. A nivel multiprocesador, como ya explicamos en la sección de GPGPU, es importante que haya muchos warps activas dispuestas a ejecutar para poder ocultar la latencia de acceso a memoria. Además, es necesario que los hilos de un mismo warp minimicen las bifurcaciones y las sincronización como barreras o mutex de escritura de memoria.

En cuanto a utilización de memoria, el primer paso es tratar de maximizar el rendimiento en el accesos a memoria de bajo ancho de banda, i.e. memoria que reside en el dispositivo. Las técnicas más utilizadas son diseñar el algoritmo para minimizar el acceso a memoria global y utilizar la memoria compartida como una cache intermedia entre la lectura - operación - escritura. El esquema básico sería :

1. Cargar los datos de memoria global a memoria local.
2. Sincronizar todos los hilos del bloque de tal modo que cada hilo pueda acceder a la memoria cargada por otra hilo de forma segura.
3. Procesar los datos en memoria compartida.
4. Sincronizar nuevamente, si es necesario, para asegurar que todos los hilos terminaron de procesar los datos.
5. Escribir los resultados nuevamente a memoria global.

Otro punto que mejora el rendimiento es seguir los patrones de accesos óptimos a memoria. Como ya explicamos en el capítulo de GPU, cada memoria tiene sus propias características. En cuanto al acceso a memoria global, el conjunto de datos accedido por los hilos de una misma warp deben estar lo más juntos posibles, alineados en segmentos de 32, 64 o 128 bytes. Recordemos que la warp hace el pedido de los segmentos necesarios para cumplir con el requerimientos de sus hilos y luego difunde los resultados entre ellos. Entonces en una instrucción de lectura/escritura deberíamos minimizar el número de segmentos requeridos. La

memoria compartida es memoria de alta velocidad que esta dividida y alineada en bancos los cuales pueden ser accedidos en forma concurrente. Por ello debemos diseñar el acceso para que utilice concurrentemente la mayor cantidad de bancos posibles. Conflictos en el accesos de dos hilos de la misma warp en el mismo banco genera la serialización del acceso.

Tener en cuenta todas estas técnicas altamente dependientes de la arquitectura mejora sustanciales el rendimiento final del algoritmo. Pero hay que tener en cuenta que el problema es el que termina determinando como aplicarlas. Por ello, descompondremos el algoritmo numérico en dos problemas que serán tratados de forma muy diferente para adaptarlos a la arquitectura. Estos son :

- multiplicación matriz-vector.
- multiplicación vector-vector, suma vector-vector, cálculo de convergencia (reducción).

4.4.1. Adaptación de la estructura de representación MTBDD

La paralelización se enfoca en analizar la estructura de datos MTBDD y tratar de adaptarla para que la representación de la matriz de transiciones siga los requerimientos de rendimiento de las arquitecturas GPU. Para ello, se analizarán una serie de intentos realizados a lo largo del trabajo hasta llegar a la representación utilizada en el algoritmo final.

4.4.1.1. Primera Representación

Si analizamos a grandes rasgos el problema inicial, tenemos que realizar el algoritmo de multiplicación de una estructura de grafo dirigido acíclico simbólico por un arreglo consecutivo de memoria. A simple vista podemos imaginar que tenemos dos problemas los cuales atentan con los puntos de optimización que vimos anteriormente. En primer lugar la estructura esta formada por nodos residentes en posiciones de memorias arbitrarias, i.e., los datos no están compactos en memoria. En segundo lugar, al ser un grafo no parece simple encontrar una forma de dividir el problema en subproblemas independientes.

Como primer paso, tomaremos la estructura de offset-labelled MTBDD con matrices inyectadas y trataremos de modificar su representación. Esta modificación tiene como objetivos principales ajustar el problema para que se adapte a la arquitectura y no generar un gran impacto en utilización de memoria adicional.

Analizando el algoritmo de multiplicación matriz-vector, este realiza una suma/multiplicación por cada elemento no-cero de la matriz (r, c, v) (una entrada de la matriz en representación COO) de la siguiente manera :

$$res[r] = res[r] + b[c] * v$$

y es importante que sin importar el orden en que se procese cada uno de los elementos no-cero. También recordaremos que la representación de la matriz de transiciones se mantiene igual a lo largo del método de reducción. En la estructura de representación de matrices con offset-labelled MTBDD sin matrices inyectadas, cada tupla (v, f, c) se consigue recorriendo un camino distinto desde la raíz a un nodo terminal. El valor v es el valor del nodo terminal y los valores f y c se calculan estableciendo ambos valores a cero cuando se comienza a recorrer el camino desde la raíz y si tomamos un camino *then* en un nodo de nivel fila se suma el offset a f y si lo mismo pasa en un nodo de nivel columna se suma el offset a c. Este procedimiento se puede hacer para todos los caminos a la vez utilizando un algoritmo DFS en tiempo proporcional a la cantidad de caminos (valores no cero) de la matriz de representación por la cantidad de niveles del MTBDD.

Para la estructura offset-labelled MTBDD con matrices inyectadas, el procedimiento es similar, sólo que en lugar de llegar a un nodo terminal con los valores absolutos de f y c, se para cuando se alcanza el nivel de las matrices inyectadas con valores de f y c relativos. Luego cada tupla (v, f, c) es el resultado de sumarle a cada elemento (v, f', c') de las matrices inyectada los valores relativos de f y c calculados a lo largo del recorrido del camino.

Volviendo a la paralelización, se busca dividir el problema de multiplicación en problemas independientes con patrones compactos de accesos a memoria. Podemos ver que cada recorrido distinto desde la raíz hasta una matriz inyectada puede ser tratado como un problema independiente. Por consiguiente, hay que analizar como representar cada uno de estos problemas en memorias sin generar un gran impacto en espacio.

Como solución se optó representar cada problema (camino distinto) como una tupla $(fila, columna, i\_sm)$ donde $fila$ y $columna$ son los offset de los índices correspondiente hasta alcanzar la matriz inyectada e i_sm es la referencia a la matriz inyectada. Esta es la información mínima para representar cada sub-problema independientemente. De este modo, para la representación de la matriz de transiciones no se utilizará la estructura MTBDD, la cual tiene elementos en memoria fragmentada, sino el conjunto de tuplas propuesto por cada camino distinto y el conjunto de matrices inyectadas en representación COO. Para que esta representación sea viable, tenemos que asegurar que la memoria necesaria para almacenar esta información, a los cuales llamaremos instancias, se mantenga cerca de los valores utilizados por el motor híbrido.

Recordando de nuevo la estructura offset-labelled MTBDD, en esta estructura se inyectan matrices en todos los nodos de un nivel específico dependiendo del consumo en memoria que genera este proceso. Por ejemplo, si inyectamos las matrices en el primer nivel, el cual tiene sólo el nodo raíz, podremos general sólo una matriz, la cual es igual a la matriz completa de transiciones, obteniendo el mismo resultado que el motor explícito. En este caso, sólo tenemos un camino posible desde la raíz hasta las matrices inyectadas. De otro modo, si inyectamos las matrices en el último nivel, sólo obtendremos matrices de dimensión 1×1 obteniendo

la misma representación que el motor híbrido sin matrices inyectadas. Así, la cantidad de caminos desde la raíz hasta las matrices inyectadas es igual a la cantidad de entradas en la matriz de transiciones, es decir un número muy grande que escala muy rápido con el tamaño del problema. En estos dos casos, la cantidad de memoria necesaria es proporcional a la utilizada por el motor explícito, lo cual no sirve para nuestro problema.

La pregunta que debemos responder entonces es si existe algún nivel en el que la memoria necesaria para guardar las matrices inyectadas y las tuplas de cada instancia de subproblema se mantiene dentro de cantidades razonables.

Cálculo de consumo de memoria

Dado un nivel específico de la estructura MTBDD, debemos sumar la cantidad de memoria necesaria para representar cada matriz inyectada y la memoria necesaria para representar cada tupla de instancia. En ambos casos se resume a contar caminos en el MTBDD.

En el primero, los caminos desde cada nodo del nivel elegido hasta los nodos terminales. Cada uno de estos caminos representa un elemento no cero en la submatriz que representa ese nodo, lo cual nos da la cantidad de memoria de la representación en COO. Esto se hace a través de un DFS sobre el grafo del MTBDD, guardando la cantidad de elementos no cero desde los nodos más profundos hacia los nodos superficiales. Es decir, la cantidad de elementos no cero (nnz) de un nodo terminal es 1 y la cantidad de elementos nnz de un nodo no terminal es la suma de la cantidad de elementos nnz de sus hijos. Este algoritmo puede ser descripto como una recursión con memorización.

En el segundo caso debemos contar los caminos distintos desde la raíz hasta los nodos del nivel elegido para inyectar matrices. La cantidad de memoria será proporcional a la cantidad de caminos por la cantidad de memoria utilizada para representar cada tupla de instancia. El algoritmo para realizar este cálculo es análogo al anterior sólo que en la recursión comenzamos de la raíz y frenamos en los nodos del nivel elegido, devolviendo en este lugar 1, correspondiente a que hay un sólo camino desde él hasta el mismo.

Al realizar estos cálculos sobre diferentes problemas de verificación de modelos, se puede ver que la naturaleza de los problemas a chequear hace que la representación en MTBDD sea muy compacta y que la memoria necesaria para la nueva representación de la matriz de transiciones grafique una función cuasi-campana con un punto de mínimo consumo de memoria. En este punto, la cantidad de memoria se mantiene en niveles muy cercanos a la cantidad de memoria necesaria para la representación con offset-labelled MTBDD con matrices inyectadas. No analizaremos en detalle la memoria necesaria para esta representación, porque como ya veremos no es la representación que se adapta de mejor modo a nuestro problema.

Esta representación permite razonar mejor sobre los sub-problemas y el algoritmo de

multiplicación. Sin embargo, los algoritmos de GPGPU probados que la utilizan resultaron no ser eficientes. Principalmente, esto se debió a que por cada sub-problema se leía el vector global k y escribía en el vector global $k+1$ independientemente, sin utilizar memoria compartida y utilizando mutex para no permitir condiciones de carrera en la escritura. Por ello se propuso una modificación a esta representación, la cual será detallada en la siguiente sección.

4.4.1.2. Segunda Representación

Al analizar los problemas de la representación anterior, se propuso una modificación en la representación de los sub-problemas. En el caso anterior, cada instancia del subproblema estaba representado por una tupla (fila, columna, *sm). Al realizar el algoritmo de multiplicación, se escribe sobre el vector res en las posiciones de memoria que van desde $[fila, fila + sm.n]$, donde $sm.n$ es la cantidad de filas de la sub-matriz sm. Si tenemos muchas de estas instancias ejecutándose concurrentemente, dos sub-problemas distintos pueden escribir sobre la misma posición de memoria global donde está almacenado el vector res. Por ello, es necesario la utilización de mutex para la escritura, penalizando el tiempo de ejecución.

En esta nueva representación, se divide el arreglo res en tramos de memoria consecutivos de tamaño 256. Ahora, cada sub-problema tiene asignado un único segmento del vector res y un conjunto de las instancias anteriores que en la multiplicación escriba al menos un elemento del segmento asociado. Las instancias son redefinidas como una tupla ($fila$, $columna$, $*sm$, $offset\_begin$, $offset\_end$), donde :

- $*sm$ representa un puntero a una de las matrices inyectadas.
- $fila$ y $columna$ representan el offset relativo para un camino a la matriz $*sm$.
- $offset\_begin$ y $offset\_end$ representan los índices del primer elemento de la sub-matriz $*sm$ que escribirá sobre el segmento de res del subproblema y el primer elemento de la sub-matriz $*sm$ que no escribirá en el segmento de vector res del sub-problema.

Recordemos que sm es una sub-matriz en representación COO y las tuplas están ordenadas primero por fila y luego por columna. Esto hace que todos los elementos del intervalo $[offset\_begin, offset\_end)$ escriban sólo en el segmento de res del subproblema. Así, se genera una independencia total en la escritura en res entre los subproblemas, haciendo que no sea necesario tener cuidado de alguna condición de carrera en la escritura del vector global res.

El algoritmo para construir el conjunto de instancias para cada subproblema es simple. Se recorre cada una de las instancias $(fila, columna, *sm)$ anteriores y se divide en tantas instancias de las nueva como sea necesario según los valores de $fila$ y $fila + sm.n$. Es

importante destacar porque se eligió 256 para el valor de partición del vector res. Por lo general, se elige un nivel del MTBDD que sea óptimo en consumo de memoria para inyectar las sub-matrices. En este nivel, las sub-matrices tienen al rededor de 1000 filas, por lo que, una instancia $(fila, columna, *sm)$ es partida en a lo sumo 6 instancias que se utilizan para representar los sub-problemas en este nuevo algoritmo. Por ello, podemos limitar el consumo de memoria a aproximadamente 6 veces más los valores analizados en la primera aproximación de solución. En la práctica, este factor suele estar muy por debajo de 6.

En la Figura 4.6 se muestra como las instancias que fueron propuestas inicialmente, aquí las llamamos *path*, son divididas en el nuevo tipo de instancias y agrupadas en sub-problemas o tasks. Las llamadas p_inst_x (donde x es un número) corresponden a instancias generadas a partir de $path_x$

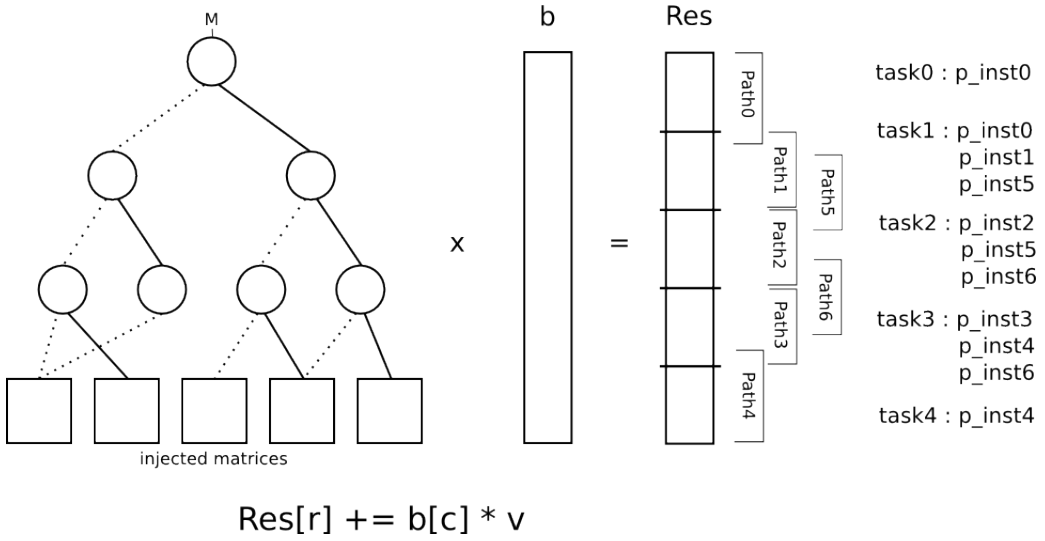


Figura 4.6: Producto Matriz - vector

Resumiendo, cada sub-problema queda representado por :

- Posición i del segmento del vector res sobre el cual se escribirá
- Conjuntos de instancias $(fila, columna, *sm, offset\_begin, offset\_end)$ los cuales escriben exclusivamente sobre $res[i * 256, (i + 1) * 256)$

y la matriz de transiciones esta representada por todo el conjunto de subproblemas y el conjunto de submatrices inyectadas.

Cálculo de consumo de memoria

Para calcular la memoria utilizada en esta nueva representación se usaron los mismos algoritmos descriptos en la representación anterior. El cálculo de la memoria necesaria para

las submatrices inyectadas es idéntico. En el caso de la cantidad de instancias, en esta nueva representación, cada instancia propuesta inicialmente puede ser dividida en varias nuevas instancias dependiendo del offset de fila y del tamaño de la submatriz a la que llega. Por esta razón, la complejidad de hacer un algoritmo que calcule el valor exacto de la cantidad de memoria sería proporcional a la cantidad de caminos posibles. En su lugar se utilizó el mismo algoritmo que cuenta la cantidad de caminos distintos desde la raíz a una matriz inyectada (proporcional al número de nodos del MTBDD utilizando memorización), pero en lugar de devolver 1 cuando se llega al nivel de inyección de matrices se devuelva la cantidad de instancias, del nuevo tipo, que se necesitarán para representar ese camino en el peor caso de acuerdo al tamaño del partición del vector res y la cantidad de filas de la matriz inyectada a la que se llega. Este valor es :

$$\lfloor \frac{sm.n}{256} \rfloor + 2$$

donde $sm.n$ es el número de filas de la matriz inyectada, 256 el tamaño de partición del vector res.

La cantidad de memoria necesaria para la representación y la paralelización es casi despreciable con respecto a la memoria necesaria para representar los arreglos para los vectores del algoritmo. En el siguiente Capítulo se detalla la cantidad de memoria necesaria para ciertos casos de estudios y se lo compara con la cantidad de memoria utilizada por el motor híbrido de PRISM.

4.4.2. Algoritmos en GPGPU

En Secciones anteriores se han analizado algunas técnicas para poder mejorar el desempeño de los algoritmos para GPGPU. Además de tenerlas en cuenta, desde la experiencia conseguida en el trabajo, vamos a proponer dos técnicas distintas a la hora de mejorar el rendimiento.

Por un lado tenemos un algoritmo de multiplicación, en donde se intentó modificar la estructura de representación hasta el punto que recorrerla y los accesos a memoria sean eficientes. Esto resulta en un aumento de eficiencia ya que las modificaciones sobre la estructura se realiza una única vez y el algoritmo de multiplicación cientos o miles de veces. Por otro lado tenemos el algoritmo que realiza operaciones vector-vector y una reducción. Las estructuras de representación son muy simples (arreglo de memoria continua) y sus valores cambian en cada iteración. Por ello, para conseguir el mejor rendimiento, se busca explotar todas las técnicas de más bajo nivel, muy relacionadas con la arquitectura de la placa.

En esta sección, daremos una noción superficial de los algoritmos, sin llegar a mostrar código real ya que este posee muchos detalles que van más allá del alcance de este trabajo. Es importante aclarar que el problema de operaciones vector-vector y reducción ya han sido

muy estudiados para arquitecturas GPGPU. En este trabajo sólo se implementó la técnica que se consideró más óptima.

Si nos retrotraemos a la comprensión del modelo de programación de GPU, tenemos que dividir el problema en sub-problemas independientes de alto nivel. Con independiente nos referimos a que cada uno de estos sub-problemas pueden ser ejecutados en cualquier orden, en forma secuencial o concurrente sin modificar la corrección del algoritmo. A cada uno de estos sub-problemas le vamos a asignar un grupo de hilos que trabajan cooperativamente, al cual llamamos bloque. Los bloques podrán utilizar memoria compartida, primitivas de sincronización de barrera para llevar a cabo su tarea. A su vez, cada bloque está dividido en conjuntos de 32 hilos, a los cuales llamaremos warps, estos hilos en realidad son ejecutados en procesadores vectoriales. Es decir, todos ejecutan al mismo tiempo la misma instrucción sobre datos distintos y por lo tanto están naturalmente sincronizados. Es importante, como ya lo analizamos, que haya la menor cantidad de bifurcaciones en su ejecución.

4.4.2.1. Algoritmo de Multiplicación Matriz-Vector

En el algoritmo de multiplicación tenemos un conjunto de sub-matrices las cuales corresponden a los nodos de un nivel particular del MTBDD de representación de la matriz de transición y los arreglos globales que representan los vectores de iteración $b, res, diagonal$. Además, tenemos la representación de un grupo de sub-problemas la cual consta de :

- Una sección de tamaño de 256 del vector res
- Conjunto de sub-instancias $(fila, colimna, *sm, offset_{begin}, offset_{end})$

El algoritmo consiste en asignar un bloque de 192 hilos para resolver cada subproblema. Cada bloque ejecutará el siguiente algoritmo.

- Inicializar un arreglo de tamaño 256 de memoria compartida a 0, el cual se utilizará como una cache intermedia en memoria compartida de lo que se escribirá en el vector res.
- Sincronizar los hilos del bloque.
- Asignar una warp a cada sub-instancia, las cuales realizaran la multiplicación normal sobre la matriz en COO, leyendo del vector b que está en memoria global y escribiendo en la memoria compartida. La escritura sobre memoria compartida es efectuada con una primitiva de suma atómica para evitar condiciones de carrera.
- Sincronizar los hilos del bloque.
- Escribir en el bloque correspondiente del vector res en memoria global.

El algoritmo en GPU es simplemente esto, como ya dijimos el mayor trabajo se realizó en la modificación de las estructuras de representación ya descriptas.

4.4.2.2. Algoritmo de Reducción

Como ya se explicó, este algoritmo ha sido muy bien estudiado y utiliza técnicas de muy bajo nivel de la arquitectura GPU. Por ello sólo explicaremos los fundamentos detrás de la técnica de paralelización con GPGPU. Más detalles sobre algoritmos de reducción para arquitecturas GPU pueden ser encontrados en [H+07].

La reducción es utilizada para calcular la convergencia, por lo cual necesitaremos una reducción con la operación mínimo. El algoritmo más genérico consiste en recorrer cada elemento y en utilizar una variable adicional para minimizar el valor. Este algoritmo funciona muy bien en procesadores secuenciales. Sin embargo, en arquitecturas paralelas no es muy eficiente. En el Algoritmo 2 presentamos un algoritmo de reducción para arquitecturas secuenciales.

```
reduccion_min(int *valores, int count)
    int minimo := inf
    for i desde 0 hasta count-1 do
        minimo = min(minimo, valor[i])
    od
ret : minimo
```

Algoritmo 2: reducción secuencial

Existe otra versión simple de la reducción. Para un arreglo de n elementos se hacen $\log(n)$ pasadas y en cada pasada se minimizan dos elementos. A continuación se presenta un posible código.

```
reduccion_pasada_paralela(int *valores, int count, int offset)
    int *copia = [valores, valores+count]
    for tid desde 0 hasta count-1 do
        if tid >= offset fi
            valores[tid] = min(copia[tid], copia[tid-offset])
    od
```

Algoritmo 3: Algoritmo de recorrido

En el Algoritmo 3, cada iteración del ciclo for se reduce el valor en $i - offset$ con el valor en i. Para un arreglo de 16 elementos, la función *reduccion_pasada_paralela* corre para un offset de $1, 2, 4, 8$. A continuación se muestra como se va modificando el arreglo *valores* después de cada llamada al Algoritmo 3.

```
reduccion_paralela(int *valores, int count)
    for offset desde 1 mientras offset < count haciendo offset *= 2 do
        reduccion_pasada_paralela(valores, count, offset)
    od
```

Algoritmo 4: Algoritmo reducción $O(n * \log(n))$

valores [11, 9, 1, 4, 3, 4, 12, 8, 7, 4, 20, 9, 4, 5, 2, 13]
offset=1 [11, 9, 1, 1, 3, 3, 4, 8, 7, 4, 4, 9, 4, 4, 2, 2]
offset=2 [11, 9, 1, 1, 1, 1, 3, 3, 4, 4, 4, 4, 4, 4, 2, 2]
offset=4 [11, 9, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 4, 4, 2, 2]
offset=8 [11, 9, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Como podemos ver, la última pasada produce un arreglo de valores, donde

$$valores[i] = min_{0<=j<=i}(valores_{iniciales}[i])$$

Claramente en arquitecturas secuenciales este segundo algoritmo de reducción es más ineficiente que el primero propuesto. Sin embargo, en arquitecturas paralelas de procesamiento vectorial cada ciclo del for en el Algoritmo 3 puede ser asignado a un hilo distinto y sincronizandar después de cada recorrido como lo hace el Algoritmo 5.

```
reduccion_paralela(int *valores, int count)
    tid = thread.id
    for offset desde 1 mientras offset < count haciendo offset *= 2 do
        if ( tid ≥ offset )
            valores[tid] = min(valores[tid], valores[tid - offset])
        fi
        barrera()
    od
ret : valores[count-1]
```

Algoritmo 5: Algoritmo para procesador vectorial con *count* hilos

El Algoritmo 5 corresponde a un kernel de CUDA escrito en pseudo-código. Este kernel se correrá con *count* hilos y cada hilo tendrá un valor distinto de *tid*. Este algoritmo es una buena aproximación a un algoritmo vectorial para reducción, sin embargo tiene varios problemas que limitan su eficiencia. Como para nombrar algunos, las cargas de procesamiento de cada hilo no están balanceadas. Cuando el *offset* es 8, por ejemplo, los ocho primeros hilos están desocupados. Otros problemas con este algoritmo es que la carga de procesamiento de cada hilo es muy poca y la sincronización con la barrera produce también baja de rendimiento.

Una solución es reducir el problema a subproblemas de tamaño constante y aprovechar la sincronización natural de las warp. Entonces un algoritmo eficiente para reducir segmentos de 256 valores utilizando sólo un warp sería:

- Reducir secuencialmente 8 valores consecutivos por cada hilo (Algoritmo 2)
- Almacenar el valor resultante de cada hilo del paso anterior memoria compartida
- Reducir los 32 valores obtenidos que se encuentran en memoria compartida de forma cooperativa utilizando la sincronización natural de las warps.
- devolver el valor obtenido.

Este algoritmo se puede escalar al nivel de bloques realizando las mismas operaciones. Es decir, cada bloque posee un número fijo de warps, cada warp reduce un tramo de 256 valores consecutivos. Luego cada warp devuelve un valor, este es puesto en memoria compartida, las warps se sincronizan y una warp del bloque los vuelve a reducir. Todo estos pasos son hechos teniendo en cuenta los accesos a memoria.

Entonces, el problema puede ser dividido en subproblemas de tamaño fijo (m), teniendo en cuenta que si el tamaño final del problema no es múltiplo de m se agregaran valores neutros para la reducción. Luego se realizan dos pasadas, primero cada bloque resuelve cada subproblema y luego el resultado de cada subproblema es vuelto a reducir utilizando un bloque.

Con esta estrategia, si cada bloque posee 8 warp y cada warp procesa segmentos de 256 valores. El tamaño de cada subproblema puede ser 2048. Por lo tanto se pueden reducir un arreglo de tamaño 2048 * 2048 valores, es decir, 4 Giga valores con sólo dos pasadas.

Finalmente diremos que las constantes de 8 valores por hilo y 8 warp por bloque no son valores elegidos al azar, sino que dependen de la arquitectura particular de la GPU que se esté utilizando. Además, el limite de 4 Giga valores puede ser extendido fácilmente de ser necesario.

Capítulo 5

Resultados

En este capítulo analizaremos los resultados obtenidos en la paralelización del chequeo de modelos utilizando el motor híbrido de PRISM y el método de Jacobi. Primero presentaremos una serie de modelos y propiedades que vamos a chequear. Luego analizaremos los resultados obtenidos, tanto en rendimiento de procesamiento como en consumo de memoria, utilizando los algoritmos presentados en los capítulos anteriores.

5.1. Casos de estudio

Comenzaremos enumerando los distintos modelos utilizados junto con una breve descripción de los mismos.

5.1.1. Colas Tandem [Tandem]

Este caso de estudio se basa en un sistema de colas Tandem [HMKS99]. Se modelan una serie de colas ordenadas de capacidad c. Los clientes deben recorrer todas las colas en orden.

En PRISM, el sistema se modela como un CTMC en donde los tiempos de servicios de cada cola están modelados con distribuciones de probabilidad exponencial.

Hay varias preguntas interesantes que podemos hacer sobre este modelo, en este caso de estudio vamos a chequear el valor esperado de la cantidad de clientes en el sistema.

5.1.2. Sistema de Sondeo Cíclico [Polling]

En este caso de estudio se modela un sistema de dos o más estaciones de trabajo y un servidor que las atiende de forma cíclica (*polling*). El servidor va revisando las estaciones y si hay un cliente en espera, lo atiende. De lo contrario, continua la estación siguiente en orden cíclico. El sistema es analizado en detalle en [IT90].

En PRISM, la representación utilizada es una CTMC que modela el servidor y las estaciones. Con N se denota la cantidad de estaciones. Cada estación tiene una cola que almancena a lo sumo un token. Los tiempos de llegadas de los toquen en las colas están modelados con una distribución de probabilidad exponencial.

Para este modelo analizaremos dos propiedades relevantes: la probabilidad que la Estación 1 este esperando ser servida (*s1*) y la probabilidad de que la Estación 1 sea atendida antes que la Estación 2 (*s1-before-s2*).

5.1.3. Sistema de Manufactura Kanban [Kanban]

Este caso está basado en el sistema de manufactura presentado en [CT96] que emplea la la metodología Kanban para controlar la producción. El sistema consta de un esquema de dependencias en la línea de producción y la demanda del consumidor determina el ritmo de la manufactura. Por lo tanto, el sistema se comporta de forma reactiva, en lugar de intentar anticiparse a la demanda realizando pronósticos de consumo.

La descripción clásica hace uso de las denominadas tarjetas Kanban. Estas tarjetas se adjuntan a los lotes procesados por cada unidad de producción. Cada unidad de producción posee un número limitado de tarjetas, por lo que cuando éstas se agotan la producción se detiene. Por otro lado, cuando una fase posterior hace uso de uno de estos lotes, esta también envía la tarjeta a la unidad que lo produjo, lo que funciona como notificación para reactivar la producción. El número de tarjetas en una unidad dependerá de factores como la velocidad de fabricación que afecten la capacidad de una unidad de suministrar insumos a sus clientes. La intención es que este número sea lo más pequeño posible.

A su vez, cada modelo de producción atiende las tarjetas que posee y cada lote puede ser procesado correctamente o detectar un error en la fabricación y el lote debe ser reprocesado hasta su correcto procesamiento.

El sistema es modelado con una CTMC, en donde la ocurrencia de todos los eventos en el proceso de manufacturación están modelados con distribuciones de probabilidad exponencial con distintos parámetros. Un parámetro t controla la cantidad de tarjetas asignadas a cada etapa de producción.

Para este caso de estudio chequearemos el valor esperado en la productividad del sistema.

5.1.4. Sistema de Manufactura Flexible [FMS]

El caso de estudio FMS es un sistema de manufactura flexible [CT93] que permite al sistema de producción reaccionar en el caso de cambios previstos o imprevistos. Se basa en un conjunto de nodos (robots de producción automáticas, sensores, maquinas de inspección) en distintos segmentos de producción interconectados que se envían mensajes entre si.

En PRISM, el sistema se modela como una CTMC en donde la interacción entre los nodos son simuladas con distribuciones de probabilidad exponenciales. Utilizaremos la variable N para denotar el número de lotes en el sistema.

Al igual que en el caso de Kanban, calcularemos el valor esperado de la productividad del sistema variando los tokens del mismo.

5.1.5. Estaciones de Trabajo [Cluster]

Este sistema se basa en un grupo de estaciones de trabajo (clusters) interconectadas, analizado en detalles en [HHK00]. Este, está compuesto de dos subgrupos de N estaciones de trabajo. Cada subgrupo de estaciones está conectado en topología estrella a través de un conmutador. A su vez, los conmutadores de cada subgrupo están conectados entre sí. Todos los componentes del sistema pueden fallar y hay una unidad de apoyo en tal caso para todas las componentes.

En PRISM el sistema esta modelado como una CTMC y son utilizadas distribuciones de probabilidad exponencial para simular los fallos de las estaciones de trabajo.

En este modelo analizaremos propiedades de calidad de servicio. En particular estamos interesados en la probabilidad de que al menos N estaciones estén operacionales y conectadas entre sí a lo largo del funcionamiento del sistema.

5.1.6. Protocolo de Retransmisión Acotada [BRP]

El sistema BRP es una DTMC o cadena de Markov de tiempo discreto que modela dos actores que desean comunicarse entre si y consta de 4 entidades: el emisor, el receptor y dos canales unidireccionales de comunicación, uno en cada sentido. El modelo cuenta con dos parámetros configurables: N , la cantidad de fragmentos a enviar y MAX , el número máximo de intentos de retransmisión. El sistema está detallado en [HSV94].

El mensaje a ser enviado se divide en N fragmentos. El emisor puede transmitir sólo un fragmento a la vez. A su vez, el receptor envía de vuelta mensajes de confirmación por cada fragmento recibido. Ambos canales tiene una probabilidad no nula de perder un mensaje. Si la trasmisión falla, el fragmento es retransmitido un número acotado de veces. Si después de un número máximo de retransmisiones no se recibe una confirmación, el emisor se da por vencido y activa una condición de error.

En este modelo vamos a chequear la probabilidad de que el emisor trasmita su mensaje exitosamente.

5.2. Experimentos y Análisis

5.2.1. Medición de rendimiento en el procesamiento

Para la experimentación y el análisis de los resultados obtenidos, solo nos concentraremos en el tiempos del motor numérico de PRISM. Por ello, hablaremos del rendimiento (speed-up) del sistema como el rendimiento obtenido en este motor en lugar del rendimiento obtenido en el chequeo completo del modelo. Cabe destacar que el motor numérico es la etapa que más tiempo toma en el chequeo de modelos.

Al analizar el rendimiento, nos interesa la proporción entre el tiempo que le toma al motor numérico híbrido de PRISM para un caso de estudio en particular corriendo en una arquitectura secuencial (tiempo de PRISM) contra el tiempo que le toma al algoritmo propuesto en la tesis (tiempo algoritmo GPU). Para este último, tendremos en cuenta tanto el tiempo secuencial de construcción de las estructuras necesarias para la paralelización como el tiempo en GPU para llevar a cabo todas las iteraciones necesarias para la convergencia. Cabe destacar que el tiempo secuencial al cual nos referimos es prácticamente despreciable en proporción al tiempo de GPU.

5.2.2. Entorno de prueba

En la Tabla 5.1, detallamos el entorno utilizado para realizar las pruebas. Es importante destacar que Kepler no solo es la más rápida, sino también la más compleja arquitectura de GPU construida por la empresa NVIDIA hasta la fecha. Tesla K20 provee un rendimiento teórico cercano a 1 TFLOPS de operaciones en punto flotante de doble precisión. En el caso de la arquitectura CPU utilizada, esta provee un rendimiento teórico de 56 GFLOPS de operaciones en doble precisión utilizando todos sus cores.

CPU	
Procesador	Intel(R) Core(TM) i7 CPU 950 a 3.07GHz
Memoria	16 GB DDR3
GPU	
Procesador	Kepler GK110 a 706MHz
Memoria	5 GB DDR5
Interfaz	PCI Express 2.0

Figura 5.1: Entorno de prueba

5.2.3. Experimentos

Los experimentos realizados a los diferentes casos de estudio anteriormente propuestos se centraron en el análisis de los tiempo de speedup del motor híbrido en GPU como explicamos

anteriormente y la cantidad de memoria necesaria para la paralelización.

5.2.3.1. Medición de memoria para la paralelización

Podemos separar en dos la memoria utilizada para llevar a cabo el algoritmo de Jacobi tanto de forma secuencial como de forma paralela. A estas dos divisiones las llamaremos memoria base ($MemB$) y memoria de representación ($MemR$). La primera es la memoria necesaria para representar los dos vectores solución, el vector de diagonales y el vector $\underline{b}$ si es necesario (véase Algoritmo 1). La segunda es la memoria necesaria para representar la matriz de transición. En el caso del algoritmo de PRISM eso incluye la memoria para la estructura offset-labelled MTBDD y las matrices inyectadas. En el caso del algoritmo paralelo esta segunda memoria incluye las instancias, las matrices inyectadas y la representación de los subtrabajos.

En los diferentes gráficos de la Figura 5.2 se analiza, para cada caso de estudio, como varía la cantidad de memoria de representación necesaria para el algoritmo paralelo dependiendo del nivel en que se inyecten las matrices en la estructura MTBDD y del tamaño del modelo. Como ya explicamos en la medición de memoria para la Sección 4.4.1.2, los resultados equivalen al peor caso, por lo que en la práctica los valores son ligeramente menores. Aunque los resultados no son exactos, los valores del peor caso son lo suficientemente buenos para sacar conclusiones. Como ya se dijo varias veces, la cantidad de memoria necesaria crece exponencialmente con relación al tamaño del modelo, por ello para poder representar los gráficos de diferente tamaño de modelo para el mismo caso de estudio, se eligió representar la coordenada y en escala logarítmica.

Lo más interesante de los gráficos de la Figura 5.2 es analizar la forma cuasi-campana de la función de cantidad de memoria de representación por nivel de inyección de matrices. Podemos ver como casi todas las funciones tienden a ser parecidas. Recordemos que en los niveles 1 y nivel máximo la cantidad de memoria necesaria para la representación de la matriz de transición es proporcional al la cantidad de memoria necesaria en el motor explícito.

En la Figura 5.3 se grafican, para cada caso de estudio, la proporción de memoria de representación con relación a la memoria base, tanto para el algoritmo de PRISM como para el algoritmo de GPU propuesto en la tesis. Las barras están agrupadas según el tamaño del modelo. Las barras rojas y azules corresponden a la memoria de representación para GPU (submatrices + instancias + subtrabajos). Las barras rojas corresponden a la proporción de memoria de representación cuando se inyectan las matrices en el nivel de memoria mínima (**Memr mmem**) y las barras azules muestran la proporción de memoria de representación cuando se inyectan las matrices en el nivel de tiempo mínimo (**Memr tmin**). Por otra parte, las barras verdes representan a la proporción de memoria de representación del algoritmo de PRISM (MTBDD + matrices inyectadas), que llamaremos **Memr PRISM**. En la coorde-

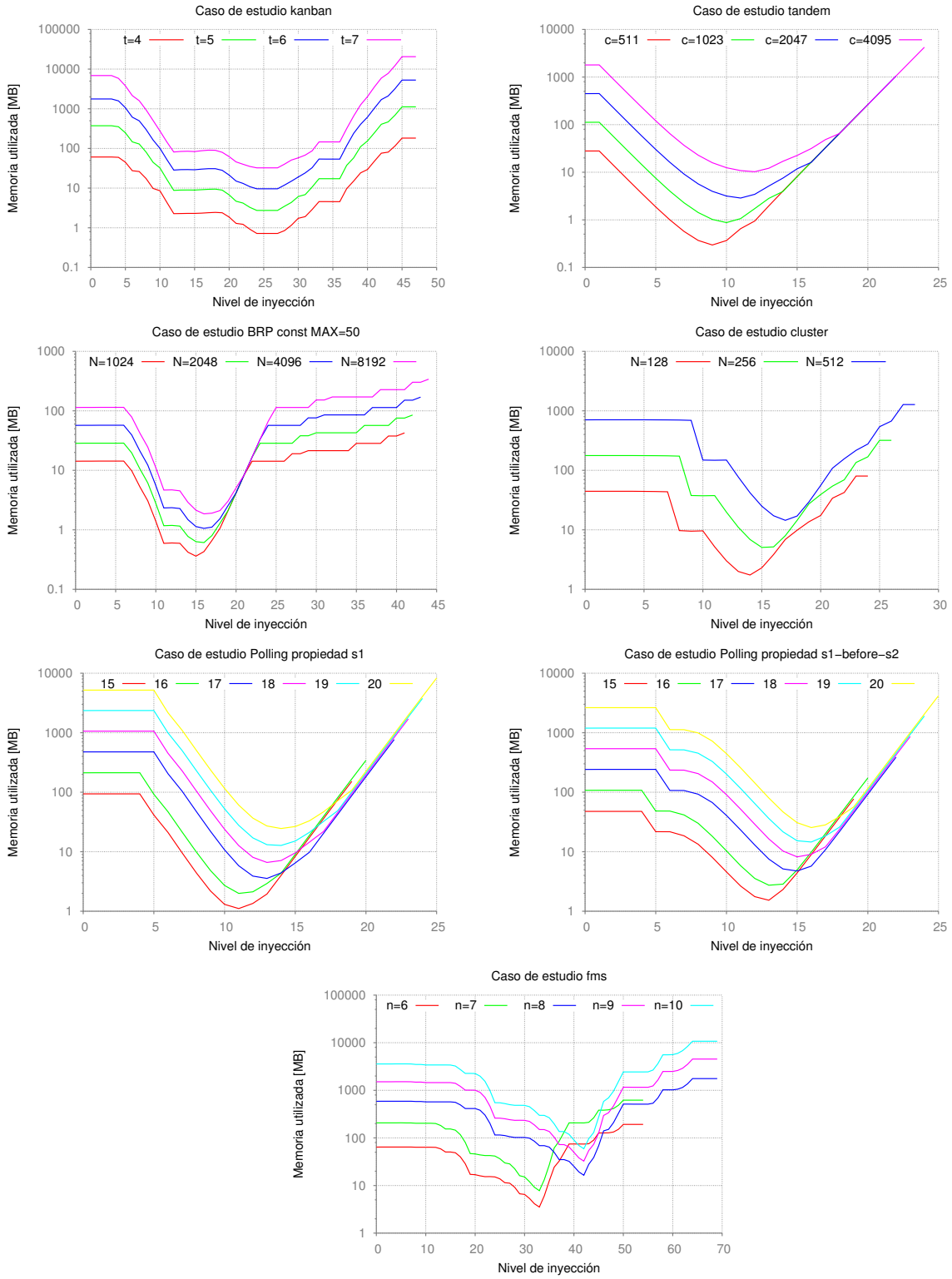


Figura 5.2: Memoria de representación

nada x, para cada conjunto de barras, se encuentra el tamaño del modelo (valor de variable) y el tamaño de la memoria base para ese caso.

Finalmente, analizando los gráficos de memoria y comparando la memoria utilizada por PRISM con la memoria utilizada por nuestra propuesta de paralelización, podemos concluir que el modelo de representación para arquitectura GPU necesita de más memoria que el propuesto en el modelo híbrido de PRISM. Sin embargo, la diferencia no es muy grande y en comparación a la memoria base, memoria que utilizan ambos algoritmos del mismo modo, en la mayoría de los casos es insignificante. Este gasto adicional de memoria y el cambio en la representación de la matriz de transiciones va a tener más sentido cuando se comparen los tiempos del motor híbrido secuencial de PRISM, con la paralelización del motor híbrido propuesta.

5.2.3.2. Resultados de rendimiento

La experimentación se realizó sobre los casos de estudio propuestos al comienzo de este Capítulo. Estamos interesados es dos resultados obtenidos durante la ejecución, por un lado la mejora relativa en el tiempo de ejecución del motor híbrido y además, la cantidad de memoria adicional necesaria para realizar la paralelización.

Como explicamos en el capítulo anterior, la cantidad de memoria adicional necesaria para la paralelización depende del nivel en que se realice la inyección de las matrices. Además, el tiempo de ejecución varía con la modificación de este nivel, siendo más rápido en los niveles cercanos al nivel de consumo de memoria mínimo. En particular, en los niveles superiores. Por ello, para cada ejecución con un tamaño particular de modelo, se detallan dos tiempos de ejecución:

- Tiempo de ejecución inyectando matrices en el nivel de memoria mínima ($\mathbf{Nivel}_{mmin}$)
- Tiempo de ejecución inyectando matrices en el nivel de tiempo mínimo ($\mathbf{Nivel}_{tmin}$)

En cuanto al consumo de memoria, se detallan:

- Consumo de memoria base, tanto para algoritmo secuencial como para la paralelización (**Mem Base**)
- Consumo de memoria adicional a la memoria base para la paralelización en el nivel de memoria mínima ($\mathbf{Memr}_{mmin}$)
- Consumo de memoria adicional a la memoria base para la paralelización en el nivel de tiempo mínimo ($\mathbf{Memr}_{tmin}$)

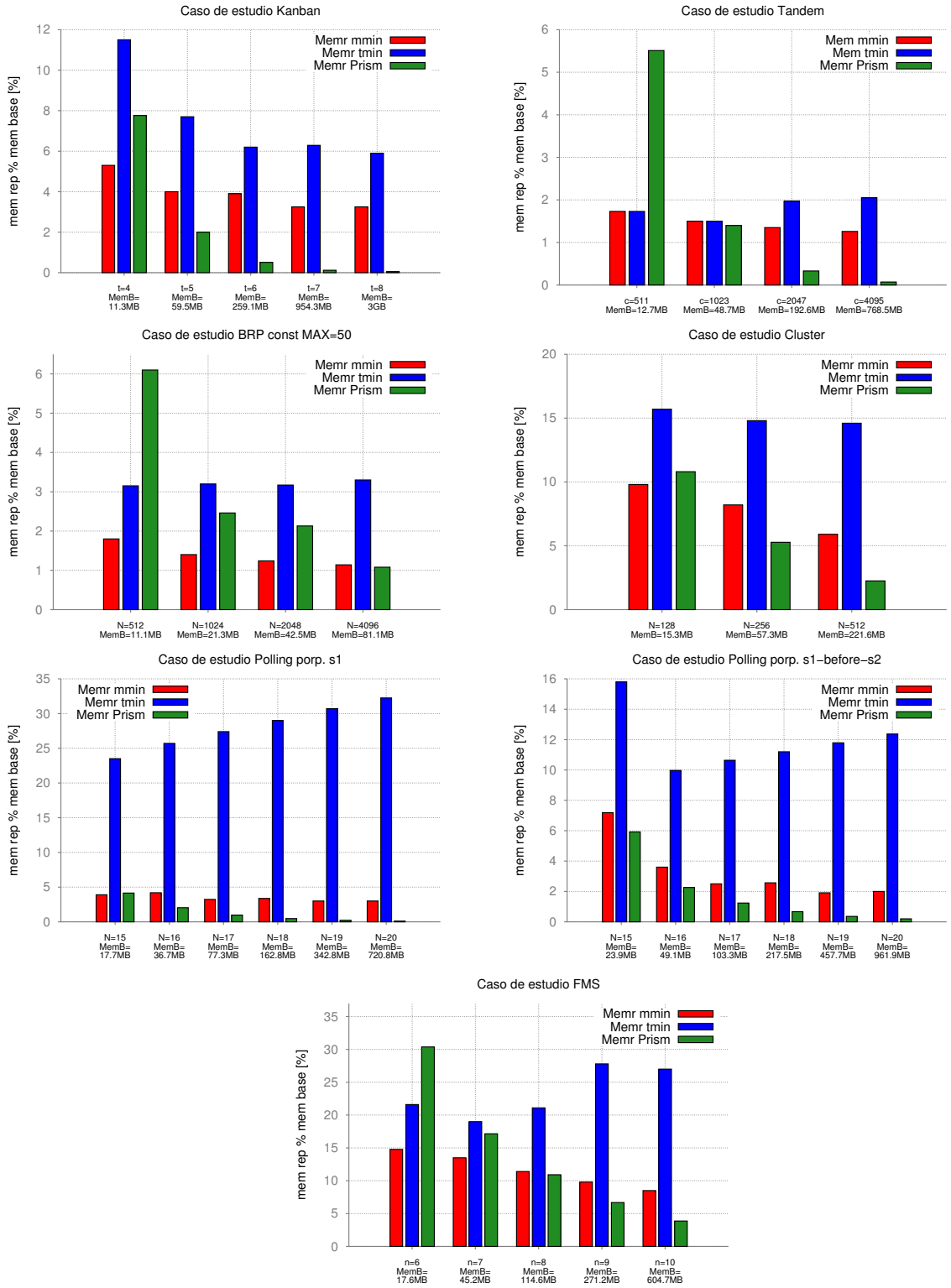


Figura 5.3: Memoria de representación en proporción a memoria base

	Tiempo (seg)					Memoria (MB)		
n	CPU	GPU_{mmin}	Speedup	GPU_{tmin}	Speedup	Mem Base	$Memr_{mmin}$	$Memr_{tmin}$
6	31.9	3.79	8.41	0.87	36.66	17.6	2.6	3.8
7	205.1	13.9	14.75	11.49	17,85	45.2	6.1	8.6
8	969.1	47.01	23,63	36.77	26.35	114.6	13.1	24.2
9	2541.22	141.66	17.93	98.55	25.78	271.2	26.6	75.5
10	8818.31	370.9	23.77	247.3	35.65	604.7	51.4	163

Figura 5.4: Tabla de resultados del caso de estudio FMS

	Tiempo (seg)					Memoria (MB)		
t	CPU	GPU_{mmin}	Speedup	GPU_{tmin}	Speedup	Mem Base	$Memr_{mmin}$	$Memr_{tmin}$
4	6.18	1.34	4.61	0.65	9.5	11.3	0.6	1.3
5	61.1	13.15	4.64	6.49	9.41	59.5	2.4	4.6
6	395.18	83.49	4.73	40.87	9.67	259.1	10.1	16.1
7	1976.26	434.93	4.54	190.87	10.35	954.3	31.1	60.1
8	8649.98	1862.29	4,64	822.24	10.52	3000	97.6	177.1

Figura 5.5: Tabla de resultados del caso de estudio Kanban

A continuación se presenta una tabla por cada caso de estudio. Cada fila corresponde a una corrida del modelo con una asignación particular de las variables. Las columnas de tiempos representan los distintos tiempos, ya sea corriendo secuencialmente o en paralelo inyectando las matrices en el nivel de uso de memoria mínima o en el nivel de tiempo mínimo. Además, las columnas de memoria representan la cantidad de memoria adicional utilizada en la paralelización en el nivel de memoria mínima y la memoria adicional en el nivel de tiempo mínimo.

Realizando un análisis general de los resultados obtenidos podemos sacar algunos resultados generales importantes.

- En primer lugar el speedup promedio conseguido en todos los casos se acerca a los **10.5X** para el algoritmo de GPU inyectando las matrices en el nivel de memoria

	Tiempo (seg)					Memoria (MB)		
N	CPU	GPU_{mmin}	Speedup	GPU_{tmin}	Speedup	Mem Base	$Memr_{mmin}$	$Memr_{tmin}$
15	72.77	7.94	9.16	6.03	12.06	23.9	1	2.2
16	176.84	25.96	6.81	12.66	13.9	49.1	1.8	4.9
17	400.54	57.21	7	28.94	13.8	103.3	2.6	11
18	914.12	149.92	6.09	61.35	14.9	217.5	5.6	24.4
19	2043.92	329.58	6.2	137.75	14.8	457.7	8.77	54
20	4590.52	807.42	5.7	303.57	15.1	961.9	19.3	119

Figura 5.6: Tabla de resultados del caso de estudio Polling-before

	Tiempo (seg)					Memoria (MB)		
N	CPU	GPU_{mmin}	Speedup	GPU_{tmin}	Speedup	Mem Base	$Memr_{mmin}$	$Memr_{tmin}$
15	16.7	4.6	3.63	1.84	9.07	17.7	0.69	4.17
16	41.14	12.41	3.31	4.09	10.05	36.7	1.54	9.43
17	98.6	28.8	3.42	9.4	10.48	77.3	2.5	21.19
18	245.03	74.44	3.29	21.76	11.26	162.8	5.5	47.3
19	556.7	170.46	3.26	50.34	11.05	342.8	10.25	105.19
20	1310.45	449.52	2.91	118.6	11.04	720.8	21.93	232.51

Figura 5.7: Tabla de resultados del caso de estudio Polling-s1

	Tiempo (seg)					Memoria (MB)		
N	CPU	GPU_{mmin}	Speedup	GPU_{tmin}	Speedup	Mem Base	$Memr_{mmin}$	$Memr_{tmin}$
128	19.19	0.86	22.3	0.43	44.6	15.3	1.5	2.4
256	238.28	43.14	5.52	19.75	12	57.3	4.7	8.5
512	9697.12	1157.67	8.37	646.89	15	221.6	13.1	32.4

Figura 5.8: Tabla de resultados del caso de estudio Cluster

	Tiempo (seg)					Memoria (MB)		
c	CPU	GPU_{mmin}	Speedup	GPU_{tmin}	Speedup	Mem Base	$Memr_{mmin}$	$Memr_{tmin}$
511	106.5	7.67	13.8	7.5	14.2	12.7	0.22	0.22
1023	788.25	52.04	15.1	34.46	22.8	48.7	0.73	0.73
2047	4393.5	308.55	14.2	218.40	20.1	192.6	2.6	3.8
4095	29527	1967	15	1338.6	22	768.5	9.7	15.8

Figura 5.9: Tabla de resultados del caso de estudio Tandem

	Tiempo (seg)					Memoria (MB)		
N,MAX	CPU	GPU_{mmin}	Speedup	GPU_{tmin}	Speedup	Mem Base	$Memr_{mmin}$	$Memr_{tmin}$
512,50	18.7	1.53	12.2	1.34	13.9	11.1	0.2	0.35
1024,50	76.58	5.38	14.2	4.49	17	21.3	0.3	0.68
2048,50	327.1	19.4	16.8	15.8	20.7	42.5	0.53	1.35
4096,50	1277.1	79.7	16	60.1	21.2	81.1	0.93	2.68

Figura 5.10: Tabla de resultados del caso de estudio BRP

mínima y de **17.8X** para el algoritmo de GPU inyectando las matrices en el nivel de tiempo mínimo.

- La memoria de representación para la paralelización se mantiene en niveles razonables comparados a la memoria base ($MemB$).
- El rendimiento en los niveles superiores al nivel de memoria mínima es notablemente mejor. Hay una compensación entre la mejora del rendimiento y la cantidad de memoria necesaria.

Capítulo 6

Conclusiones

Teniendo en mente todos los temas en que se ha basado esta tesis, vamos a dividir las conclusiones en dos partes. En primer lugar sacaremos las conclusiones relacionadas al ámbito académico, es decir, analizaremos el estudio de todos los temas y el desarrollo del trabajo desde el punto de vista del conocimiento que ha dejado. En segundo lugar, analizaremos como este trabajo puede incidir en la evolución de la paralelización de algoritmos de chequeo de modelos con arquitecturas de procesamiento masivo.

6.1. Conclusiones académicas

A lo largo de esta tesis hemos estudiado y presentado diferentes campos de estudio de las Ciencias de la Computación para luego poder combinarlos y obtener resultados satisfactorios.

En un comienzo, hemos estudiado como modelar sistemas estocásticos ó con comportamiento probabilístico con formalismos matemáticos como son las cadenas de Markov. También, hemos estudiado cómo representar propiedades en lógicas temporales como PCTL o CSL y como chequear estas lógicas en modelos DTMC y CTMC. En esta parte del trabajo, hemos mostrado como el chequeo de ciertas propiedades se reducen a un sistema lineal de ecuaciones y la solución de este sistema utilizando el método iterativo de Jacobi, tema principal del trabajo.

En cuanto a Computación de Alta Performance, hemos argumentado por qué la arquitectura de procesamiento masivo o GPU es una arquitectura adecuada para resolver problemas que demandan de un gran procesamiento numérico. Además, hemos presentado el marco teórico y he explicado las técnicas para obtener el máximo rendimiento. Finalmente, hemos estudiado la estructura de datos BDD para representar matrices y vectores, y he investigado sobre cómo modificarlas para aprovechar los beneficios de las arquitecturas GPU.

Todo este proceso ha demandado el estudio de los temas a través de la lectura de publicaciones científicas, libros académicos y tutoriales. También ha requerido investigar, proponer

soluciones y demostrar empíricamente si funcionaban del modo requerido. Para finalizar, hemos tratado de escribir de forma simple y técnica el trabajo desarrollado.

Por todo ello, como conclusión general del proceso académico a lo largo del trabajo, podemos afirmar que la tesis sirvió como medio para introducirme en el proceso de investigación y ampliar mis conocimientos en campos muy variados de las Ciencias de la Computación.

6.2. Conclusión de resultados obtenidos

Hemos presentado en este trabajo un algoritmo paralelo utilizado para el problema de verificación de modelos con estructuras simbólicas. Todo está basado en la paralelización del método iterativo de Jacobi con estructuras de representación simbólicas utilizando arquitecturas de procesamiento masivo como son las GPU. Hemos demostrado, a través de la experimentación, como el algoritmo propuesto mejora considerablemente el tiempo requerido para el chequeo de modelos con una penalidad relativamente pequeña en el consumo de memoria. Así se ha alcanzado el objetivo de mejorar los tiempos de los algoritmos numérico manteniendo el consumo de memoria cercano al motor híbrido de PRISM.

La paralelización del chequeo de modelos probabilísticos simbólico con arquitecturas SMP, es decir, múltiples procesadores trabajando en memoria uniforme, ha sido ampliamente estudiado. Algunos resultados pueden ser encontrado en [KPZM04, ZPK05, Lag11]. Sin embargo, la paralelización con arquitecturas de procesamiento masivo no ha sido muy estudiada todavía. Se conocen algunos resultados obtenidos en [BESW10], pero en todos los casos se centran en la paralelización del motor explícito. Por ello, consideramos que este trabajo y los resultados obtenidos son un gran aporte para la comunidad que estudia las técnicas de verificación de modelos. Además, algunas de las técnicas y modificaciones hechas a las estructuras de representación pueden ser un punto de partida para la paralelización del método de Gauss-Seidel o para el chequeo de modelos MDP (procesos de decisión de Markov). Fuera del ámbito de sistemas dependibles, las estructuras MTBDD son muy utilizadas, por lo que nuestro trabajo también puede ser importante en otros campos de estudio.

Desde el punto de vista de la viabilidad, las arquitecturas de GPU para el procesamiento de propósito general se están convirtiendo en una opción muy popular, lo que está haciendo que sus precios decrezcan y su ciclos de actualización avancen rápidamente. Gracias a esto y a que el procesamiento masivo ofrece un desempeño mayor por Watt, creemos que esta solución para la paralelización del chequeo de modelos cobra aún mayor importancia y puede ser considerada como una solución a futuro.

Como futuros trabajos, esta tesis puede ser un punto de partida, como ya dijimos, para paralelizar otros algoritmos numéricos como Gauss-Saidel o para la paralelización del chequeo de modelos MDP. Además, este trabajo se puede extender proponiendo técnicas para escoger los niveles del inyección de matrices automáticamente y de este modo eliminar esta decisión

de las manos del usuario.

References

[Ake78] Sheldon B. Akers. Binary decision diagrams. *Computers, IEEE Transactions on*, 100(6):509–516, 1978. 29

[ASSB96] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Verifying continuous time markov chains. In *Computer Aided Verification*, pages 269–276. Springer, 1996. 9

[BCM+92] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 10 to the power of 20 states and beyond. *Information and computation*, 98(2):142–170, 1992. 29

[BESW10] D. Bosnacki, S. Edelkamp, D. Sulewski, and A. Wijs. GPU-PRISM: An extension of PRISM for general purpose graphics processing units. In *Proc. 9th International Workshop on Parallel and Distributed Methods in Verification*, 2010. 3, 62

[BFG+97] R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebric decision diagrams and their applications. *Formal methods in system design*, 10(2-3):171–206, 1997. 31

[BHHK00] Christel Baier, Boudewijn Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model checking continuous-time markov chains by transient analysis. In *Computer Aided Verification*, pages 358–372. Springer, 2000. 13

[BKH99] Christel Baier, Joost-Pieter Katoen, and Holger Hermanns. Approximative symbolic model checking of continuous-time markov chains. In *CONCUR'99 Concurrency Theory*, pages 146–161. Springer, 1999. 9

[BLL+96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal — a tool suite for automatic verification of real-time systems. In Rajeev Alur, ThomasA. Henzinger, and EduardoD. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer Berlin Heidelberg, 1996. 2

[CCGR99] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model verifier. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499. Springer Berlin Heidelberg, 1999. 2

[CES86] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986. 10

[CT93] G. Ciardo and K. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993. 50

[CT96] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stocastic Petri nets. ICASE Report 96-35, Institute for Computer Applications in Science and Engineering, 1996. 50

[CY88] Costas Courcoubetis and Mihalis Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pages 338–345. IEEE, 1988. 10

[FMY97] Masahiro Fujita, Patrick C. McGeer, and JC-Y Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal methods in system design*, 10(2-3):149–169, 1997. 31

[H+07] Mark Harris et al. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2, 2007. 45

[HHK00] B. Haverkort, H. Hermanns, and J.-P. Katoen. On the use of model checking techniques for dependability evaluation. In *Proc. 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 228–237, Erlangen, Germany, October 2000. 51

[HJ94] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994. 7, 10

[HMKS99] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time markov chains, 1999. 49

[Hol97] G.J. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997. 2

[HSV94] L. Helmink, M. Sellink, and F. Vaandrager. Proof-checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Proc. International Workshop on Types for Proofs and Programs (TYPES'93)*, volume 806 of *LNCS*, pages 127–165. Springer, 1994. 51

[IT90] O. Ibe and K. Trivedi. Stochastic Petri net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990. 49

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011. 2

[KPZM04] M. Kwiatkowska, D. Parker, Y. Zhang, and R. Mehmood. Dual-processor parallelisation of symbolic probabilistic model checking. In D. DeGroot and P. Harrison, editors, *Proc. 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*, pages 123–130. IEEE Computer Society Press, 2004. 62

[KSK66] John G Kemeny, James Laurie Snell, and Anthony W Knapp. *Denumerable markov chains*. Van Nostrand Princeton, 1966. 6

[Lag11] Pablo Dal Lago. Paralelización de algoritmos para verificación simbólica de modelos probabilísticos. Master's thesis, Universidad Nacional de Córdoba - Facultad de Matemática, Astronomía y Física, 2011. 62

[Lee59] Chang-Yeong Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959. 29

[McM93] Kenneth L McMillan. *Symbolic model checking*. Springer, 1993. 29

[Par02a] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002. 32, 35, 36

[Par02b] David Anthony Parker. *Implementation of symbolic model checking for probabilistic systems*. PhD thesis, University of Birmingham, 2002. 12

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005. 3

[WB12] AntonJ. Wijs and Dragan Bošnački. Improving gpu sparse matrix-vector multiplication for probabilistic model checking. In Alastair Donaldson and David Parker, editors, *Model Checking Software*, volume 7385 of *Lecture Notes in Computer Science*, pages 98–116. Springer Berlin Heidelberg, 2012. 3

[ZPK05] Y. Zhang, D. Parker, and M. Kwiatkowska. A wavefront parallelisation of CTMC solution using MTBDDs. In *Proc. International Conference on Dependable Systems and Networks (DSN'05)*, pages 732–742. IEEE Computer Society Press, 2005. 62