Facultad de Matemtica, Astronoma y Fsica

Universidad Nacional de Crdoba

# Implementación del algoritmo Gauss-Seidel en CUDA para la verificación simbólica de modelos probabilísticos.

Gastón Ingaramo

Director: Dr. Pedro R. D'Argenio

Córdoba, 17 de diciembre de 2013

# Agradecimientos

Quisiera agradecer en primer lugar a Pedro D'Argenio y Carlos Bederián, quienes sin su paciencia y ayuda este trabajo no hubiera sido posible.

Segundo a Matías Tealdi, mi amigo desde que ingresé a esta facultad, con quien en conjunto hemos compartido tantas experiencias reconfortantes.

Finalmente, deseo expresar mi mas profunda gratitud y admiración a mis padres Horacio, Alicia, quienes tanto me apoyaron a lo largo de la carrera y a mi hermano Lucas a quien le debo mis comienzos en computación.

# Resumen

Presentamos los distintos lenguajes para el modelado de sistemas, tanto de tiempo discreto como de tiempo continuo, DTMC y CTMC respectivamente. Luego estudiamos las lógicas correspondientes para su verificación prestando especial atención a los operadores que requieren de multiplicaciones matriz-vector. Explicamos como PRISM, el model checker simbólico utilizado, implementa estos operadores y que estructuras de datos utiliza.

Finalmente presentamos nuestra implementación de un algoritmo pseudo Gauss-Seidel en arquitecturas GPGPU, principalmente en CUDA. Hacemos dos distinciones, cuando la memoria de la tarjeta es suficiente para contener todos los vectores y cuando se requiere dividir el problema. Concluimos con un análisis experimental que demuestra las mejoras del desempeño obtenidas.

**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

# Capítulo 1

# Introducción

## 1.1. Motivación

Vivimos en un mundo donde las computadoras están presentes en cualquier aspecto de nuestra vida. Éstas nos facilitan las tareas de la vida cotidiana, desde la navegación por GPS hasta el reloj despertador. Pero también componen sistemas críticos como el controlador de una máquina de rayos X o el sistema de navegación de un cohete por ejemplo. Estos sistemas críticos son desarrollados por varios individuos, y como el ser humano no está libre de cometer errores tampoco lo están estos sistemas. En la historia contamos con varios ejemplos de casos en los cuales estos sistemas han pasado exitosamente controles de calidad pero han resultado en catástrofes con consecuencias fatales. Esto se debe a que los sistemas críticos normalmente están compuestos por varios sistemas de gran complejidad individual. La práctica del testing ha demostrado ser útil para detectar errores, pero no para demostrar su ausencia.

Es por estas razones que resulta imperativo para seguir confiando en los sistemas críticos tener herramientas de verificación formal, en donde el resultado de verificar una propiedad sea una garantía de que no existen fallas en el modelo. Esta técnica se denomina model checking. Como muchos de estos sistemas también interactúan con el mundo real del cual no sabemos a priori que sucederá en cada instante sino que conocemos una distribución de probabilidad, nos centraremos en el model checking probabilístico.

Uno de los obstáculos de dicha herramienta es la explosión de estados al agregar variables al sistema. Es por esto que la investigación y desarrollo de técnicas que nos permitan modelar y verificar modelos de mayor tamaño es de tan importante. Una de estas técnicas es la utilización de estructuras de datos simbólicas para la representación de los modelos que nos permite analizar modelos mas grandes que las técnicas convencionales. Existen numerosas herramientas para la verificación formal simbólica como NuSMV, Prism, Redlib y SMART Model Checker.

Otro de los obstáculos que presenta la técnica de model checking probabilístico es la cantidad de cálculos que deben realizarse para verificar algunas propiedades, lo que se traduce en tiempos de verificación poco prácticos. Es por esto que decidimos investigar la viabilidad de utilizar arquitecturas no convencionales, en nuestro caso arquitecturas GPGPU, para reducir el tiempo de verificación de dichas propiedades y aumentar la eficiencia.

## 1.2. Objetivos del trabajo

Nosotros utilizaremos para este trabajo el model checker simbólico probabilista PRISM. Este es un proyecto de código libre y publicado bajo licencia GNU GPL, lo que nos permite modificar el código agregando nuestra implementación GPGPU modificando solamente los módulos relacionados al cómputo de probabilidades.

Estos módulos implementan algoritmos iterativos para la solución de sistemas de ecuaciones lineales para obtener los resultados cuantitativos de las propiedades, como el método Jacobi, Power y Gauss-Seidel. Centraremos nuestra atención en la implementación Gauss-Seidel ya que esta requiere de un menor número de iteraciones que los otros dos métodos.

La arquitectura GPGPU que utilizaremos será la de CUDA (Compute Unified Device Architecture) desarrollada por la empresa Nvidia. Estos dispositivos poseen una capacidad pico teórica de computo que sobrepasa por ordenes de magnitud a los sistemas convencionales, es por esto que resultan prometedores para el problema que intentamos resolver.

## 1.3. Organización del trabajo

El resto de la tesis esta estructurada de la siguiente manera:

- En el Capítulo 2 se da una introducción al model checking y se presentan los formalismos utilizados para modelar de sistemas y propiedades, así como también una introducción a las estructuras de datos utilizadas para representar dichos modelos.
- En el Capítulo 3 se da una introducción a la arquitectura CUDA, mencionando los componentes principales y las distintas variables que afectan al performance de una aplicación CUDA.
- En el Capítulo 4 se presenta la solución explorada para implementar Gauss-Seidel en CUDA y mencionamos también un método que sobrepasa la restricción de memoria impuesta por estos dispositivos.
- En el Capítulo 5 mostraremos los resultados de los experimentos corridos con nuestra implementación, contrastando los mismos con la implementación CPU y el Gauss-Seidel utilizado por PRISM.

# Capítulo 2

# Introducción al Model Checking

En este capítulo presentaremos los conceptos básicos que necesitamos conocer sobre la técnica de Model Checking para poder entender la utilidad y razones de nuestro trabajo. Comenzaremos analizando sus comienzos y definiendo los primeros modelos utilizados para representar sistemas concurrentes de manera formal, para poder presentar una definición formal del problema. Expondremos luego las lógicas formales probabilísticas PCTL y CSL que son lenguajes matemáticos utilizados para expresar propiedades sobre modelos probabilísticos de tiempo discreto (DTMC) y tiempo continuo (CTMC) respectivamente. Repasaremos la implementación de los operadores probabilísticos de ambas lógicas y analizaremos el algoritmo para la solución de sistemas de ecuaciones lineales Gauss-Seidel, que es utilizado por PRISM en su implementación. Finalizaremos presentando las estructuras de datos de matrices ralas y diagramas binarios de decisión multi-terminales que utilizaremos mas tarde en el Capítulo 3 para nuestra implementación.

## 2.1. Nacimiento del Model Checking

La verificación de modelos o model checking surgió en los años 80 de la necesidad de resolver un problema específico: la verificación de programas concurrentes. Encontrar errores en programas concurrentes es muy difícil debido a que son difíciles de reproducir, por lo que la necesidad de *demostrar formalmente* la pres-

encia (o ausencia) de errores era imperativa. Gran parte de la investigación llevada a cabo para resolver este problema se baso en la construcción de pruebas utilizando una lógica del estilo Floyd-Hoare, pero a fines de los años 70 Pnueli, Owicki y Lamport propusieron el uso de *lógicas temporales* para especificar comportamiento en programas concurrentes.

Las lógicas temporales son útiles para expresar propiedades de safety y liveness, definidas en la Sección 2.2.2, para programas concurrentes. Estas lógicas varian en su nivel de expresividad y en los modelos sobre los cuales pueden ser verificadas. En los años 80 E.M. Clarke y A. Emerson combinaron la técnica de exploración de espacio con la verificación de lógicas temporales, implementando algoritmos polinomiales en el tamaño del modelo. Esto demostró la utilidad de la lógica temporal para casos no triviales, abriendo camino a las primeras aplicaciones comerciales de esta técnica como lo fue la verificación del protocolo de cache "Gigamax" [MS92].

## 2.2. ¿Qué es el Model Checking?

El Model Checking es una técnica de *verificación formal* para determinar de *forma automática* si el *modelo* construido a partir de un sistema finito satisface una propiedad especificada en una *lógica temporal*. Generalmente se resuelve haciendo búsqueda exhaustiva del espacio de estados, ejecutando algoritmos para el análisis de su grafo de transiciones.

Si bien es cierto que hay sistemas complejos de gran escala que funcionan sin la necesidad de esta técnica, en el futuro veremos muchas mas aplicaciones con un dominio mas complejo y con menos tolerancia a errores. Aprender de los errores y luego corregirlos cuesta dinero, y en muchas ocasiones, vidas humanas. Entre los casos mas famosos encontramos a la falla en el sistema de frenos del Toyota Prius, el prematuro inicio de los mecanismos de autodestrucción del Ariane 5 y el fallido sistema de asignación de ambulancias automatizado de Londres, que se estima causo la muerte de entre 20 y 30 personas.

Todas estas pérdidas humanas y económicas se podrían haber evitado utilizando una técnica de verificación formal como esta. Recordemos que el testing, que es normalmente la técnica mediante la cual uno se asegura que un sistema

funciona, solo puede mostrar la presencia de errores, pero no puede asegurar su ausencia.

En lo que resta de esta sección nos concentraremos en definir formalmente el problema que intenta solucionar esta técnica.

### 2.2.1. Estructura de Kripke

Las estructuras de Kripke son grafos cuyos nodos representan los estados alcanzables del sistema y sus aristas representan las transiciones del mismo. Estas estructuras son una variación de los autómatas no-deterministas y fueron propuestas por Saul Kripke para representar el comportamiento de un sistema de manera formal. Su estructura logra capturar la idea de una maquina de computación sin elementos innecesarios. Las estructuras de Kripke se definen formalmente de la siguiente manera:

**Definición:** Sea $AP$ un conjunto de proposiciones atómicas o labels. Una estructura de Kripke es una 4-upla $M = (S, I, R, L)$, donde:

- $S$ es un conjunto de estados.
- $I$ es un conjunto de estados iniciales, tal que $I \subseteq S$.
- $R \subseteq S \times S$ es una relación de transición, donde todo nodo tiene al menos una imagen, i.e. $\forall s \in S : \exists s' S$ tal que $(s, s') \in R$.
- $L \in S \to 2^{AP}$ es una función de rotulación o interpretación.

A continuación definiremos lo que es un camino:

**Definición:** Diremos que un *camino* es una secuencia de estados $\omega = s_0 s_1 ...$, tal que $\forall i \geq 0 : s_i \in S$ y $(s_i, s_{i+1}) \in R$.

Dado un camino, verificar si cumple cierta propiedad expresada en una lógica temporal es bastante simple; solo basta con analizar de a un estado a la vez. Pero cuando queremos verificar una propiedad en un modelo, necesitamos que la misma se verifique para todos los caminos infinitos posibles dentro del mismo que comiencen desde un estado en $I$. Debido a la manera en que definimos $R$, va

a ser siempre posible construir un camino infinito en esta estructura. Podemos modelar los estados deadlocks, donde no hay más transiciones a otros estados, con un nodo que posee una sola arista saliente a si mismo y con una etiqueta distinguida.

### 2.2.2. Lógicas Temporales

Una lógica temporal es un lenguaje formal para especificar y razonar sobre el comportamiento de un sistema a lo largo del tiempo. Se extienden de la lógica proposicional con operadores modales. En estas lógicas podemos expresar proposiciones muy útiles para la verificación de sistemas como "siempre que un cliente pida un recurso, eventualmente se le concederá"y "nunca se concederá el mismo recurso a dos clientes al mismo tiempo".

### 2.2.3. CTL

CTL (o Computation Tree Logic), es una lógica que modela el tiempo representandolo en una estructura de arbol en el cual el futuro no esta determinado; existen diferentes caminos que pueden tomarse en un estado dado y cualquiera de estos es posible.

Su gramática es la siguiente:

$$\phi ::= true \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \mathcal{A}\mathcal{X}\phi \mid \mathcal{A}(\phi \mathcal{U} \phi) \mid \mathcal{E}(\phi \mathcal{U} \phi)$$

Estas fórmulas se evaluan sobre un estado y permíten expresar propiedades sobre los caminos posibles a partir de dicho estado. Podemos explicar informalmente su semántica de la siguiente manera:

- $\mathcal{A}\mathcal{X}\phi$: el siguiente paso de todos los caminos posibles cumple con $\phi$.
- $\mathcal{A}(\phi_1 \mathcal{U} \phi_2)$: todos los caminos posibles (posiblemente vacíos) cumplen con $\phi_1$ hasta que se cumpla $\phi_2$.
- $\mathcal{E}(\phi_1 \mathcal{U} \phi_2)$: algún caminos posible (posiblemente vacío) cumple con $\phi_1$ hasta que se cumpla $\phi_2$.

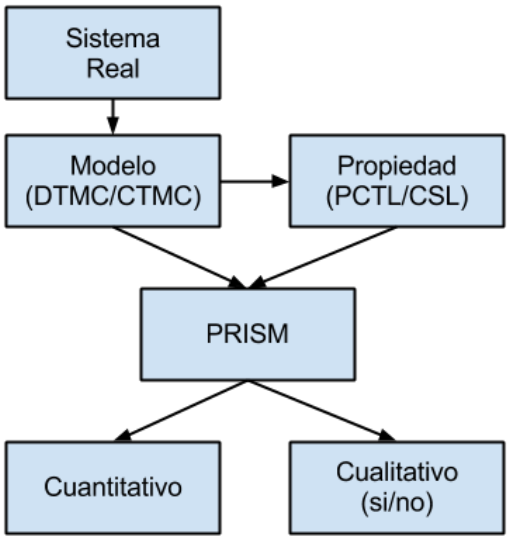



**Figura 2.1:** Flujo de trabajo en PRISM

### 2.2.4. Definición Formal del Problema

**Definición:** Sea $M$ una estructura de Kripke, sea $f$ una fórmula en una lógica temporal. Encontrar todos los estados $s$ de $M$ que satisfacen $M, s \models f$.

Para la verificación de propiedades sobre estructuras de Kripke solo necesitamos correr algoritmos de orden polinomial en el tamaño de nuestro modelo para obtener un resultado. Estos algoritmos pertenecen a la clase de algoritmos para el análisis de grafos, como ser componentes fuertemente conexas, orden topológico, etc. En los casos probabilísticos, que veremos en las siguientes secciones, suele ser necesario computar valores de probabilidad que interrelacionan los estados del modelo y por lo tanto forman un sistema de ecuaciones lineales. Para dichos casos utilizaremos algoritmos adecuados para resolver estos sistemas de ecuaciones lineales y será nuestro trabajo acelerar dichos algoritmos utilizando arquitecturas GPUs.

### 2.2.5. Verificación de Modelos con PRISM

PRISM es un model checker creado por David Parker [Par02] en la Universidad de Birmingham. Nos basamos en su implementación debido a que es uno de los

pocos model checkers probabilísticos de código abierto con una viva comunidad científica.

En la Figura 2.1 podemos apreciar la manera en la que se utiliza esta herramienta. Se parte de un sistema en el mundo real, del cual creamos un modelo finito utilizando una lógica de nuestra elección. Las lógicas para la creación de modelos sobre las que trabajaremos serán, como ya mencionamos, las de tiempo discreto y tiempo continuo. Una vez creado el modelo, formulamos la propiedad a verificar en la lógica correspondiente a nuestro modelo. Por último, alimentamos a PRISM con nuestro modelo y nuestra propiedad para obtener un resultado cualitativo o cuantitativo según corresponda.

La verificación de algunas de las propiedades requiere de miles de millones de operaciones aritméticas y en conjunto pueden llegar a demorar varios días en completarse. Esto significa muchas veces que simplificaremos el modelo o relajaremos la propiedad para obtener una respuesta más pronto. Nuestro objetivo es poder verificar propiedades más complejas sobre modelos más grandes y para eso utilizaremos el poder de cómputo de las GPUs.

## 2.3. Verificación de Modelos Probabilísticos

El Model Checking probabilístico es una técnica de verificación formal para modelar y analizar sistemas que exhiben *comportamiento probabilístico*, permitiendo hacer un análisis *cualitativo* y *cuantitativo* del mismo.Algunos sistemas son inherentemente probabilísticos, por lo que la capacidad de extender la técnica original habilita su análisis y verificación.

En esencia lo que haremos será modificar la función de transición definida para las estructuras de Kripke de forma tal que las transiciones sean probabilísticas. Definiremos un par de restricciones y libertades para cada caso, pero seguirán siendo muy similares a las estructuras de Kripke originales.

Los modelos probabilísticos que analizaremos son variantes de las cadenas de Markov. Por lo que tiene la propiedad de ser *memoryless*, eso significa que cada transición solo depende del estado actual y no de como llegó hasta ahí.

## 2.4. Cadenas de Markov de Tiempo Discreto

**Definición:** Sea $AP$ un conjunto de proposiciones atómicas o labels. Una *cadena de Markov de tiempo discreto* sobre un conjunto de proposiciones atómicas $AP$ es una 4-upla $M = (S, s_{init}, \mathbf{P}, L)$, Donde:

- $S$ es un conjunto de estados.
- $s_{init} \in S$ es un estado inicial.
- $\mathbf{P} : S \times S \to [0, 1]$ es la *matriz de probabilidad de transiciones* tal que $\forall s \in S : \sum_{s' \in S} \mathbf{P}(s, s') = 1$.
- Una función de rotulación o interpretación $L \in S \to 2^{AP}$.

Notar que $\mathbf{P}$ es lo que se denomina una *matriz estocástica* y que como cada estado debe tener una función de distribución de probabilidad, a los estados *absorbentes* que no poseen aristas salientes a otros estados les asignaremos una arista a si mismos definiendo $\mathbf{P}(s,s) = 1$.

**Definición:** Diremos que un *camino* es una secuencia de estados $\omega = s_0 s_1 s_2..$ tal que $\forall i \geq 0$ $\mathbf{P}(s_i, s_{i+1}) > 0$. Donde: $\omega[..i]$ denotará el prefijo finito terminando en el estado $(i+1)$, $\omega[i..]$ denotará el sufijo infinito que comienza en el estado $(i+1)$, $\omega(i)$ denotará el $(i+1)$-ésimo estado y $Path_s$ denotará el conjunto de todos los caminos infinitos que parten de $s$.

Dado que un camino representa una evolución del modelo, para poder razonar sobre las probabilidades del sistema debemos poder medir las probabilidades de tomar ciertos caminos. Por consiguiente para cada estado $s$ definimos la medida de probabilidad $Prob_s$ sobre $Path_s$. $Prob_s$ se infiere a partir de la matriz $\mathbf{P}$ de la siguiente manera. Primero definimos la probabilidad $P(\omega_{fin})$ de un camino finito de la siguiente manera.

$$P(\omega_{fin}) = \begin{cases} 1 & \text{si } \omega_{fin} \text{ contiene solamente al estado } s_0 \\ \prod_{i=0}^{n-1} \mathbf{P}(s_i, s_{i+1}) & \text{si } \omega_{fin} \text{ es igual a } s_0 s_1..s_n \end{cases}$$

Seguido esto, definimos el *conjunto cilindro* $C(\omega_{fin})$, correspondiente a un camino finito $\omega_{fin}$, como el conjunto de todos los caminos que posean a $\omega_{fin}$ como prefijo. Luego, sea $\Sigma_s$ la menor $\sigma$-álgebra sobre $Path_s$, que contiene a todos los conjuntos $C(\omega_{fin})$ donde $\omega_{fin}$ pertenece al conjunto de todos los caminos finitos que parten de $s$. Definiremos la medida de probabilidad $Prob_s$ sobre $\Sigma_s$ como la única medida que satisface $Prob_s(C(\omega_{fin})) = P(\omega_{fin})$. Ahora podemos cuantificar la probabilidad de que un DTMC se comporte de una manera determinada al identificar el conjunto de caminos con ese comportamiento y utilizar la medida apropiada $Prob_s$.

Estos modelos nos permiten trabajar con tiempos discretos y distribuciones de probabilidad discretas, por lo que nos permiten modelar correctamente sistemas donde las unidades de tiempo en las cuales cambia de estado son discretas como los sistemas electrónicos embebidos en donde el cambio de estado se da en base a un pulso o clock, y sistemas en los cuales el tiempo es abstracto.

## 2.5. Cadenas de Markov de Tiempo Continuo

Cuando precisamos de un lenguaje de modelado que nos permita expresar modelos en los que el tiempo es continuo, es decir que las transiciones se pueden dar en cualquier instante de tiempo, necesitaremos utilizar los CTMC. Este modelado de transiciones en tiempo continuo se logra utilizando la distribución exponencial negativa para modelar los tiempos entre eventos de un proceso de Poisson, en el cual los eventos son las transiciones del sistema. Esto lo podemos interpretar de la siguiente manera: si estamos situados en el nodo $s$, todas las transiciones que salen de $s$ se simulan y luego se ejecuta la que sucedió primero en dicha simulación.

**Definición:** Sea $AP$ un conjunto de proposiciones atómicas o labels. Una *cadena de Markov de tiempo continuo* sobre un conjunto de proposiciones atómicas AP es una 4-upla $M = (S, s_{init}, \mathbf{R}, L)$, Donde:

- $S$ es un conjunto de estados.
- $s_{init} \in S$ es un estado inicial.

- $\mathbf{R} : S \times S \to \mathbb{R}_{\geq 0}$ es la matriz de *frecuencia de transiciones*.
- $L \in S \to 2^{AP}$ es una función de rotulación o interpretación.

Ahora en lugar de una matriz de probabilidad de transiciones tenemos una matriz de frecuencia de transiciones que será utilizada como parámetro para una distribución exponencial negativa. Hay una transición de $s$ a $s'$ siempre que $\mathbf{R}(s, s') > 0$ y la probabilidad de que el evento ocurra antes de $t$ unidades de tiempo es igual a $1 - e^{-\mathbf{R}(s,s')\cdot t}$. Notar que no hay mas restricciones sobre los nodos que no tienen aristas salientes a otro nodo.

**Definición:** Un *camino* en un CTMC sera una secuencia no vacía $\omega = s_0 t_0 s_1 t_1 ...$ donde $\mathbf{R}(s_i, s_{i+1}) > 0$ y $t_i \in \mathbb{R}_{>0}$ para todo $i \geq 0$. El valor $t_i$ representa el tiempo que se detuvo en $s_i$. Al igual que en los DTMC denotaremos $\omega(i)$ al $i$-ésimo estado del camino $\omega$. Adicionalmente llamaremos $\omega@t$ al estado en el camino al instante de tiempo $t$, i.e. $\omega(k)$ donde $k$ es el menor índice que cumple $\sum_{i=0}^{k} t_i \geq t$.

El tiempo de espera en un estado antes de avanzar a una transición es el mínimo de las distribuciones exponenciales asociadas al mismo, por lo que definiendo $E(s) = \sum_{s' \in S} \mathbf{R}(s, s')$ la probabilidad de dejar el estado $s$ en el intervalo $[0, t]$ es de $1 - e^{-E(s)\cdot t}$. Por eso decimos que $E(s)$ es la frecuencia de salida de $s$. Llamaremos *absorbentes* a los estados en los que $E(s) = 0$.

Debido a que las transiciones son un conjunto de variables exponenciales negativas, la siguiente transición es independiente del tiempo en el cual ocurre. Podemos verificar esto ya que si $X_1$ y $X_2$ son variables exponenciales con parámetros $\lambda_1$ y $\lambda_2$ respectivamente, la probabilidad de que $X_1$ ocurra antes que $X_2$, $P(X_1 < X_2)$, es igual a $\frac{\lambda_1}{\lambda_1 + \lambda_2}$. Debido a esto podemos generar un DTMC subyacente al CTMC dado, definiendo la matriz de probabilidades del nuevo modelo $\mathbf{P}^{emb(C)}$ como:

$$\mathbf{P}^{emb(C)}(s, s') = \begin{cases} \mathbf{R}(s, s')/E(s) & \text{if } E(s) > 0 \\ 1 & \text{si } E(s) = 0 \text{ y } s = s' \\ 0 & \text{caso contrario} \end{cases}$$

Nuevamente denotaremos $Path_s$ al conjunto de caminos que parten de $s$. A continuación definiremos la medida de probabilidad $Prob_s$ sobre $Path_s$. Si

los estados $s_0$, ... $s_n \in S$, satisfacen que $\forall i \geq 0.\mathbf{R}(s, s') > 0$ y $I_0$, .. $I_n$ son subintervalos no vacíos de $\mathbb{R}_{\geq 0}$, entonces el *conjunto cilindro* $C(s_0, I_0, .., I_{n-1}, s_n)$ sera el conjunto que contiene a todos los caminos $s'_0 t'_0 s'_1 t'_1 s'_2$.. donde $s'_i = s_i$ para $i \leq n$ y $t_i \in I_i$ para $i < n$.

Luego sea $\sum_s$ la $\sigma$-álgebra mas pequeña sobre $Path_s$ que contiene a todos los conjuntos cilindro $C(s_0, I_0, .., I_{n-1}, s_n)$ donde $s_0 s_1 .. s_n \in S$ con $s_0 = s$ y $R(s_i, s_{i+1}) > 0$ para $0 \leq i < n$, y $I_0, .., I_{n-1}$ varia sobre todas las secuencias no vacias de intervalos de $\mathbb{R}_{\geq 0}$. La medida de probabilidad $Prob_s$ sobre $\sum_s$ es entonces la única medida definida inductivamente por $Prob_s(s_0) = 1$ y $Prob_s(s_0, ...s_n, I_n, s_{n+1})$ igual a:

$$Prob_s(C(s_0, .., s_n)) \cdot \mathbf{P}^{emb(C)}(s_n, s_{n+1}) \cdot (e^{-\inf I_n \cdot \sum_{s' \in S} \mathbf{Q}(s_n, s')} - e^{-\sup I_n \cdot \sum_{s' \in S} \mathbf{Q}(s_n, s')})$$

Esto nos da lugar a dos interpretaciones de un CTMC: como **eventos concurrentes**, en la que cada evento se simula con una exponencial y el siguiente estado es el que ocurre antes, o como modelando la **separación entre demora y transición** para cada estado, donde el tiempo detenido en un estado esta dado por la exponencial negativa de parámetro $E(s)$ y la transición esta dada por una distribución de probabilidad $\mathbf{P}^{emb(C)}$.

Consideraremos además otras dos propiedades de los CTMC: comportamiento *transitorio*, que se relaciona al estado del modelo en un instante determinado y comportamiento de *estado-estable*, que logra capturar el estado de un CTMC a lo largo de una corrida infinita. La probabilidad transitoria $\pi_{s\,t}(s')$ se define como la probabilidad de, al haber arrancado en el estado $s$, encontrarse en el estado $s'$ al instante $t$. La probabilidad de estado-estable de $\pi_s(s')$ se define como $\lim_{t \to \infty} \pi_{s\,t}(s')$. Para los CTMC que son *ergódicos*, como los representables por PRISM, se ha demostrado que la probabilidad de estado-estable no depende del nodo inicial, por lo que solo hace falta calcularla para un nodo cualquiera del sistema.

Con esta definición hemos aumentado la expresividad, pudiendo ahora abarcar modelos de colas, reacciones químicas, procesos biológicos, etc.

## 2.6. Lógicas Temporales Utilizadas

La lógica que vimos en 2.2.3 sirve para modelos no probabilistas, nosotros utilizaremos PCTL o *Probabilistic CTL* para los modelos DTMC y CSL o *Continuous Stochastic Logic* para CTMC.

### 2.6.1. PCTL

PCTL es la lógica utilizada para expresar propiedades a verificar sobre un DTMC. A continuación definiremos su gramática:

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

donde $a$ es una proposición atómica que pertenece a $AP$, $\bowtie \in \{\leq, <, \geq, >\}$, $p \in [0, 1]$ y $k \in \mathbb{N}$ . Las propiedades de los modelos serán definidas siempre como propiedades de estados. Las fórmulas sobre caminos pueden aparecer solamente dentro del *operador de probabilidad de caminos* $\mathcal{P}_{\bowtie p}[\psi]$. Intuitivamente, un estado $s$ satisface $\mathcal{P}_{\bowtie p}[\psi]$ si la probabilidad de tomar un camino partiendo desde $s$ que satisfaga $\psi$ se encuentra en el intervalo especificado por $\bowtie p$.

Como fórmulas de caminos utilizaremos los operadores $\mathcal{X}$ (*next*), $\mathcal{U}^{\leq k}$ (*bounded until*) y $\mathcal{U}$ (*until*) utilizados normalmente en las lógicas temporales. Intuitivamente, $\mathcal{X}\phi$ es verdadera si $\phi$ se cumple en el siguiente estado; $\phi_1 \mathcal{U}^{\leq k} \phi_2$ es verdadera si $\phi_2$ se cumple luego de a lo sumo $k$ pasos y $\phi_1$ se cumple hasta ese punto; $\phi_1 \mathcal{U} \phi_2$ es verdadera si $\phi_2$ se cumple en algún momento en el futuro y $\phi_1$ es verdadera hasta entonces.

#### Semántica de PCTL sobre DTMC

Para un DTMC ($S$, $s_{ini}$, $\mathbf{P}$, $L$), estado $s \in S$ y una fórmula PCTL $\phi$, diremos que $s \models \phi$ para indicar que $s$ satisface $\phi$. Análogamente diremos que para un camino $\omega$ y una fórmula de camino PCTL $\psi$, $\omega \models \psi$ si $\psi$ se cumple para $\omega$. Ahora podemos dar la semántica denotacional para formulas de estados y caminos de PCTL sobre DTMC.

Dado un camino $\omega$:

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

Dado un estado $s \in S$ :

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

Donde $p_s(\psi) = Prob_s(\omega \in Path_s | \omega \models \psi)$.

## Operadores Adicionales

Podemos a partir de los operadores de lógica introducidos, definir operadores adicionales para facilitar su lectura y manipulación.

$$\begin{aligned}
false &\equiv \neg true \\
(\phi_1 \vee \phi_2) &\equiv \neg(\neg\phi_1 \wedge \neg\phi_2) \\
(\phi_1 \rightarrow \phi_2) &\equiv \neg\phi_1 \vee \phi_2
\end{aligned}$$

También permitiremos a las formulas tener el operador $\diamondsuit$ (finalmente) que es habitual en las lógicas temporales. Intuitivamente el operador $\diamondsuit\phi$ expresa que $\phi$ se hace verdadera en algún momento en el futuro. También consideraremos a su variante $\diamondsuit^{\leq k}\phi$ que expresa que $\phi$ será verdadera en a lo sumo $k$ pasos. Todos

estos operadores se pueden expresar en PCTL de la siguiente manera:

$$\Diamond\phi \equiv \text{ true } \mathcal{U}\phi$$
$$\Diamond^{\leq k}\phi \equiv \text{ true } \mathcal{U}^{\leq k}\phi$$

Este operador, análogo al cuantificador $\exists$, cuenta con su contraparte análoga a $\forall$ que se define como $\Box\phi = \neg\Diamond\neg\phi$, pero dado que no definimos la negación de caminos, no podemos derivarla de la sintaxis de PCTL. Sin embargo podemos intuitivamente darle significado a $\mathcal{P}_{\bowtie p}(\neg\psi)$ interpretándolo como $\mathcal{P}_{\neg\bowtie p}(\psi)$. Donde $\neg\bowtie p$ seria el intervalo correspondiente al complemento de $\bowtie p$ acotado a [0, 1], dado que $p$ representa una probabilidad.

Notar que hay algunas propiedades útiles de los DTMC que la lógica PCTL no nos permite expresar, de hecho es bastante limitada. Para poder expresar dichas propiedades en los DTMC deberíamos utilizar una lógica como LTL (*Linear Time Temporal Logic*). Por ejemplo podríamos computar la probabilidad del conjunto de caminos que cumplen $\Diamond\phi_1 \wedge \Diamond\phi_2$, donde $\phi_1$ y $\phi_2$ se cumplen eventualmente, pero no necesariamente al mismo tiempo. Notar que esta fórmula no puede derivarse de las formulas $\Diamond\phi_1$ y $\Diamond\phi_2$ por separado.

Podríamos obtener mas expresividad utilizando PCTL$^*$ que combina PCTL y LTL, pero esto incrementaría el costo de la verificación de probabilidades de forma exponencial en el tamaño de la fórmula.

### 2.6.2. CSL

La lógica CSL (Continuous Stochastic Logic) es similar a la lógica PCTL, pero fue diseñada para especificar propiedades sobre los modelos CTMC. CSL provee una forma de describir propiedades de estado-estable y comportamiento transitorio, que son tradicionales en el análisis de los modelos CTMC. Su sintáxis es la siguiente:

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

donde $a$ es una proposición atómica que pertenece a $AP$, $\bowtie \in \{\leq, <, \geq, >\}$ y $t \in \mathbb{R}_{\geq 0}$.

Al igual que en PCTL, $\mathcal{P}_{\bowtie p}[\psi]$ indica la probabilidad de que la fórmula de camino $\psi$ sea verdadera para un camino con una probabilidad $\bowtie p$. Las formulas de camino son iguales que las de PCTL con la diferencia de que $t$ es ahora un real no negativo. En consecuencia, la fórmula $\phi_1 \mathcal{U}^{\leq t} \phi_2$ expresa la probabilidad de que un camino cumpla $\phi_2$ en un instante de tiempo $t'$ tal que $t' \in [0, t]$ y cumpla $\phi_1$ todos los instantes anteriores.

El nuevo operador $\mathcal{S}_{\bowtie p}[\phi]$, expresa el comportamiento de estado-estable de un CTMC. Intuitivamente esta fórmula es verdadera si la probabilidad de estar en un estado en una *corrida larga* que cumpla $\phi$ es $\bowtie p$.

## Semántica de CSL sobre CTMC

Al igual que para PCTL escribiremos $s \models \phi$ para indicar que la fórmula CSL $\phi$ se satisface en el estado $s$ de un CTMC y llamaremos $Sat(\phi)$ al conjunto $\{s \in S | s \models \phi\}$. Similarmente para una fórmula $\psi$ que es verdadera para un camino $\omega$, escribiremos $\omega \models \psi$. A continuación definiremos su semántica denotacional.

Dado un camino $\omega$:

$$
\begin{aligned}
\omega &\models \mathcal{X}\phi && \iff \omega(1) \models \phi \\
\omega &\models \phi_1 \mathcal{U}^{\leq t} \phi_2 && \iff \exists x \in [0,t].(\omega@x \models \phi_2 \wedge (\forall y \in [0,x) : \omega@y \models \phi_1)) \\
\omega &\models \phi_1 \mathcal{U} \phi_2 && \iff \exists t \geq 0.\omega \models \phi_1 \mathcal{U}^{\leq t} \phi_2
\end{aligned}
$$

Dado un estado $s \in S$ :

$$
\begin{aligned}
& s \models true && \forall s \in S \\
& s \models a && \iff a \in L(s) \\
& s \models \phi_1 \wedge \phi_2 && \iff s \models \phi_1 \wedge s \models \phi_2 \\
& s \models \neg\phi && \iff s \not\models \phi \\
& s \models \mathcal{P}_{\bowtie p}[\psi] && \iff p_s(\psi) \bowtie p \\
& s \models \mathcal{S}_{\bowtie p}[\phi] && \iff \sum_{s' \models \phi} \pi_{s_{ini}}(s') \bowtie p
\end{aligned}
$$

Donde $p_s(\psi) = Prob_s(\omega \in Path_s | \omega \models \psi)$ y $\pi_{s_{ini}}$ es la función de probabilidad de estado-estable partiendo del estado inicial.

### Operadores Adicionales

Al igual que para PCTL se pueden derivar los operadores $false$, $\vee$, $\rightarrow$ y $\diamondsuit$. Sin embargo debemos notar la diferencia en cuanto al operador $\diamondsuit^{\leq t}$, pues ahora $t \in \mathbb{R}_{\geq 0}$.

## 2.7. Implementación de los Operadores Lógicos

En esta sección deremos una introducción a los algoritmos de model checking para determinar si una DTMC o CTMC satisface una fórmula PCTL o CSL respectivamente utilizado por PRISM. Por el momento dejaremos de lado las estructuras de datos utilizadas para su implementación real, presentando estas operaciones de forma abstracta y luego indagaremos en aquellas que nos sean pertinentes.

Básicamente, el algoritmo de model checking para PCTL toma un modelo dado en términos de una DTMC y una fórmula PCTL $\phi$ y retorna un conjunto de estados $Sat(\phi)$ que son los estados que satisfacen $\phi$. Similarmente el algoritmo de model checking para CSL toma un modelo dado como una CTMC y la fórmula CSL $\phi$ a verificar y retorna el conjunto de estados $Sat(\phi)$ que satisface $\phi$

La estructura de los algoritmos es la misma para los dos casos, se parte de una fórmula $\phi$ y se genera el árbol de subexpresiones. Las hojas de este árbol son o bien *true* o bien una proposición atómica $a$. Se computa desde las hojas hasta la raíz de forma recursiva los conjuntos de estados que satisfacen las subfórmulas y al finalizar tendremos el conjunto de estados que satisfacen $\phi$.

La verificación de operadores no probabilísticos de ambas lógicas se realiza de forma idéntica para DTMC y CTMC. Dado una proposición atómica $a$, $Sat(a) = \{s : a \in L(s)\}$ y su cálculo se desprende directamente del modelo. Los operadores $\wedge$ y $\neg$ se interprentan respectivamente como la interesección y el complemento de conjuntos. Los casos mas complejos corresponden a los operadores $\mathcal{P}$ y $\mathcal{S}$, en estos casos es necesario primero obtener las probabilidades de satisfacer la fórmula para cada estado y luego computar los estados que satisfacen la condición $\bowtie p$. El calculo de probabilidades para cada operación sera explicado en las siguientes subsecciones y forma parte fundamental de nuestro trabajo, ya que es la parte que hemos acelerado con las GPUs.

La complejidad de estas operaciones es en el peor caso polinomial en el tamaño del modelo, por lo tanto la computación de los operadores de PCTL y CSL es lineal en el tamaño de la fórmula y polinomial en el tamaño del modelo.

### 2.7.1. PCTL Model Checking Sobre DTMC

Para verificar una fórmula PCTL $\mathcal{P}_{\bowtie p}[\psi]$ sobre un DTMC $M = (S, s_{init}, \mathbf{P}, L)$ necesitamos computar la probabilidad de que un camino que comience en $s$ satisfaga la fórmula de camino $\psi$. En los casos en los que $\psi$ sea el operador PCTL next ($\mathcal{X}\phi$), bounded until ($\phi_1 \mathcal{U}^{\leq k} \phi_2$), o until ($\phi_1 \mathcal{U} \phi_2$), calcularemos para todos los estados $s \in S$ la probabilidad $p_s(\mathcal{X}\phi)$, $p_s(\phi_1 \mathcal{U}^{\leq k} \phi_2)$ y $p_s(\phi_1 \mathcal{U} \phi_2)$ respectivamente. Luego computaremos $Sat(\mathcal{P}_{\bowtie p}(\psi))$ como $\{s \in S | p_s(\psi) \bowtie p\}$.

Debido a la naturaleza recursiva de estos algoritmos podemos asumir que los conjuntos $Sat(\phi)$, $Sat(\phi_1)$ y $Sat(\phi_2)$ son conocidos.

#### PCTL Next

$p_s(\mathcal{X}\phi)$ es la probabilidad de que estando en el estado $s$, la transición lleve a un estado $s'$ que cumpla con $\phi$. Por consiguiente $p_s(\mathcal{X}\phi) = \sum_{s' \in Sat(\phi)} \mathbf{P}(s, s')$. Por

lo que si tenemos un vector $\underline{\phi}$ donde $\underline{\phi}(s) = 1$ sii $s \models \phi$, podemos computar el vector de probabilidades para todos los estados $\underline{p(\mathcal{X}\phi)}$ como $\underline{p(\mathcal{X}\phi)} = \mathbf{P} \cdot \underline{\phi}$. Esto se puede computar con una sola multiplicación matriz-vector.

### PCTL Bounded Until

$p_s(\phi_1 \mathcal{U}^{\leq k} \phi_2)$ es la probabilidad de que partiendo del estado $s$ en al menos $k$ transiciones se llegue a un estado que satisface $\phi_2$ y hasta ese momento se cumplió $\phi_1$ en todos los estados intermedios. Para poder calcular esta probabilidad dividiremos los estados en tres conjuntos disjuntos $S^{yes}$, $S^{no}$ y $S^{?}$. Estos serán los conjuntos de estados que satisfacen la fórmula trivialmente, los que no la pueden satisfacer y los que podrían satisfacerla respectivamente. Calculamos estos conjuntos como $S^{yes} = Sat(\phi_2)$, $S^{no} = S \setminus (Sat(\phi_1) \cup Sat(\phi_2))$ y el conjunto $S^{?} = S \setminus (S^{yes} \cup S^{no})$ que contiene todos los estados para los cuales debemos calcular las probabilidades. Para estos estados tenemos:

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

Para poder computar esta probabilidad, definiremos la matriz $\mathbf{P'}$ de la siguiente manera:

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

y abreviaremos el vector de probabilidades $\underline{p(\phi_1 \mathcal{U}^{\leq k} \phi_2)}$ como $\underline{p_k}$. A continuación veremos como calcularlo. Para $k = 0$, $\underline{p_0}(s) = 1$ sii $s \in S^{yes}$ y 0 en caso contrario. Para $k > 0$ tenemos que $p_k = \mathbf{P'} \cdot p_{k-1}$. En total esta operación requiere $k$ multiplicaciones matriz-vector.

### PCTL Until

Al igual que para el operador bounded until, podemos identificar los estados en los cuales $p_s(\phi_1 \mathcal{U} \phi_2)$ es inmediatamente 0 o 1. Pero para este caso nos resulta útil extender los conjuntos $S^{yes}$ y $S^{no}$ para todos los estados en los cuales es

exactamente 0 o 1. Recordemos que antes solo calculábamos los estados para los cuales la fórmula era inicialmente válida o no, al extender estos conjuntos obtendremos también los estados cuyas transiciones nos llevan con probabilidad 1 a otros estados que satisfacen la fórmula dada ($S^{yes}$) o con probabilidad 1 a los que nunca transicionan a un estado que satisfaga la fórmula ($S^{no}$).

Los algoritmos para extender estos conjuntos fueron presentados en [Par02] y son algoritmos de punto fijo. Estos dos algoritmos de precomputo son eficientes para analizar propiedades cualitativas donde la probabilidad $\bowtie p$ es 0 o 1, o cuando estos conjuntos abarcan completamente a $S$. Para esos casos no hace falta ningún método numérico para la computación de probabilidades. Otra ventaja de utilizar estos métodos analíticos, es que estamos reduciendo el error que podría acarrearse si utilizáramos métodos numéricos.

Para computar las probabilidades de los estados en $S^{?} = S \setminus (S^{yes} \cup S^{no})$ y calcular $p_s(\phi_1 \mathcal{U} \phi_2)$, podemos resolver el siguiente sistema de ecuaciones en las variables $\{x_s | s \in S\}$:

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

finalmente asignando $p_s(\phi_1 \mathcal{U} \phi_2) = x_s$. Para reescribirlo en la forma tradicional $\mathbf{A} \cdot \underline{x} = \underline{b}$, asignaremos $\mathbf{A} = \mathbf{I} - \mathbf{P'}$ donde $\mathbf{I}$ es la matriz identidad y $\mathbf{P'}$ esta dada por:

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

y $\underline{\text{b}}$ es la columna sobre estados donde $\underline{\text{b}}(s)$ es 1 si $s \in S^{yes}$ y 0 en caso contrario. El sistema $\mathbf{A} \cdot \underline{x} = \underline{b}$ puede ser resuelto con cualquier algoritmo estándar, como métodos directos, Jacobi y Gauss-Seidel. Como intentamos resolver sistemas muy grandes nos centraremos en los métodos iterativos ya que como explicaremos en las siguientes secciones son los mas apropiados para nuestras estructuras de datos.

### 2.7.2. CSL Model Checking Sobre CTMC

Los operadores next y until dependen solamente de la probabilidad de la transición y no del tiempo en el que ocurren, por lo que se pueden verificar utilizando los algoritmos ya mencionados sobre el DTMC subyacente. Por lo que en esta sección consideraremos solamente los otros dos operadores que nos quedan por definir, el time-bounded until y el steady-state.

#### CSL Time-Bounded Until

Para este operador debemos calcular la probabilidad $p_s(\phi_1 \mathcal{U}^{\leq t} \phi_2)$ para todos los estados $s$. Al igual que en los casos anteriores, la probabilidad para los estados que satisfacen $\phi_2$ es trivialmente 1. Reutilizaremos el algoritmo presentado en [Par02] para calcular los estados que no pueden satisfacer $\phi_2$ pasando solamente por estados que satisfacen $\phi_1$. El resto de las probabilidades deben ser computadas numéricamente.

La idea básica es modificar el CTMC, eliminando todas las transiciones salientes de los estados que satisfacen $\phi_2$ o están en $S^{no}$. Dado que todo camino en el nuevo CTMC no puede salir de un estado satisfaciendo $\phi_2$ ni tampoco de un estado en $S^{no}$, la probabilidad $p_s(\phi_1 \mathcal{U}^{\leq t} \phi_2)$ en el CTMC original es igual a la probabilidad de estar en un estado que satisfaga $\phi_2$ en el instante de tiempo $t$ en el nuevo CTMC, i.e. $\sum_{s' \models \phi_2} \pi_{s,t}(s')$, por lo que el problema se reduce a computar la probabilidad transitoria $\pi_{s,t}(s')$ para todos los estados $s$ y $s'$ del nuevo CTMC.

Para esto se utiliza una técnica llamada *uniformization*. A partir de la matriz *generadora* $\mathbf{Q}$ definida como $\mathbf{Q}(s,\ s') = \mathbf{R}(s,\ s')$ si $s \neq s'$ y $\mathbf{Q}(s,\ s') = -\sum_{s'' \neq s} \mathbf{R}(s, s'')$ si $s = s'$, computamos el $\mathbf{P}$ del DTMC *uniformizado*, dado por $\mathbf{P} = \mathbf{I} + \mathbf{Q}/q$, donde $q \geq \max_{s \in S} |\mathbf{Q}(s, s)|$. Escribiendo el vector de probabilidades transitorias $\pi_{s,t}(s')$ sobre todos los estados $s' \in S$ como $\underline{\pi_{s,t}}$, obtenemos:

$$\underline{\pi_{s,t}} = \underline{\pi_{s,0}} \cdot \sum_{i=0}^{\infty} \gamma_{i,q \cdot t} \cdot \mathbf{P}^i$$

donde $\gamma_{i,q \cdot t}$ es la $i$-ésima probabilidad de Poisson con parámetro $q \cdot t$, i.e. $\gamma_{i,q \cdot t} = e^{-q \cdot t} \cdot (q \cdot t)^i / i!$ y representa la probabilidad de haber realizado $i$ saltos de estado en $t$ unidades de tiempo. El vector $\underline{\pi_{s,0}}$ nos da la probabilidad de estar cada estado

en tiempo $t = 0$. Dado que comenzamos en el estado $s$ tenemos que $\underline{\pi_{s,0}}(s') = 1$ sii $s = s'$ y 0 en caso contrario. La suma ponderada de las potencias de $\mathbf{P}$ nos da la matriz de probabilidades de transición de un estado dado del CTMC a cualquier otro en $t$ unidades de tiempo.

Para computar las probabilidades transitorias numéricamente, podemos truncar la suma infinita para una precisión deseada $\epsilon$. Haciendo esto obtenemos dos limites $L_\epsilon$ y $R_\epsilon$, sobre los cuales debemos hacer la suma de potencias de $\mathbf{P}$.

Como ha sido demostrado en [KKNP01], la uniformización puede ser adaptada para calcular el operador de CSL time bounded-until. Sea $\underline{p(\phi_1 \mathcal{U}^{\leq t} \phi_2)}$ el vector de probabilidades requerido para calcular $p_s(\phi_1 \mathcal{U}^{\leq t} \phi_2)$ para cada estado $s$. Y sea el vector $\underline{\phi_2}$ tal que $\underline{\phi_2}(s) = 1$ si $s \models \phi_2$ y 0 para el caso contrario, tenemos que:

$$\begin{aligned} \underline{p(\phi_1 \mathcal{U}^{\leq t} \phi_2)} &= (\sum_{L_\epsilon}^{R_\epsilon} \gamma_{i,q\cdot t} \cdot \mathbf{P}^i) \cdot \underline{\phi_2} \\ &= \sum_{L_\epsilon}^{R_\epsilon} (\gamma_{i,q\cdot t} \cdot \mathbf{P}^i \cdot \underline{\phi_2}) \end{aligned}$$

Vale mencionar que la inclusión de $\underline{\phi_2}$ dentro de los paréntesis es lo que nos permite evitar el cálculo de las sucesivas potencias $\mathbf{P}^i$. Lo que hacemos en su lugar es computar cada producto $\mathbf{P}^i \cdot \phi_2$ como $\mathbf{P} \cdot (\mathbf{P}^{i-1} \cdot \phi_2)$ reutilizando en cada paso la computación anterior, por lo que se reduce el número de operaciones matriz-vector a $R_\epsilon$-$L_\epsilon$+1.

### CSL Steady-State

Un estado $s$ satisface la formula steady-state $\mathcal{S}_{\bowtie p}[\phi]$ si $\sum_{s' \models \phi} \pi_s(s') \bowtie p$. Es decir si la suma de las probabilidades de estar en un estado $s'$ que satisface $\phi$ es $\bowtie p$ dado que, inicialmente, el sistema se encontraba en el estado $s$. Debido a que PRISM solo modela CTMC's *ergódicos*, estas probabilidades son independientes del estado inicial. Por lo tanto solo basta con calcular la única probabilidad de steady-state del CTMC y ponderar sobre los estados que satisfacen $\phi$. Esto quiere decir que o bien todos los estados satisfacen la formula $\mathcal{S}_{\bowtie p}[\phi]$ o ninguno lo hace. Denotaremos el vector conteniendo las probabilidades steady-state de todo

los estados como $\underline{\pi}$. $\pi$ puede ser computado resolviendo el siguiente sistema de ecuaciones [Ste04]:

$$\underline{\pi} \cdot \mathbf{Q} = \underline{0} \text{ y } \sum_{s' \in S} \pi(s') = 1$$

Este sistema se puede expresar también como $\mathbf{Q}^T \cdot \underline{\pi}^T$ que podemos resolver utilizando los métodos estándar de solución de ecuaciones lineales al igual que con el operador until de PCTL. Nuevamente, PRISM opta por los métodos iterativos.

## 2.8. Métodos Iterativos

Como indicamos en la sección anterior, la verificación de varias operaciones de PCTL y CSL pueden ser reducidas a la solución de un sistema de ecuaciones lineales. Asumiremos que este sistema es de la forma $\mathbf{A} \cdot \underline{x} = \underline{b}$, donde $\mathbf{A}$ es una matriz con valores reales, $\underline{\text{b}}$ es un vector de valores reales y $\underline{\text{x}}$ es el vector que contiene la solución al sistema. Dado que nuestras variables serán los estados, asumiremos que las dimensiones de las matrices y vectores serán de $|S|^2$ y $|S|$ respectivamente.

La solución de sistemas de ecuaciones lineales es un tema bastante estudiado en el cual podemos dividir sus soluciones en dos tipos, los métodos *directos* y los métodos *iterativos*. Los métodos directos poseen la ventaja de calcular el valor real de la solución, modulo errores numéricos introducidos al operar sobre máquinas de precisión finita. Entre los mas famosos encontramos a la eliminación Gaussiana y descomposición L/U. Los métodos iterativos generan aproximaciones a la solución y uno detiene el cómputo cuando considera que se aproximó lo suficiente a la solución. Ésta es la clase de algoritmos que utiliza PRISM y entre ellos encontramos a Jacobi, Gauss-Seidel y el método Power.

Dado que los modelos más simples pueden generan matrices del orden de los millones de filas, la elección de la familia de algoritmos iterativa por parte de PRISM se hace evidente ya que los métodos directos sufren de un fenómeno denominado *fill-in*. Esto significa que la matriz se irá llenando de valores no nulos a medida que vayamos avanzando en su resolución analítica. En consecuencia los

métodos directos requieren que la matriz se actualice numerosas veces. Además, dado que estas matrices en su forma explícita, es decir sin un esquema de compresión, no entran en la memoria de una computadora actual, implementar estructuras de datos que soporten la modificación ocasionada por el fenómeno de fill-in haría que la complejidad de su manipulación incremente demasiado. Finalmente, dado que estas matrices exhiben una regularidad excepcional, PRISM con su motor híbrido logra resolver sistemas con millones de estados, cuyo tamaño de almacenamiento en memoria es despreciable con respecto a la memoria de una computadora comercial.

En lo que resta de esta sección analizaremos el algoritmo iterativo Gauss-Seidel.

### 2.8.1. Método Gauss-Seidel

El esquema general del algoritmos es simple. Se parte de una aproximación inicial $\underline{x}^{(0)}$, y con cada iteración se produce un nuevo vector que se aproxima más a la solución real del sistema. La aproximación computada en la iteración $k$ será denominada $\underline{x}^{(k)}$. Nos referimos a este vector como *vector solución* o *vector iteración*.

Cada estimación $\underline{x}^{(k)}$ es calculada en base a la anterior $\underline{x}^{(k-1)}$. Este proceso se detiene cuando se considera que la solución es lo suficientemente cercana a la solución real del sistema. La forma de determinar esta convergencia es calculando las sucesivas diferencias *relativas* de los vectores, y cuando estas diferencias se hacen menores a un $\epsilon$, se detiene el proceso. Esta diferencia se calcula para cada paso como sigue:

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

En PRISM, por defecto este valor $\epsilon$ es igual a $10^{-6}$.

Tanto el método de Jacobi como el de Gauss-Seidel se basan en la observación

de que el sistema de ecuaciones lineales $\mathbf{A} \cdot \underline{x} = \underline{b}$, planteado como:

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

Puede ser reescrita como:

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

A partir de esta fórmula podemos calcular los sucesivos valores $\underline{x}^{(k)}$ a partir de los anteriores $\underline{x}^{(k-1)}$ de la siguiente manera:

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

Notar que para los modelos que podemos especificar, los elementos de la diagonal van a ser siempre distintos de cero. El algoritmo presentado es el de Jacobi, estos nuevos elementos se calculan en algún orden, normalmente desde la fila cero hasta la última. La observación crucial que diferencia al algoritmo de Gauss-Seidel es que para el cómputo de la $k$-ésima iteración podemos utilizar los valores ya calculados en la misma. Por ejemplo, si el computo se realiza de forma ascendente en las filas, para el calculo de $\underline{x}^k(i)$ podemos utilizar los valores de $\underline{x}^{(k)}(j)$ en lugar de $\underline{x}^{(k-1)}(j)$ siempre que $j < i$. Esto esta expresado en la siguiente ecuación:

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

Esto converge usualmente mucho mas rápido que Jacobi. Además solo necesitamos un vector solución en memoria en todo momento, ya que una vez calculado un nuevo valor en la posición $i$-ésima, éste reemplaza directamente al viejo valor.

$$\underline{x}^{(k)}(i) := (1-\omega)\cdot\underline{x}^{(k-1)}(i) + \omega\cdot\frac{\left(\underline{b}(i) - \sum_{j<i}\mathbf{A}(i,j)\cdot\underline{x}^{(k)}(j) - \sum_{j>i}\mathbf{A}(i,j)\cdot\underline{x}^{(k-1)}(j)\right)}{\mathbf{A}(i,i)}$$

**Figura 2.2:** Gauss-Seidel Succesive Over-Relaxation (SOR)

### Técnica de Sobre-Relajación

Se puede aumentar la convergencia de los algoritmos iterativos mediante la utilización de una técnica llamada *sobre-relajación*. En ésta se calcula el vector de cada iteración como bien mencionamos, pero luego se computa cada nuevo valor como una combinación lineal entre el valor en el vector viejo y el valor en el nuevo vector. Este parámetro de sobre-relajación se llama $\omega$. Presentamos la aplicación sobre Gauss-Seidel en la Figura 2.2.

Este nuevo método se denomina SOR (Succesive Over-Relaxation). La convergencia de este método depende del valor de $\omega$, pero es sabido que para Jacobi y Gauss-Seidel, con $\omega \in (0, 2)$ la convergencia se acelera. Lamentablemente calcular el valor óptimo de $\omega$ es poco práctico, por lo que su elección es en base a de heurísticas.

## 2.9. Motor Híbrido de PRISM

Cuando hablamos de "motor de PRISM"nos estamos refiriendo al núcleo que realiza los cálculos; esto incluye algoritmos, técnicas y estructuras de datos. PRISM posee 4 motores: el **explícito**, en el cual las matrices se representan como arreglo de arreglos ordinarios al igual que los vectores; el motor **ralo** que almacena las matrices de manera mas compacta y almacena los vectores de manera reducida siempre que sea posible; el motor **simbólico**, en el cual la matriz se almacena como un árbol binario de decisión multi-terminal al igual que los vectores; y por último el motor **híbrido** que mezcla los beneficios del almacenamiento ralo y almacenamiento en árbol.

Vamos a analizar estas dos estructuras de datos, las matrices ralas y los árboles de decisión binaria multi-terminales o MTBDD que corresponden a las estruc-

turas utilizadas en nuestra implementación, para luego introducir una estructura híbrida que toma lo mejor de ambos.

### 2.9.1. Matrices Ralas

Como mencionamos anteriormente, existen diferentes técnicas para la compresión de matrices. Una de las mas utilizadas cuando estas contienen muchos ceros es la representación rala. Se basa principalmente en almacenar de manera ordenada todos los valores distintos de cero de la misma.

Dentro de esta técnica hay muchas variaciones, como las que están ordenadas por filas o por columnas, las que solamente guardan los offset de inicio y fin para los valores de columna de cada fila, etc. Nosotros utilizaremos la representación de *lista de coordenadas* o COO por sus siglas en inglés. La razón de porqué escogimos esta representación en lugar de otras mas compactas es que, debido a la implementación de la arquitectura CUDA, estas resultan mas convenientes en cuanto a acceso de memoria se refiere. El espacio extra que se ocupa al utilizar este esquema es lineal en el tamaño de las matrices, con respecto a otros esquemas, y por lo tanto no representa un inconveniente al momento de utilizarlos.

$$\begin{pmatrix} . & 0{,}5 & . & 0{,}5 \\ . & . & 1 & . \\ 0{,}3 & . & . & 0{,}7 \\ 1 & . & . & . \end{pmatrix} =$$

| rows | 0 | 0 | 1 | 2 | 2 | 3 |
|---|---|---|---|---|---|---|
| cols | 1 | 3 | 2 | 0 | 3 | 0 |
| vals | 0.5 | 0.5 | 1 | 0.3 | 0.7 | 1 |

**Figura 2.3:** Representación rala en COO de una matriz

Podemos apreciar en la Figura 2.3 como es que se comprime una matriz de $4 \times 4$ utilizando este esquema. Lo que se hace es especificar todas las coordenadas donde hay un valor distinto de cero, guardando en los arreglos *rows* y *cols* su posición dentro de la matriz y en *vals* su valor. Esta especificación se realiza de manera ordenada desde la primera fila hasta la última y dentro de cada fila de izquierda a derecha. La estructura resultante termina ocupando solamente 96 bytes en lugar de 128 bytes de la versión explicita, si tenemos en cuenta que *rows* y *vals* son arreglos de `int32` y *vals* un arreglo de `double`. Este ejemplo

no es muy representativo pero recordemos que para matrices de modelos grandes esta codificación puede llegar a ocupar un par de Megabytes en lugar de varios Gigabytes.

Podemos calcular que tan grande será la representación en bytes de una matriz en función de la cantidad de valores distintos de cero computando $nnz*16$, donde $nnz$ es la cantidad de valores distintos de cero.

### 2.9.2. Diagramas de Decisión Binaria Multi-Terminales

Los *diagramas de decisión binaria multi-terminales* o MTBDDs por sus siglas en inglés, que son en realidad una extensión de los *diagramas de decisión binaria* o BDDs. Un BDD es un grafo con raíz, acíclico y dirigido que representa una función binaria de la forma $f : \mathbb{B}^n \to \mathbb{B}$.

Los MTBDD extienden los BDD representando funciones que pueden tomar valores arbitrarios de un conjunto $D$, no solo de $\mathbb{B}$, es decir funciones de la forma $f : \mathbb{B}^n \to D$. Para nuestro caso $D$ será $\mathbb{R}$

**Definición:** Sea $\{x_1, \ldots x_n\}$ un conjunto de variables booleanas distintas entre si con un orden total de forma tal que : $x_1 < \ldots < x_n$. Un MTBDD $M$ sobre $\underline{x} = (x_1, .., x_n)$ es un grafo con raíz, acíclico y dirigido. Denominaremos a los vértices del grafo como *nodos*. Cada nodo $m$ de M es un nodo *terminal* o *no terminal*. Un nodo no terminal $m$ representa una variable $var(m) \in \underline{x}$ y tiene exactamente dos hijos, llamados $then(m)$ y $else(m)$. Un nodo $m$ terminal estará rotulado con un valor real $val(m)$ y no tendrá hijos.

Impondremos el orden $<$ de las variables booleanas sobre $M$. Para dos nodos no terminales, $m_1$ y $m_2$, si $var(m_1) < var(m_2)$ definimos $m_1 < m_2$. Si $m_1$ es no terminal y $m_2$ es terminal, definimos $m_1 < m_2$. Adicionalmente para todo nodo no terminal $m$ pediremos que $m < then(m)$ y $m < else(m)$.

Un MTBDD $M$ sobre las variables $\underline{x} = (x_1, .., x_n)$ representa una función $f_M(x_1, .., x_n) : \mathbb{B}^n \to \mathbb{R}$. El valor de $f_M(x_1, .., x_n)$ (o $f_M[\underline{x} = (x_1, .., x_n)]$ para abreviar) se determina recorriendo un camino en M desde la raíz hasta un nodo terminal de la siguiente forma: para cada nodo no terminal $m$, tomar el eje hacia el hijo $then(m)$ si $var(m)$ es $true$ o $else(m)$ si es $false$. Podemos expresar la función que realiza dicha evaluación de manera recursiva definiendo $eval_M : MTBDD \times$

$\mathbb{B}^n \to \mathbb{R}$:

$$eval_M(m, \underline{x}) = \begin{cases} eval_M(then(m), \underline{x}) & \text{si } m \text{ es no terminal} \wedge \underline{x}@var_M(m) \text{ es } true \\ eval_M(else(m), \underline{x}) & \text{si } m \text{ es no terminal} \wedge \underline{x}@var_M(m) \text{ es } false \\ val_M(m) & \text{caso contrario} \end{cases}$$

La razón de por qué los MTBDDs pueden proveer un almacenamiento compacto es porque se almacenan en una forma reducida. Primero, si dos nodos no terminales $m_1$ y $m_2$ son idénticos, es decir $var(m_1) = var(m_2), then(m_1) = then(m_2)$ y $else(m_1) = else(m_2)$, o bien $m_1$ y $m_2$ son nodos terminales y $val(m_1) = val(m_2)$, solo una copia es almacenada. A esto lo denominamos *compartir nodos*. Segundo, si un nodo $m$ satisface que $then(m) = else(m)$, el mismo es removido y todos los ejes provenientes de nodos más arriba de él son dirigidos hacia su único hijo. A esto lo denominamos *saltear niveles*.

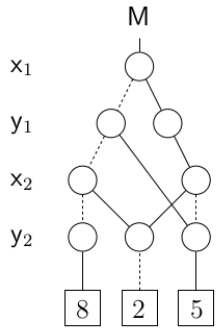
Otro detalle a considerar desde el punto de vista práctico es el orden de las variables. El tamaño de un MTBDD que representa a una función es extremadamente sensible al orden de sus variables booleanas. Esto tiene un efecto directo sobre el espacio requerido para su almacenamiento y también sobre el tiempo necesario para efectuar operaciones sobre el mismo.

El tamaño de un MTBDD también se ve afectado por el número de terminales que contiene, o lo que es lo mismo, el número de valores distintos que la función puede tomar. Un alto número de valores terminales reduce la capacidad de compartir nodos y por lo tanto incrementa el número de nodos en un MTBDD.

### Vectores y Matrices

Una de las más interesantes aplicaciones de los MTBDDs es la representación de vectores y matrices. Para ello consideremos un vector de valores reales $\underline{v}$ de longitud $2^n$. Podemos pensar a $\underline{v}$ como un mapeo de índices a valores reales, i.e. $\underline{v} : \{0, .., 2^{n-1}\} \to \mathbb{R}$. Dado una codificación de los $2^n$ índices en $n$ variables booleanas, i.e. una biyección $enc : \{0, .., 2^{n-1}\} \to \mathbb{B}^n$ como lo es la representación de un entero en binario, podemos representar a $\underline{v}$ como un MTBDD $v$ sobre las variables $\underline{x} = (x_1, .., x_n)$. Podremos decir entonces que $v$ representa a $\underline{v}$ sii $f_v[\underline{x} = enc(i)] = \underline{v}(i)$ para $0 \leq i \leq 2^{n-1}$.

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




| $x_1$ | $y_1$ | $x_2$ | $y_2$ | $f_M$ | Entry in $\mathbf{M}$ |
|---|---|---|---|---|---|
| 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 2.4:** Matriz representada como MTBDD

Podemos extender fácilmente esta idea a matrices. Podemos pensar en una matriz $\mathbf{M}$ de tamaño $2^n \times 2^n$ , como un mapeo de $\{0,..,2^{n-1}\} \times \{0,..,2^{n-1}\}$ sobre $\mathbb{R}$. Nuevamente asumiendo un encoding biyectivo $enc : \{0,..,2^{n-1}\} \to \mathbb{B}^n$, podemos representar a $\mathbf{M}$ por un MTBDD $M$ sobre $2 \cdot n$ variables, $n$ de las cuales corresponden a los índices de fila y $n$ a los índices de columna. Utilizando las variables de fila $\underline{x} = (x_1,..,x_n)$ y de columna $\underline{y} = (y_1,..,y_n)$, podemos decir que $M$ representa a $\mathbf{M}$ sii $f_M[\underline{x} = enc(i), \underline{y} = enc(j)] = \mathbf{M}(i,j)$ para $0 \leq i,j \leq 2^{n-1}$. Podemos apreciar dicha aplicación en la Figura 2.4 Notar que no es necesario que los tamaños de las matrices o vectores sean potencias de dos ya que podemos agregar filas y columnas cero según sea necesario.

# Capítulo 3

# Modelo Computacional CUDA

En este capítulo daremos una introducción a la arquitectura Kepler de las placas NVIDIA con soporte CUDA (*Compute Unified Device Architecture*). Mostraremos los componentes principales que forman parte de la misma, prestando especial atención a los factores de cada componente que guían nuestro desarrollo. Presentaremos la jerarquía de hilos, que agrupa a los hilos en distintos niveles para aumentar la escalabilidad y permitir la cooperación entre ellos de manera eficiente. También presentaremos la jerarquía de las memorias del dispositivo, que utilizadas correctamente, mejora el desempeño de nuestros programas.

## 3.1. Arquitectura del Hardware CUDA

La empresa NVIDIA comenzó en la industria del hardware fabricando GPUs (*Graphic Processing Units*), que son placas aceleradoras gráficas para videojuegos. Estas placas tuvieron mucho éxito en el mercado ya que aceleraban el renderizado de los gráficos en ordenes de magnitud comparado con los renderizados por una CPU, permitiendo la creación de universos virtuales con niveles de detalle que sorprenden aún hoy en día.

En el comienzo estas placas de vídeo contaban con un pipeline grafico donde se ejecutaban en orden distintos programas para completar el renderizado de una imagen. En ese entonces el programador no tenía control sobre estos programas, pero conforme la tecnología de las GPUs fue avanzando se comenzó a permitir la

ejecución de código por parte del programador a través de APIs como DirectX y OpenGL. Algunas personas fueron tomando conciencia de que ahora podían utilizar estos pasos del pipeline para realizar gran cantidad de cálculos matemáticos. Se dieron cuenta de que si formulaban los programas que se ejecutaban en la GPU de manera inteligente, podían hacer que cada píxel representara una función en base a los datos de entrada, como ser texturas y vértices.

Con el tiempo los pasos de este pipeline se fueron pareciendo mas unos a otros. NVIDIA notó esto y decidió unificarlos en una arquitectura que fuera capaz de ejecutar todos estos pasos, además de proveer de un compilador a los desarrolladores que permitiera escribir código en el lenguaje C para ser ejecutado en esta nueva arquitectura. En esta nueva arquitectura, se priorizaban las operaciones matemáticas en paralelo, la velocidad de acceso a memoria y la ejecución concurrente de programas sin bifurcaciones. Estas decisiones de diseño no fueron casualidad, fueron una consecuencia de la estructura de los algoritmos utilizados para renderizar gráficos 3D.

La forma en la que están organizados los componentes en la arquitectura CUDA es lo que permite que las GPUs sean eficientes para realizar tareas intensivas en cálculos. Las arquitecturas CPU están diseñadas para ejecutar concurrentemente un número reducido de hilos, poseen memorias cache muy grandes para reducir los tiempos de acceso y latencia a memoria y una lógica de control muy complicada para la predicción de bifurcaciones y ejecución fuera de orden. En cambio las GPUs están diseñadas para la ejecución concurrente de miles de hilos, donde los programas que ejecutan poseen pocas bifurcaciones y las latencias a memoria global se esconden alternando la ejecución de cientos de hilos.

Al no poseer una lógica complicada para la predicción de bifurcaciones y reducir las latencias a memoria con mas hilos en lugar de usar caches mas grandes, el tamaño de estas unidades de procesamiento es bastante pequeño. Como consecuencia de reducir el tamaño de cada unidad, se logró una mayor concentración de unidades en el chip y un menor consumo energético.

Podemos ver un esquema organizacional de la arquitectura CUDA en la Figura 3.1.

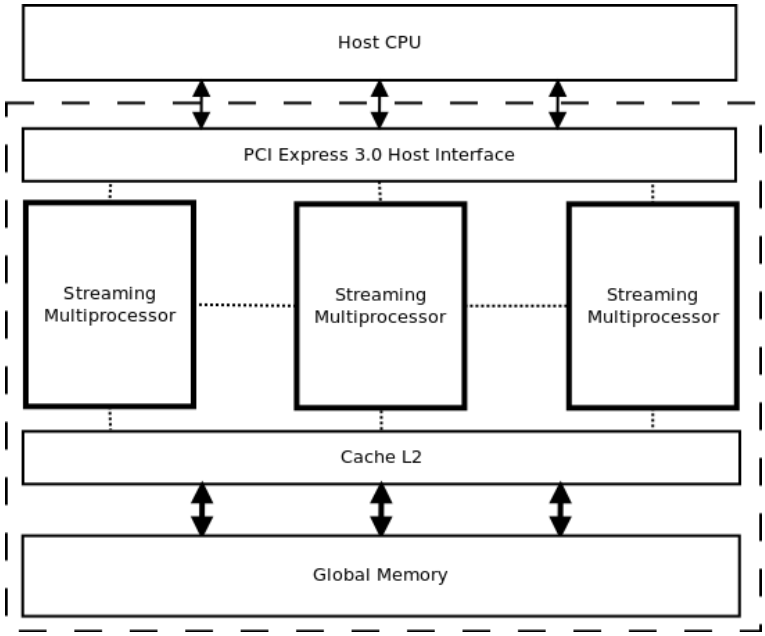



**Figura 3.1:** Arquitectura de un dispositivo CUDA

## 3.2. Arquitectura Kepler

Kepler es la familia mas reciente de los chips CUDA, es por esta razón que nos basaremos en su arquitectura para explicar como funciona la GPU.

Los componentes mas importantes para nosotros en la Figura 3.1, son la **memoria global** y el hardware multithreading conformado por los **Streaming Multiprocessors**. En esta sección presentaremos ambos componentes y señalaremos los factores que nos guiaran en nuestro desarrollo.

### 3.2.1. Memoria Global

Todo dispositivo CUDA cuenta con bancos de memoria GDDR5 ubicados físicamente en el mismo. Esta es una memoria que en la arquitectura Kepler tiene un tamaño de hasta 6 GB y es utilizada como memoria global del dispositivo. Todos los datos que nuestra aplicación necesite deben ser cargados en esta memoria antes de iniciar su ejecución.

La memoria global es accedida por medio de transacciones independientes de 32, 64 o 128 bytes. Estas transacciones deben estar alineadas a su tamaño para

poder ejecutarse eficientemente. Esto significa que si queremos leer un segmento de 64 bytes que comienza en una posición que esta alineada a 32, requeriremos de dos lecturas de tamaño 32 para obtener dicho segmento. Afortunadamente el programador no tiene que crear estas transacciones de memoria explícitamente.

Los hilos que se ejecutan en un dispositivo CUDA se dividen en grupos de 32 hilos llamados **warps**. Dentro de un warp todos los hilos ejecutan la misma instrucción simultáneamente. Cuando un warp ejecuta una instrucción que requiere acceso a memoria global estos accesos son agrupados en una o mas transacciones de memoria automáticamente, dependiendo del tamaño de los datos a leer, su posición y de si la misma se encuentra alineada correctamente. Los hilos de un warp deben esperar hasta la finalización de todas las transacciones , que fueron creadas como consecuencia de su acceso a memoria, antes de continuar su ejecución.

A mayor número de transacciones, mayor será el tiempo que un warp permanecerá suspendido. Por lo tanto para maximizar el throughput, es importante aumentar la coordinación de los accesos a memoria, utilizando tipos de datos que cumplan con el tamaño y la alineación requerida. Esto puede ir tan lejos como agregar padding si es necesario entre los datos, para así forzar que los mismos estén alineados. Volveremos a revisar este concepto en la subsección 3.4.2 donde consideraremos los factores que afectan a la performance de nuestros códigos CUDA.

Los accesos a memoria global requieren de aproximadamente 500 ciclos de reloj para completarse. Afortunadamente estas latencias pueden ser ocultadas casi por completo.

### 3.2.2. Streaming Multiprocessors

La arquitectura CUDA de NVIDIA esta compuesta de un arreglo escalable de *Streaming Multiprocessors* (SMs de ahora en adelante). Cuando el programador invoca una función o *kernel* CUDA, el dispositivo se encarga de distribuir la ejecución a lo largo de los SMs.

Una invocación a kernel, se hace a través de lo que se denomina una **grilla de ejecución**. Podemos imaginar a una grilla de ejecución como un cubo tridi-

mensional, donde cada punto del cubo en el espacio representa un **bloque**. Estos bloques nuevamente son representables como un cubo en un espacio tridimensional donde cada punto del cubo representa un **hilo**. Es tarea del programador definir las dimensiones de la grilla en tiempo de ejecución.

Los bloques son el mecanismo que tiene CUDA para lograr que los kernels escalen con la cantidad de SMs. Idealmente, cada bloque debe resolver una conjunto de datos de manera independiente y la cantidad de trabajo debe ser constante, o lo que es lo mismo, un programa CUDA debe presentar paralelismo de datos y no de tareas. De esta manera conforme las siguientes arquitecturas CUDA agreguen mas núcleos, nuestro código escalará en ellas de manera eficiente.

Como ya mencionamos, dentro de cada bloque se agrupan los hilos por **warps**. Esta jerarquía (grilla, bloque, warp, hilo) nos permite identificar unívocamente cada thread en base a su posición dentro de estos cubos virtuales y de esta forma obtener los datos a computar. Es tarea del programador establecer un mapeo entre las posiciones virtuales de los hilos y los datos a computar.

Todos los hilos de un bloque se asignan a un único multiprocesador y múltiples bloques pueden ser asignados a un mismo multiprocesador. A medida que los bloques vayan finalizando su ejecución, nuevos bloques que no están asignados a ningún multiprocesador los reemplazarán.

Un multiprocesador puede tener a cargo la ejecución de miles de hilos, pero está diseñado para ejecutar concurrentemente solo un número reducido. Para poder manejar tal cantidad de hilos, se utiliza una arquitectura de instrucciones llamada *SIMT* (Single-Instruction, Multiple-Thread). En esta arquitectura, 32 unidades de procesamiento llamadas *Stream Processors* (SPs de ahora en adelante) ejecutarán la misma instrucción en todas los hilos de un warp de manera simultánea. Un dispositivo de arquitectura Kepler cuenta con hasta 15 multiprocesadores y cada uno con 192 stream processors, por lo que un único dispositivo puede ejecutar concurrentemente 2496 hilos. Recordemos que actualmente los procesadores CPU solo pueden ejecutar un máximo de 8 hilos simultáneamente.

El contexto de ejecución de cada warp (registros, program counters, etc.) es almacenado en la memoria interna de cada SM durante toda la ejecución del warp. Esto implica que cambiar de un contexto de ejecución a otro no tiene costo, y en cada instante de tiempo donde haya que despachar una instrucción, un scheduler

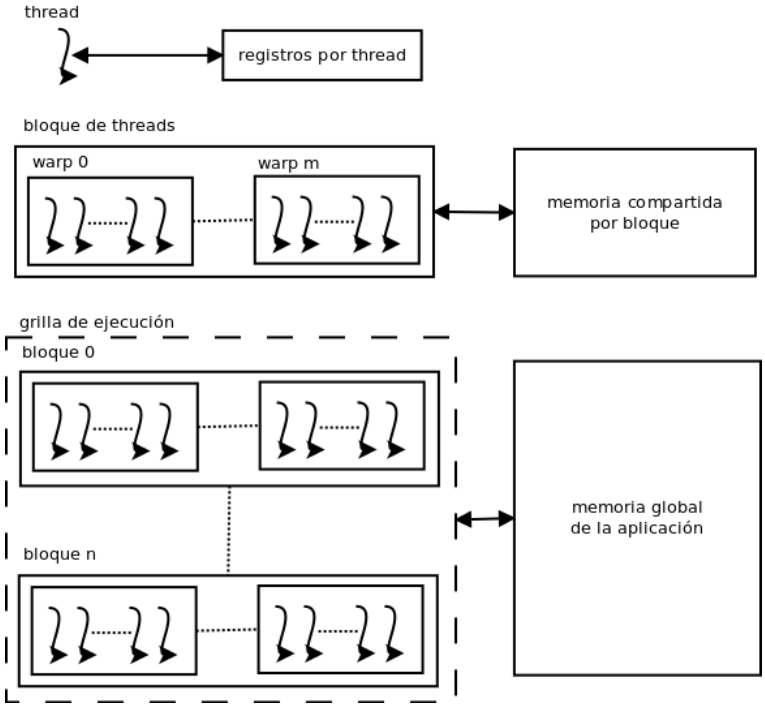



**Figura 3.2:** Jerarquía de memoria e hilos

de warps que tiene warps listos para ejecutar su siguiente instrucción pondrá en ejecución alguno de estos.

### 3.2.3. Memoria de un Streaming Multiprocesor

En la Figura 3.2 podemos observar la jerarquía de memoria y de hilos. En esta figura podemos ver que cada multiprocesador tiene una *memoria de registros*, que se particiona entre los hilos/warps, y una cache de datos o *memoria compartida*, que se reparte entre los bloques. Este es un hecho importante de la arquitectura para nosotros, ya que debemos asegurarnos al momento de programar nuestros kernels que el uso de registros y memoria compartida por parte de los bloques permita la completa ocupación de los SPs en cada SM.

El número máximo de hilos que pueden residir en un SM en un momento dado en la arquitectura Kepler es 2048, el número máximo de warps 64 y el número máximo de bloques 16. Esto significa que si bien podemos modificar libremente la cantidad de hilos en un bloque, estaremos limitados en cuanto a la utilización del

SM si no elegimos valores adecuados. Por ejemplo, podemos crear bloques de 64 hilos, pero esto nos limitaría a poder correr 1024 (64 hilos × 16 bloques máximo por SM) hilos en un SM dado. Si bien una utilización de los SMs al 100 % no implica un aumento en el rendimiento, es recomendable para *ocultar la latencia* a los accesos a memoria global, ya que cuando un warp se bloquee esperando una lectura o escritura en memoria, otro warp podrá tomar su lugar inmediatamente.

Dicho esto, cuando tenemos una utilización menor al 100 % la estrategia para evitar las latencias a los accesos de memoria global es utilizar la memoria on-chip del SM de manera intensiva.

A continuación daremos una breve explicación de las memorias de un SM.

### Memoria Compartida

Al estar on-chip esta memoria tiene un mayor ancho de banda y una menor latencia que la memoria global. Está compuesta por 32 bancos de memoria que pueden ser accedidos simultáneamente. Sin embargo, si múltiples direcciones de memoria apuntan al mismo banco estos accesos serán serializados. La única excepción es cuando múltiples hilos de un warp acceden a la misma dirección de memoria, ya que solo se necesitará una lectura para estos hilos. Su tiempo de acceso es de 6 ciclos y su tamaño de 64KB, que se comparten con la **Cache L1**. Esto permite tener configuraciones de memoria compartida/cache de 48 KB/16 KB, 32 KB/32 KB o 16 KB/48 KB.

### Cache de Memoria Constante

Esta es otra cache on-chip pero de solo lectura. Su tamaño es de 8 KB por multiprocesador y es utilizada para almacenar arreglos o estructuras de datos constantes.

### Cache de Memoria de Texturas

La cache de memoria de texturas es similar a la memoria constante, pero se diferencia en que esta optimizada para los accesos a arreglos bidimensionales.

## 3.3. Anatomía de un Programa en CUDA

La estructura de un programa CUDA consiste en una o mas fases que son ejecutadas en el CPU host o en la GPU device. Las fases que no posean mucho paralelismo serán implementadas en el código host y las fases que exhiban paralelismo serán implementadas en el código CUDA. Un programa CUDA C contiene código a ejecutar en el host (C estandar) como también en el dispositivo (un subconjunto de C al cual se agregan decoradores para la declaración de atributos a las variables y funciones). El compilador CUDA C de NVIDIA (*nvcc*) es el que separa ambos fragmentos al momento de compilar. El código host al ser simplemente código C estándar es procesado por el compilador C de la máquina y corre como cualquier otro proceso del CPU. El código CUDA, por otro lado, es traducido a *PTX* que es el equivalente al assembler de las arquitectura x86 para dispositivos CUDA y finalmente compilado para el dispositivo objeto.

La ejecución de un programa típico CUDA está esquematizada en la Figura 3.3. Su ejecución comienza con el código host, que luego invoca a las llamadas a kernel para explotar el paralelismo del problema a resolver. El código serial puede invocar la ejecución de más de un kernel en un dispositivo o también para varios dispositivos CUDA en la misma máquina. Cada kernel se ejecutará con una grilla de ejecución cuyas dimensiones son determinadas por el programa serial.

El programa serial es el encargado de cargar en memoria global de los dispositivos a utilizar todos los datos que serán requeridos para realizar los cómputos. Como podemos invocar varios kernels simultáneamente, superponer transferencias de memoria desde la memoria de host hacia la memoria del dispositivo con la computación de otros datos independientes suele ser ventajoso para ocultar la latencia de comunicación host-device.

### 3.3.1. Ocupación

Como ya mencionamos, la ocupación es uno de los factores que nos guiarán en el desarrollo de programas en esta arquitectura y se refiere al porcentaje ocupación de los SMs. Conocer las razones que nos limitan es fundamental para desarrollar un códigos que utilice el dispositivo de manera eficiente. Definiendo $n_{sched}$ como el número de hilos asignadas a un SM en un momento dado y $N_{sched}$ como la

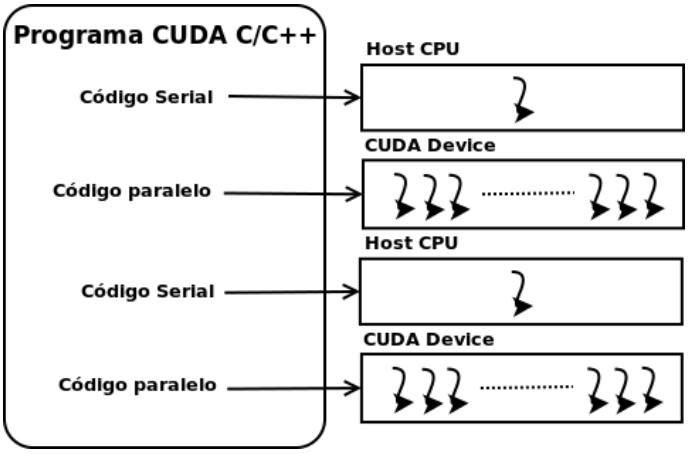



**Figura 3.3:** Estructura de un programa CUDA

cantidad máxima de hilos que cada SM puede manejar en un momento dado, podemos calcular la ocupación de la siguiente manera:

$$Occupancy = \frac{n_{sched}}{N_{sched}} \tag{3.1}$$

Si bien tener una ocupación del 100 % no implica que nuestro código se ejecute mas rápido, suele ser siempre el caso para los kernels que están limitados por el ancho de banda a memoria, ya que se logra ocultar la latencia completamente.

### 3.3.2. Throughput

Otro de los factores que guiarán nuestro desarrollo es el throughput y se mide como el trabajo efectivo realizado por un recurso en una unidad de tiempo. Cualquier trabajo que podamos medir podrá ser objeto de un análisis de performance. Normalmente cuando hablamos del throughput nos referimos a la cantidad de instrucciones o de operaciones de memoria realizadas por el dispositivo en una unidad de tiempo. Este factor está altamente relacionado a la ocupación como analizaremos en las siguientes secciones.

Idealmente queremos mantener la ocupación al 100 % y aumentar el throughput a los límites del hardware. Esto se logra con una buena estrategia para partir el problema y sucesivos análisis de los datos obtenidos por las herramientas uti-

lizadas para realizar profiling.

### 3.3.3. Paralelismo de Datos

El paralelismo de datos es la propiedad que tiene un programa para realizar operaciones aritméticas de manera segura sobre las estructuras de datos de manera simultánea. Como ejemplo clásico podemos mencionar a la multiplicación de matrices, ya que en ella cada elemento de la matriz resultante **P** se computa realizando el producto punto entre las filas de una matriz **N** y las columnas de una matriz **M**. En este caso podemos realizar los productos punto de manera simultánea para cada elemento de la matriz **P**.

Programas que no exhiban paralelismo de datos no son candidatos a ser implementados en CUDA, ya que no se lograrán speedups sustanciales o bien correran mas lentos.

## 3.4. Consideraciones de Performance

Aunque un kernel pueda ejecutarse correctamente en un dispositivo CUDA, su performance variará en función de las restricciones de los recursos del dispositivo. Podemos aumentar la performance de manera drástica sabiendo cuales son los factores limitantes e intercambiando el uso de un recurso por otro. Pero sin un conocimiento profundo de la arquitectura y de sus restricciones solo podemos intentar adivinar estrategias para optimizar nuestro kernel. Es por esta razón que revisaremos nuevamente los factores que afectan la performance de nuestros kernels.

Mencionaremos primero los factores que afectan a la performance de la ejecución de los hilos.

### 3.4.1. Ejecución de Hilos

Conceptualmente no podemos asumir nada sobre el orden de ejecución de los hilos en un bloque, así como de los bloques en la grilla de ejecución, pero podemos utilizar barreras de sincronización dentro de nuestros bloques para asegurarnos

de que todos los hilos completaron una fase de la ejecución antes de comenzar la siguiente fase. Tales estrategias de sincronización afectan de manera negativa el performance, por lo que debemos intentar evitarlas siempre que sea posible.

Como ya vimos, los hilos de cada bloque se agrupan en warps. Esta técnica de implementación ayuda a reducir el costo de hardware y permite algunas optimizaciones para servir los accesos a memoria. Los bloques se particionan en warps basándose en el índice de los hilos. Si un bloque tiene una sola dimensión, la partición es directa. Si un bloque tiene dos o mas dimensiones, las dimensiones se proyectaran en un orden lineal para particionarlas en warps. La proyección se hace primero en el eje $x$, luego en el $y$, y finalmente en el eje $z$.

Una vez realizada la partición el hardware puede ejecutar una instrucción para todos los hilos de un mismo warp, antes de moverse a la siguiente instrucción. Este estilo de ejecución es lo que se llama SIMT. Esto funciona a la perfección cuando todos los hilos de un warp recorren el mismo camino en el código. Por ejemplo, en una instrucción **if**, si todos los hilos del warp toman el mismo camino no habrá inconvenientes en la ejecución. Pero cuando alguno de esos hilos toma el camino **else**, la ejecución SIMT presenta problemas. Cuando esto sucede la ejecución del warp va a requerir de dos o mas pasadas por el mismo código, una por cada *bifurcación* que suceda en el mismo. Estas pasadas son secuenciales, por lo que aumentan el tiempo de ejecución. Cuando esto sucede diremos que los hilos de un warp *divergen*.

Las divergencias también pueden surgir en los **while** loops. Por ejemplo, si el warp ejecuta un loop en el cual algunos hilos ingresan 6, otros 7 u 8, todos terminarán la ejecución de las primeras 6 pasadas en conjunto, pero luego algunos continuarán hasta terminar las restantes.

### 3.4.2. Ancho de Banda a Memoria Global

Uno de los factores mas importantes en cuanto a la performance de los kernels es el máximo aprovechamiento del ancho de banda a memoria global. Los programas CUDA explotan el paralelismo masivo de datos, esto quiere decir que necesitan acceder a una gran cantidad de datos y deben hacerlo en un periodo de tiempo pequeño. Para lograr esto podemos generar distintas estrategias para

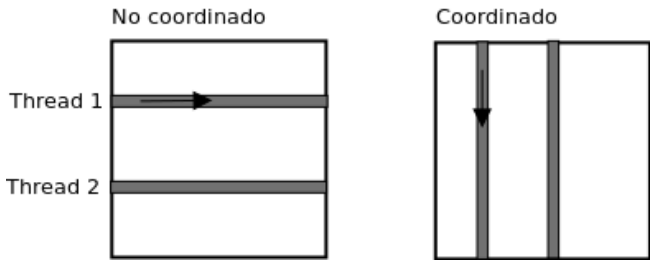



**Figura 3.4:** Accesos de memoria coordinados y no coordinados

utilizar la memoria compartida como una cache manejada por el programador a memoria global y de esa forma aumentar el throughput. La estrategia mas común es dividir los datos necesarios para computar nuestro resultado en *tiles* que entren en memoria compartida de un bloque. De esta manera, los valores de dicha tile serán cargados una sola vez desde memoria global, y luego podremos acceder a estos datos con accesos no coordinados sin tener que pagar la penalización de acceder a memoria global.

Para obtener un alto throughput en los accesos a memoria global muchas veces suele ser favorable reorganizar los datos o cambiar nuestro algoritmo. Como ejemplo tenemos a la Figura 3.4 donde podemos ver que el acceso secuencial por parte de un thread a los valores de una fila de la matriz no genera accesos coordinados. Esto sucede porque la matriz al estar representada en una memoria lineal, ubica una fila después de otra, por lo que si dos hilos de un mismo warp piden el elemento $i$-ésimo de dos filas distintas generarán dos transacciones de memoria distintas. En cambio, esto no sucede cuando ambos hilos acceden a columnas contiguas de una misma fila ya que se crea una sola transacción.

Como mencionamos anteriormente también es importante que los accesos a memoria estén alineados. Como ejemplo podemos ver la Figura 3.5, en donde vemos que por más que los accesos sean coordinados, se generan mas transacciones que las que generaría una lectura a una posición de memoria correctamente alineada. En este caso se utilizarán dos transacciones en lugar de una, que es lo necesario para leer 32 `int32` (4 bytes por `int` $\times$ 32 hilos por warp). Estrictamente hablando se realizará una transacción de 32 bits para los primeros dos valores y una de 128 bits para los siguientes 30 valores.

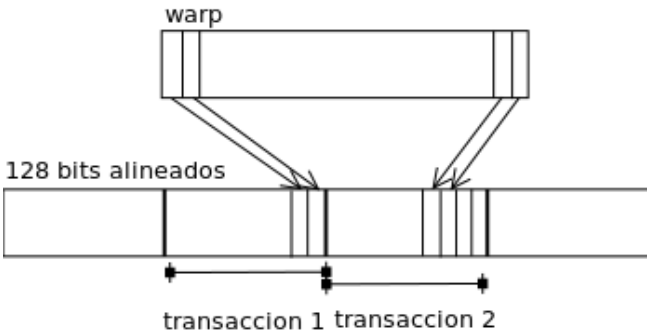



**Figura 3.5:** Accesos de memoria coordinados pero no alineado

### 3.4.3. Restricciones de Recursos

La principal limitación de recursos es el tamaño de la memoria global. Si no hay suficiente memoria global en el dispositivo para almacenar todas las estructuras de datos, tendremos que particionar el problema de una manera diferente. Como último recurso también contamos con el mapeo de memoria de dispositivo a memoria host. Dado que el dispositivo se encuentra conectado al host por medio del puerto PCI Express, la latencia de los accesos a memoria host es muy grande con respecto a la velocidad de la memoria en la placa.

Otros componentes que cuentan con recursos limitados son los SMs. Si no utilizamos correctamente sus recursos podemos tener como resultado una ocupación menor al 100 %. Recordemos que cada SM cuenta con una memoria compartida (48/32/16 KB), una memoria de almacenar los registros (64KB), un número máximo de warps y un número máximo de bloques que puede coordinar.

La memoria compartida y de registros debe distribuirse entre todos los bloques asignados al SM en un instante dado. Esto significa que si los distintos bloques de un kernel asignados a un SM utilizarán en conjunto mas registros que los que entrarían en 64 KB, los registros restantes (*spilled registers*) serán trasladados a memoria compartida. Esta estrategia se define al momento de compilación y es transparente para el programador.

Podemos calcular la cantidad de hilos que podrán ser coordinadas por un SM conociendo las dimensiones de nuestros bloques y su utilización de memoria compartida. Esto nos permitirá conocer la ocupación teórica del dispositivo. Para calcular la ocupación efectiva debemos utilizar el profiler.

A continuación presentaremos distintas fórmulas que nos ayudarán a calcular la ocupación teórica.

- **Límite por utilización de memoria**: dado que la memoria compartida se distribuye entre todos los bloques de un SM, la cantidad de bloques que puede manejar sera igual al cociente entre el tamaño de la memoria compartida y la cantidad que requiere cada bloque para operar correctamente.

$$bloques_1^{max} = \left\lfloor \frac{\text{Memoria compartida por SM}}{\text{Memoria utilizada por bloque}} \right\rfloor \tag{3.2}$$

- **Límite de hilos**: cada SM posee un número reducido de *warp schedulers*, estos son los encargados del bookkeeping de los warps y de invocar la ejecución de instrucciones en los SP. Es por esta razón que el número de warps que pueden ser coordinados por un SM tiene un límite. En la arquitectura Kepler ese límite es de 64 warps por SM.

$$bloques_2^{max} = \left\lfloor \frac{\text{Warps scheduleables}}{\text{Warps por bloque}} \right\rfloor \tag{3.3}$$

- **Límite de bloques**: adicionalmente a los warps, la cantidad de bloques que pueden ser asignados a un SM en un momento dado también tiene un límite. Este valor es una constante que varía con cada familia de SMs CUDA que llamaremos $bloques_3^{max}$. En la arquitectura Kepler este límite es 16 por SM.

Podríamos entonces calcular la ocupación teórica de nuestros SMs haciendo uso de la ecuación 3.4.

$$Occupancy = \frac{\text{máximo número de hilos por SM}}{(min(bloques_1^{max}, bloques_2^{max}, bloques_3^{max})) \cdot \text{hilos por bloque}} \tag{3.4}$$

# Capítulo 4

# Model Checking en CUDA

En este capítulo explicaremos nuestra solución para acelerar el model checker probabilístico PRISM implementando una variación del algoritmo Gauss-Seidel para la plataforma CUDA. De esta manera logramos reducir el tiempo necesario para verificar propiedades cuantitativas de manera significativa. Comenzaremos analizando como es que PRISM utiliza los MTBDDs y las matrices ralas para representar matrices de transición, ambos introducidos en las subsecciones 2.9.2 y 2.9.1. Luego presentaremos la estructura que combina a ambas y que es utilizada por el motor híbrido para representar las matrices de transición de manera compacta y de eficiente acceso.

Seguido a esto, explicaremos nuestra estrategia para implementar Pseudo Gauss-Seidel de forma eficiente en CUDA. Debido al gran tamaño que las estructuras de datos pueden requerir para distintos modelos analizaremos dos soluciones, una *directa* y otra *splitted* o partida. Finalizaremos este capítulo dando una breve descripción de la herramienta *libcudasles* (CUDA Sparse Linear Equation Solver Library) resultante como subproducto de esta tesis.

## 4.1. Representación de las Matrices de Transición

Como ya vimos en la Sección 2.7, la implementación de los operadores que calculan las probabilidades que tienen los estados de un modelo de satisfacer una

propiedad se reducen a realizar operaciones de multiplicación matriz-vector. Estas operaciones se realizan sobre las matrices de transición de los modelos DTMC y CTMC. Las matrices de transición pueden llegar a tener millones de filas, por lo que es necesario explorar distintas formas para almacenarlas en memoria y poder operar con ellas de manera eficiente. Vamos a revisar las dos formas de representación anteriormente mencionadas y presentaremos también la estructura *Offset-labelled* MTBDD que utiliza por el motor híbrido de PRISM.

### 4.1.1. Biyección entre Estados y Filas

Dado que las matrices en la computadora solo pueden ser indexadas en números naturales y no en estados, existen distintas biyecciones entre las filas de las matrices de transición y los distintos estados del sistema. Podríamos enumerar todos los estados alcanzables del sistema y de esa manera obtener una biyección. Este método es denominado *método de enumeración*. Notemos también que a partir del valor de las variables del sistema, es posible saber en que estado se encuentra el mismo. El método que toma el conjunto de variables del sistema con su estado y retorna un numero de fila se denomina *método estructurado*.

Para ilustrar como funcionaría el método estructurado supongamos que nuestro sistema cuenta con solo dos variables $x$ e $y$, donde $x \in [0, 3]$ e $y \in [0, 4]$. Podemos representar ambas variables con 2 y 3 variables booleanas respectivamente, que corresponderían a la representación binaria del valor de $x$ e $y$. Si nuestro sistema se encuentra en el estado $x = 2$,$y = 4$, nuestra codificación binaria sería $x_1 = 1, x_0 = 0, y_2 = 1, y_1 = 0, y_0 = 0$. Concatenando estas variables podemos formar el vector $\underline{z} = (1, 0, 1, 0, 0)$. Si ahora interpretamos ese vector como la representación en base 2 de un número obtendríamos el número 20. Por lo que las transiciones del estado $x = 2$, $y = 4$ se encontraría en la fila numero 20 de la matriz de transiciones.

Utilizando este método, la cantidad de filas crece exponencialmente con el número de variables que tengamos en el sistema para el método estructurado. Ya que por cada variable que agreguemos debemos multiplicar el numero de filas por $2^m$ donde $m$ es la primera potencia de dos que es mayor al valor máximo de la variable a insertar. Esto significa que por mas que nuestro ejemplo anterior posea

4 estados alcanzables, utilizaremos una matriz de transiciones de $2^5 \times 2^5$ para representar sus transiciones.

A pesar de esto, el método utilizado por PRISM para representar las matrices en el motor híbrido es el estructurado, ya que logra reducir la cantidad de nodos del MTBDDs a menos de un 1 % respecto al método de enumeración.

### 4.1.2. Transiciones como Matrices Ralas

Esta es la manera mas simple de almacenar las transiciones. Dado que un estado de un modelo con millones de estados puede llegar a tener no mas de un par de cientos de transiciones, almacenar solo los valores distintos de cero resulta muy ventajoso. Sin embargo, a diferencia del almacenamiento explicito, si queremos acceder al valor de una celda en particular tendremos que realizar dos búsquedas binarias sobre los arreglos *rows* y *cols* respectivamente. Pero si queremos leer las filas y columnas en orden este es el método mas conveniente.

Otro inconveniente de utilizar esta estructura es que los mismos valores reales son almacenados muchas veces. Es decir, todo valor se repite cuantas veces aparezca en la matriz explícita. Esto no sucede con los MTBDDs donde solo se crea un nodo por cada valor distinto en la imagen y luego su referencia se repite. No obstante, el motor ralo de PRISM es el más rápido para los modelos en los que es posible su verificación.

### 4.1.3. Transiciones como MTBDDs

Como mencionamos anteriormente en la sección 2.9.2, el orden de las variables afecta enormemente el tamaño resultante del MTBDD. En [Par02] se demostró empíricamente que el tamaño del MTBDD se reduce cuando alternamos la variables booleanas de las filas y las columnas. De esta forma el MTBDD logra reducir su tamaño al aprovechar la regularidad entre los distintos estados. En esta codificación los nodos de niveles pares dividirán la matriz original en dos a lo largo de las filas y los nodos de niveles impares la dividirán en dos a lo largo de las columnas.

Nuevamente si queremos acceder al valor de una celda debemos recorrer un *camino* partiendo desde la raíz hasta llegar a un nodo terminal, tomando en cada

paso la arista que nos lleve a la sub-matriz que contiene a la celda. Este orden alternado hace que la implementación clásica del algoritmo Gauss-Seidel sea poco práctico ya que nos obliga a visitar numerosas veces nodos que corresponden a las variables de fila. Por ejemplo, si las variables correspondientes a los índices de las filas estuvieran antes que las variables correspondientes a los índices de las columnas solo tendríamos que recorrer un sub-árbol del MTBDD de solo $n$ niveles en lugar de $2 \cdot n$ niveles. Pero de esta manera el MTBDD no puede beneficiarse de la regularidad de la matriz de transiciones.

Sin embargo podemos escribir un algoritmo Pseudo Gauss-Seidel en donde en lugar de calcular una fila después de la otra calculamos un conjunto de filas contiguas al mismo tiempo. Esto hace que al recorrer el MTBDD aprovechemos mas las visitas a cada nodo. Esta operación puede afectar negativamente el número de iteraciones que debemos realizar con respecto al Gauss-Seidel original, y requiere además que guardemos en un vector los valores intermedios para cada fila. Esto se debe a que dentro de cada conjunto de filas que calculamos conjuntamente, todas ellas necesitan utilizar el valor de la iteración anterior al multiplicarse con valores del vector solución que se encuentren en los mismos índices de columna que las filas que estamos calculando.

Combinando la ventaja de explotar la regularidad de la matriz de transiciones y el rápido acceso a los valores de las matrices ralas es que podemos crear una estructura híbrida para almacenar matrices grandes y aumentar la eficiencia con respecto al recorrido del MTBDDs.

### 4.1.4. Transiciones como una Estructura Híbrida

Con la idea de combinar matrices ralas y MTBDDs es como se crea la estructura híbrida mencionada a continuación, propuesta en [Par02]. Recordemos que con la partición escogida, cada par de nodos adyacentes (nivel par, nivel impar) describe una submatriz cuadrada de la matriz original de dimensiones $2^n \times 2^n$.

La idea clave es que si bien los índices de fila y columna de las entradas en cada submatriz del MTBDD son diferentes cada vez que ocurren en la matriz original, los índices locales relativos a la submatriz son los mismos. Por lo que la optimización propuesta almacena las submatrices de cada nodo como matrices

ralas, que almacenan solo los índices relativos dentro de la submatriz en el árbol. Esto reduce el tiempo de acceso a cada valor al no tener que recorrer un árbol para obtener los valores debajo de cada nodo, solo un barrido lineal en los arreglos de la matriz rala sera suficiente.

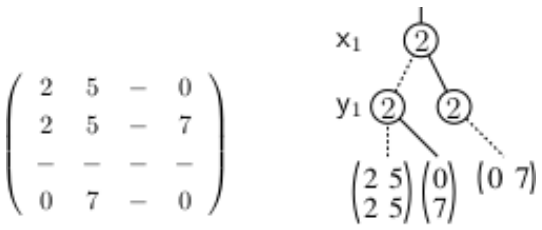
La manera en que se implementa es la siguiente: para cada nivel del MTBDD se calcula el consumo de memoria que impondría crear matrices explicitas para cada nodo y se selecciona el nivel mas conveniente. Esto evita la duplicación de matrices ya que si las insertamos en distintos niveles es posible que dos nodos a distintas alturas del árbol que no tienen relación compartan un mismo nodo hijo. Al insertar matrices ralas para esos dos nodos tendríamos redundancia de los datos de este nodo hijo.

Dado que la cantidad de estados alcanzables es menor al numero de filas de las matrices, el motor híbrido utiliza arreglos para almacenar los vectores $\underline{x}$ y $\underline{b}$ que tienen una longitud igual al número de estados alcanzables. En estos vectores los índices no cumplen con la biyección entre índices y estados como las filas de la matriz. Por lo que para poder utilizar el MTBDD en el motor híbrido, la nueva estructura deberá eliminar virtualmente las filas y columnas nulas o que corresponden a estados no alcanzables.

Antes de aplicar esta última optimización, si queríamos obtener los índices de fila y columna de los valores almacenados en el MTBDD para cada paso desde $m$ hacia $then(m)$ teníamos que sumar una potencia de dos a las filas o columnas. Al eliminar estados no alcanzables esto ya no se cumple. Por lo que para obtener los índices de las entradas en la nueva matriz contraída en cada paso se deberá sumar un offset que estará almacenado en el nodo correspondiente y que sera menor que la potencia de dos almacenada anteriormente. Este offset indicará el número de filas o columnas no nulas y alcanzables que posee la submatriz almacenada por la arista $else(m)$. Esta estructura se denomina *Offset-labelled MTBDD*. En la Figura 4.1 podemos apreciar un ejemplo.

## 4.2. Algoritmo Pseudo Gauss-Seidel

Ahora presentaremos nuestra implementación de un algoritmo pseudo Gauss-Seidel que utilizará la estructura mencionada en la sección anterior. Este algo-



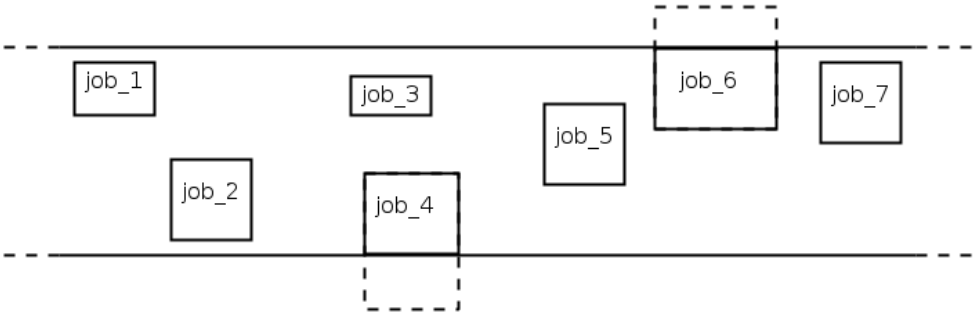

| Path | | Offsets | | Local | Global |
|---|---|---|---|---|---|
| $x_1$ | $y_1$ | $x_1$ | $y_1$ | entry | entry |
| 0 | 0 | - | - | $(0,0) = 2$ | $(0,0) = 2$ |
| | | | | $(0,1) = 5$ | $(0,1) = 5$ |
| | | | | $(1,0) = 2$ | $(1,0) = 2$ |
| | | | | $(1,1) = 5$ | $(1,1) = 5$ |
| 0 | 1 | - | 2 | $(1,0) = 7$ | $(1,2) = 7$ |
| 1 | 0 | 2 | - | $(0,1) = 7$ | $(2,1) = 7$ |

**Figura 4.1:** Ilustración de la estructura Offset-labelled MTBDD

ritmo tiene la denominación pseudo debido a que no es estrictamente igual al algoritmo original.

En el algoritmo original, presentado en la Subsección 2.8.1, los nuevos valores del vector $\underline{x}$ se calculan de manera ordenada desde la primer fila hasta la última. En nuestro caso adoptaremos otra estrategia para calcular los valores de $\underline{x}$ en la cual calcularemos simultáneamente conjuntos de filas contiguas que denominaremos *slices*. Cada bloque en nuestra grilla de ejecución calculara solamente un slice del vector solución. Seguimos explotando el paralelismo de datos de la multiplicación de matrices ya que cada bloque podrá calcular su resultado de manera independiente a los otros bloques.

Cada bloque necesitará saber cuales son las matrices ralas que contienen valores del slice que calcula, para así poder multiplicarlas con el vector solución. Además, debe de saber cuales son los valores dentro de cada matriz rala que corresponden a las filas de su slice, ya que una matriz rala puede contener valores de varias filas y slices. Toda esta información será almacenada en lo que denominaremos una estructura *job* o trabajo. Adicionalmente esta estructura almacenará la posición absoluta dentro de la matriz original. A continuación presentamos una lista de los items que almacena junto con una breve descripción de los mismos:




| Asignación de trabajos | | |
|---|---|---|
| $warp_0$ | $job_1$ | $job_5$ |
| $warp_1$ | $job_2$ | $job_6$ |
| $warp_2$ | $job_3$ | $job_7$ |
| $warp_3$ | $job_4$ | |

**Figura 4.2:** Asignación de trabajos a warps de un bloque

- **sm_ptr** : un puntero que referencia a la matriz descrita por el trabajo.
- **offset**$_{row}$ : offset de fila de esta instancia de la matriz dentro de la matriz original.
- **offset**$_{col}$ : offset de columna de esta instancia de la matriz dentro de la matriz original.
- **begin** : índice de los arreglos $row$, $cols$ y $vals$, desde donde comienzan los valores pertenecientes al slice del job.
- **end** : índice de los arreglos $row$, $cols$ y $vals$, que indica donde terminan los valores pertenecientes al slice del job.

Mencionamos que cada bloque calculará un slice, y por lo tanto cada bloque procesará varios jobs. Estos jobs serán distribuidos a los distintos warps de un bloque de manera uniforme. Es decir, cada warp procesará exclusivamente un job al mismo tiempo y un job no podrá ser procesado por mas de un warp. La manera en que se distribuyen esta ilustrada en la Figura 4.2.

A continuación presentaremos el algoritmo ejecutado por cada warp que multiplica las matrices ralas utilizando la información de los jobs.

### 4.2.1. Multiplicación de las Matrices Ralas

En la Figura 4.3 mostramos una implementación escrita en pseudo-código del algoritmo de multiplicación de matrices ralas para CUDA. Cada llamada a la función **sm_cuda_mult** será invocada por los distintos warp y solo los parámetros *sm_ptr*, *begin* y *end* variarán en cada invocación para las warps de un bloque.

Como mencionamos anteriormente, cada bloque calcula un slice del vector solución, por lo que durante la computación de ese slice es de nuestra conveniencia mantener los valores intermedios ya calculados en la memoria compartida, ya que de esta manera evitamos accesos a memoria global cada vez que guardemos los valores de las operaciones punto a punto.

Nuestra función **sm_cuda_mult** posee varios parámetros de entrada que listaremos a continuación: **sm** la matriz rala a multiplicar; **x** el vector solución; $\mathbf{offset}_{row}$ y $\mathbf{offset}_{col}$ que corresponden a la posición de **sm** dentro de la matriz original; **begin** y **end** que almacenan los índices de comienzo y final de los datos dentro de **sm** que corresponden a el slice siendo calculado por el bloque. Para la colaboración entre los hilos del bloque contamos con el arreglo compartido $\mathbf{x}_{new}$, que ira almacenando los valores parciales calculados por los distintos hilos. La longitud de $\mathbf{x}_{new}$ es igual al tamaño del slice. Los tamaños de los slices serán siempre valores potencias de dos, esto es así para facilitar las operaciones que veremos a continuación.

Si queremos saber cual es el slice al que pertenece la fila $n$, podemos hacer $\lfloor n/slice\_size \rfloor$. Y si queremos obtener el offset dentro de ese slice de la misma fila $n$, asumiendo que el tamaño del slice es $2^m$, bastará solo con tomar los $m$ bits menos significativos de $n$ (la operación bitwise $n\&(m-1)$). Si slice_size no fuera una potencia de dos deberíamos calcular el offset como $n\,\%slice\_size$. Esto requeriría de muchos mas cálculos para obtener el offset de una fila relativo a su slice.

A continuación explicaremos el algoritmo. Inicialmente en la línea 3 se le asigna a $i$ el índice de hilo relativo al warp que lo contiene, sumado al índice de comienzo de datos para nuestro slice. Como podemos ver en la línea numero 9, cada hilo realizará solo una operación cada **warp_size** elementos contiguos de la matriz rala comenzando desde su índice inicial. Esto es idéntico a la partición

```
1.  shared x_new[slice_size ]
2.  proc sm_cuda_mult(sm, x, offset_row, offset_col, begin, end)
3.      i := begin + (thread_id & (warp_size -1))
4.      while (i < end)
5.          r := sm.rows[i]+offset_row
6.          c := sm.cols[i]+offset_col
7.          offset_x_new := r & (slice_size -1) // equivale a r % slice_size
8.          x_new[offset_x_new] +:= sm.vals[i] × x[c]
9.          i := i + warp_size
10.     endwhile
11. endproc
```

**Figura 4.3:** Pseudo-código de algoritmo para multiplicación de matrices COO

que hacemos de los jobs sobre los warps y nos motiva el hecho de lograr accesos a memoria global *coordinados*, ya que se pedirán datos a memoria que corresponden a direcciones contiguas cada vez que el warp quiera acceder a $rows$, $cols$ o a $vals$. De esta manera logramos aumentar el throughput de la memoria global ya que alcanzará solo con una transacción de memoria para que el warp obtenga los valores de los arreglos $rows$ y $cols$, y solo dos para los valores de $vals$.

Entre las lineas 5 y 8 tenemos el código que ejecuta una operación punto a punto, primero, en las lineas 5 y 6, se le asignan a $r$ y $c$ los índices de fila y columna reales del valor a multiplicar. En la línea 7 se calcula el índice dentro de $\mathbf{x}_{new}$ donde almacenaremos el resultado de la operación y finalmente en la línea 8 se multiplica la entrada de la matriz con el elemento de $\mathbf{x}$ correspondiente y se acumula el resultado en $\mathbf{x}_{new}$.

Podemos apreciar en la Figura 4.4 el patrón de lectura de cada warp y cuales son las operaciones que realizará cada hilo dentro del warp.

A continuación presentaremos el algoritmo para computar efectivamente los valores del vector $\underline{x}$ correspondientes a un slice.

| Patrón de lectura del warp | | | | | | | |
|---|---|---|---|---|---|---|---|
| threads | $t_0$ | $t_1$ | ... | $t_{31}$ | $t_0$ | $t_1$ | ... |
| rows | 0 | 0 | ... | 4 | 5 | 5 | ... |
| cols | 1 | 3 | ... | 19 | 3 | 9 | ... |
| vals | 0.5 | 0.5 | ... | 0.3 | 0.2 | 0.5 | ... |

| Operaciones por thread | |
|---|---|
| $t_0$ | $x^{new}[0]$ += x[1] * 0.5; |
| $t_0$ | $x^{new}[5]$ += x[3] * 0.2; |
| $t_1$ | $x^{new}[0]$ += x[3] * 0.5; |
| $t_1$ | $x^{new}[5]$ += x[9] * 0.5; |
| ... | |

**Figura 4.4:** Patrones de lectura sobre una matriz COO realizados por el algoritmo

### 4.2.2. Computando los Slices

Dividiremos la presentación del algoritmo en tres partes: la etapa de inicialización, la etapa de multiplicación y la etapa de reducción. Para poder comprender como funciona primero analizaremos la manera en la que se dividirá el cálculo de los slices en la grilla de ejecución.

Para computar los slices los dividimos en clases de equivalencia de la función "congruente modulo *color*". Luego calcularemos en orden las clases de equivalencia con cada llamada a kernel. Esto quiere decir que primero lanzaremos un kernel que calcula los slices congruentes a 0 modulo *color*, luego una vez completada su ejecución los congruentes a 1 modulo *color*, etc. Por lo que garantizamos que para cada clase de equivalencia se utilizarán valores "nuevos"para todas las columnas correspondientes a una clase de equivalencia menor. Esta es la parte pseudo Gauss-Seidel de nuestro algoritmo y logra reducir las iteraciones en hasta un 57 % con respecto al algoritmo Jacobi del motor híbrido de PRISM. Podemos apreciar en la Figura 4.5 un ejemplo de dicha estrategia.

De todas las estrategias que intentamos, la partición en clases de congruencia resultó ser la mas simple y elegante de todas. Además esta solución escala conforme agregamos más y más núcleos a nuestras GPUs. Esto se debe a que si nos aumentan el número de filas, podemos aumentar los colores y con ello reducir las

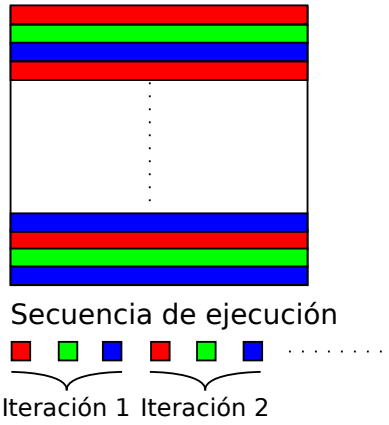



**Figura 4.5:** Secuencia de ejecución de los kernels

iteraciones. Y si aumentamos la cantidad de núcleos a nuestra GPU, podemos reducir el color hasta llegar a una utilización del 100 %.

Esta división por clases de equivalencia es muy simple de implementar ya que sólo requiere que en cada iteración pasemos al kernel una variable llamada *color* que indique cual es la clase de congruencia que esta siendo calculada. Para obtener un slice a partir de un numero de bloque y del color de la actual iteración realizamos la siguiente operación:

$$slice_{id} = block_{id} \times coloring + color$$

Donde $block_{id}$ se obtiene como proyección lineal de nuestra posición en la grilla de ejecución, *coloring* es la cantidad de clases de equivalencia y *color* es la clase que estamos calculando actualmente. La cantidad de bloques necesarios para cada clase de equivalencia se puede calcular como $\lceil n/(\textbf{slice\_size} \cdot coloring) \rceil$ donde $n$ es la longitud del vector solución.

Continuaremos presentando la función **slice_cuda_compute** que podemos ver en la Figura 4.6 y será la función invocada en cada llamada a kernel. Esta función posee muchos parámetros de entrada y solo los pertinentes al algoritmo están expresados por cuestiones de simplicidad.

Enumeraremos a continuación sus parámetros de entrada mas importantes y explicaremos brevemente cada uno :

- **jobs** : un arreglo que contiene todos los jobs resultantes de la matriz.

```
1.  proc slice_cuda_compute (jobs, job_starts, sms, x, b, diags, flag, color)
2.      slice_id := block_id * coloring + color
3.      if (slice_id ≥ num_slices) then
4.          return
5.      endif
6.      warp := thread_id / warp_size
7.      shared x_new[slice_size ]
8.      if (b ≠ ∅) then
9.          i := thread_id;
10.         while (i < slice_size )
11.             j := slice_id * slice_size + i
12.             if (j < n) then x_new[i] := b[j] endif
13.             i := i + thread_per_block
14.         endwhile
15.     else
...         // copia de las líneas 9-14 con la diferencia de que x_new[i] := 0
22.     endif
23.     ...
```

**Figura 4.6:** Pseudo-código correspondiente a la etapa de inicialización

- **job_starts** : un arreglo que contiene los índices de comienzo y fin en el arreglo **jobs** para los slices, tal que para el slice $i$ sus jobs estarán entre las posiciones $[job\_starts[i], job\_starts[i+1])$ del arreglo **jobs**.
- **sms** : un arreglo que contiene las matrices ralas que forman parte de la matriz original.
- **x** : el vector solución.
- **b** : el vector b de la ecuación en la Figura 2.2.
- **diags** : el vector de las diagonales *invertidas*.
- **flag** : puntero a una variable en memoria global a la cual se le debe asignar false cuando alguna fila no cumpla con el criterio de convergencia.
- **color** : variable que indica qué clase de equivalencia estamos calculando en este momento.

Comenzamos en la línea 2 calculando el id del slice correspondiente al bloque,

```
23.   ...
24.   syncthreads()
25.   i := job_starts[slice_id] + warp
26.   while (i < job_starts[slice_id+1])
27.      sm_cuda_mult(...)
28.      i := i + theads_per_block/warp_size
29.   endwhile
30.   syncthreads()
31.   ...
```

**Figura 4.7:** Pseudo-código de la etapa de multiplicación

de acuerdo a la fórmula presentada anteriormente. Seguido esto verificamos que no estemos calculando un slice no existente, esto puede suceder si las clases de equivalencia no tienen la misma cardinalidad. En la línea 6 obtenemos el número de warp que nos corresponde dado nuestro $\text{thread}_{id}$ y seguido esto declaramos la memoria compartida que mencionamos en el algoritmo de la Figura 4.3.

El código entre las lineas 9-14 y 16-21 es muy similar, uno corresponde a inicializar $x_{new}$ con los valores de b cuando este es no nulo y otro a inicializarlo con 0, por lo que explicaremos solo el primero. Lo que hacemos básicamente es realizar una lectura coordinada de la memoria, como hacíamos en el algoritmo de multiplicación de matrices para el arreglo *rows* por ejemplo, con la diferencia de que ahora debemos avanzar de a **threads_per_block** en lugar de **warp_size**, debido a que éste codigo será ejecutado por todos los hilos del bloque. En la línea 9 obtenemos nuestro $\text{thread}_{id}$ en la variable $i$, valor sobre el cual iteraremos, y que aumentaremos en **threads_per_block** en la línea 13. Esto nos asegura que los accesos a memoria global tanto para lectura como para escritura son coordinados, aumentando el throughput de memoria.

Esto completa la etapa de inicialización de la memoria compartida del bloque, ahora podemos pasar al algoritmo de multiplicación. En la Figura 4.7 podemos apreciar tal procedimiento. Comenzamos en la línea 24 sincronizando el bloque. Esto lo hacemos para asegurarnos de que en el arreglo compartido $x_{new}$ todos sus valores están inicializados correctamente, ya que si un warp termina antes que

```
31.   ...
32.   term := true
33.   i := thread_id;
34.   while (i < slice_size )
35.     j := slice_id * slice_size + i
36.     if (j < n) then
37.       x_new[i] := x_new[i] × diags[j]
38.       x_new[i] := (1 - omega) × x[j] + omega × x_new[i]
39.       if (|x_new[i]-x[j]|/x_new[i] > ε) then term := false endif
40.     endif
41.     i := i + theads_per_block
42.   endwhile
43.   term = __ballot(term)
44.   if (term ≠ (2^32-1) and (thread_id % 32) = 0) then flag := false endif
45.   return
```

**Figura 4.8:** Pseudo-código de la etapa de reducción

otro podríamos comenzar a multiplicar teniendo valores no inicializados. Luego cada warp accederá a un elemento distinto de $[job\_starts[i], job\_starts[i+1])$ de manera coordinada como lo venimos haciendo. La multiplicación del job se realiza llamando a **sm_cuda_mult** en la línea 27, procedimiento que mostramos en la Figura 4.3, que como dijimos era ejecutado por un solo warp a la vez.

Solo nos queda la etapa de almacenamiento o reducción en memoria principal, donde tenemos que verificar nuestra condición de terminación luego de aplicar la sobre-relajación presentada en la Figura 2.2. Antes de arrancar esa etapa, tenemos que sincronizar los hilos del bloque para asegurarnos que todas ellas hayan terminado de calcular sus trabajos. Esto lo hacemos mediante la barrera ubicada en la línea 30.

En la Figura 4.8 podemos apreciar la última parte de nuestro código. Comenzamos en la línea 32 por setear la variable local 'term'en true, que señalizará que todos los hilos del warp cumplen con la condición de terminación. Dado que nuestro objetivo es reducir la cantidad de escrituras a memoria global utilizamos la variable intermedia term, sobre la cual cada hilo indicará si existe una fila que

no cumpla con la condición de terminación. Luego en la línea 43 utilizamos una función propia de CUDA, que al ser llamada con una dato booleano tiene el efecto de retornar una mascara de bits con un 1 en la i-ésima potencia de dos si el dato en el i-ésimo hilo del warp es true y 0 en caso contrario. Esto significa que todos los hilos del warp verán el mismo resultado para la línea 43. Luego solo un hilo del warp escribirá 'flag'en caso de que la condición de terminación no se cumplió para alguna fila.

Como hicimos en la etapa de inicialización en la línea 33 asignamos a i, la variable de iteración, el número de nuestro $\text{thread}_{id}$. De hecho lo único que cambia en este bucle con respecto a los bucles de inicialización se encuentra comprendido entre las líneas 37-39. En la línea 37 multiplicamos por la diagonal invertida. En la línea 38 aplicamos la sobre-relajación de $\text{x}_{new}$[i], y en la línea 39 comprobamos si se viola el criterio de convergencia para la fila en cuestión.

Con esto concluiría el análisis del código GPU para el caso normal, en donde el tamaño de todos los vectores y matrices entran en memoria del dispositivo.

### 4.2.3. Llamadas desde CPU

Ahora solo nos queda analizar como desde la CPU debemos realizar las llamadas a kernel para lograr completar una iteración.

Primero debemos copiar todos nuestros arreglos y matrices ralas a memoria de dispositivo. Una vez que están todas nuestras estructuras ya estén copiadas en el dispositivo, podemos llamar a los kernels de la GPU para que realicen sus operaciones. El algoritmo esta ilustrado en la Figura 4.9. En el mismo podemos observar como llamamos a los sucesivos **slice_cuda_compute** con su grilla de ejecución correspondiente con el parámetro <**grid**>.

Por lo que podemos apreciar en las lineas 2 y 9 de la figura, las razones por las cuales terminaremos el algoritmo serán porque todas las filas cumplen con los criterios de convergencia, o bien porque el algoritmo falló en converger luego de max_iters iteraciones.

Cabe notar que mientras la GPU esta realizando cálculos la CPU no esta haciendo nada útil con su capacidad de cómputo. Esto no es una razón para alarmarse ya que nuestro algoritmo es el que no admite tal paralelismo.

```
1. ...
2. while (iters < max_iters)
3.     flag := true
4.     color := 0
5.     while (color < colors)
6.         slice_cuda_compute<grid>(...)
7.         color := color + 1
8.     endwhile
9.     if (flag = true) then break endif
10.    i := i + 1
11. endwhile
12. ...
```

**Figura 4.9:** Pseudo-código host

Así es como concretamos el análisis de nuestra implementación GPU del algoritmo Pseudo Gauss-Seidel Over-relaxed que llamamos *directo*. En la siguiente sección analizaremos nuestra estrategia para poder resolver sistemas que antes no eran factibles debido a el gran tamaño requerido para almacenar las estructuras de datos necesarias.

## 4.3. Algoritmo Pseudo Gauss-Seidel Partido

En esta sección analizaremos nuestra estrategia para poder aplicar Gauss-Seidel Over-relaxed cuando nuestras estructuras de datos no entran en memoria del dispositivo. Para este fin emulamos el mecanismo utilizado por el sistema operativo para poder ejecutar procesos que consumen giga-bytes de memoria utilizando solo cientos de mega-bytes de memoria física en lo que se denomina **paginación**.

La paginación en el CPU es básicamente utilizar una memoria secundaria (como el disco duro) para almacenar regiones de la memoria principal (RAM) que no están siendo utilizadas actualmente. Luego, cuando un proceso desea acceder a una posición de memoria que se encuentra almacenada en la memoria secundaria, se genera lo que se denomina un *fallo de página*. Esto hace que el sistema operativo

ubique en memoria principal dicha página faltante para que el proceso pueda continuar su ejecución, moviendo alguna otra página en memoria principal a memoria secundaria si no hay suficiente espacio disponible.

La principal diferencia es que nosotros manejaremos las memorias de manera explicita, antes de que ocurra un fallo de página. Esto se debe a que conocemos los patrones de acceso a memoria global de nuestro kernel. Adicionalmente, para los arreglos que leeremos solo una vez por iteración y de manera coordinada, como los arreglos b y diags, utilizaremos una nueva capacidad agregada recientemente por NVIDIA denominada *Unified Virtual Address*. Con UVA podemos acceder desde un puntero en GPU a memoria *pin-locked* en la máquina host. Esta lectura de memoria hará que la información sea copiada implícitamente por el hardware del dispositivo a memoria global del mismo. Pin-locked es la denominación que se da a una región de memoria en la cual tenemos la garantía de que no será paginada al $swapfile$ por el sistema operativo.

### 4.3.1. Paginación con Streams

La diferencia entre el método directo y el método partido es que el último no posee en memoria global los vectores x (solo para escritura), b y diags. Ya que estos son accedidos una sola vez, los accederemos utilizando UVA como ya mencionamos. Pero para leer x, lo que haremos será dividirlo en segmentos que sean lo mas grande posible tal que dos segmentos quepan en memoria global de la GPU. Estos dos segmentos que caben en la memoria de GPU son los que utilizaremos para la paginación.

Utilizaremos uno de los dos segmentos para realizar los cálculos. Concurrentemente a la ejecución del kernel de multiplicación realizaremos una transferencia de memoria desde CPU a GPU del siguiente segmento a computar en el segmento que no está siendo utilizado por el GPU. Esto quiere decir que solaparemos cálculos con transferencia de memoria, práctica altamente recomendada para aplicaciones CUDA.

La manera que tiene CUDA de proveernos de paralelismo de kernels es a través de *streams*. Podemos pensar a cada stream como una cola FIFO de llamadas a kernel, donde se irán tomando trabajos de manera ordenada hasta que las colas

| Scheduling de los kernels en los streams | | | | |
|---|---|---|---|---|
| $stream_0$ | $copy(segment_0)$ | $compute(segment_0)$ | $copy(segment_2)$ | ... |
| $stream_1$ | - | $copy(segment_1)$ | $compute(segment_1)$ | ... |

**Figura 4.10:** Solapación entre computación y transferencia de memoria

estén vacías. Esta es la herramienta que utilizamos para poder hacer transferencias de memoria y computación con la GPU concurrentemente. Podemos apreciar en la Figura 4.10 como es que se encolan los distintos kernels.

### 4.3.2. Unified Virtual Address

Una vez que resolvimos qué hacer con el único arreglo al cual accedemos de forma irregular podemos revisar las soluciones por medio de UVA. Como ya adelantamos previamente, esta técnica nos permite tener regiones de memoria host que sean visibles desde CPU y GPU. La forma en que se logra esto es por medio del manejador de memoria de la placa, que cuando detecta que una dirección de memoria se refiere a memoria host realiza la copia de los datos hacia la memoria del GPU de manera transparente para el kernel y el programador.

Dado que los accesos a los arreglos b, diags y x (solo para el caso de la reducción) son coordinados, realizaremos un traspaso de memoria cada cierta cantidad de valores, aprovechando al máximo cada byte transportado por el puerto PCI Express.

### 4.3.3. Análisis del Algoritmo

Lamentablemente, una vez que nuestros datos no caben en memoria GPU golpeamos inevitablemente un límite real en cuanto a GPU-computing se refiere. Pero esto no es razón para alarmarse ya que se espera que en el futuro la memoria de la CPU y la GPU se encuentren unificadas. Por lo tanto no habrá que traspasar datos ni preocuparse por que estos no quepan en memoria como hacemos nosotros.

Nuestra solución no produce tan buenos resultados como en el método directo, sin embargo logramos obtener speedups de alrededor de 3x y 4x. Analizando las gráficas obtenidas por el NVIDIA Visual Profiler, podemos concluir que hemos hecho lo posible para maximizar la utilización del puerto PCI Express de nuestro

host, y será imposible lograr avances sustanciales en el rendimiento del algoritmo sin mover el foco de nuestra optimización a una reformulación algorítmica del problema.

Anteriormente no hacían falta arreglos para almacenar valores intermedios, ya que los mismos se almacenaba en $x_{new}$ que formaba parte de la memoria compartida del bloque. Pero ahora que debemos ejecutar varios kernels de multiplicación para calcular un color, nos vemos obligados a almacenar los valores intermedios generados por cada kernel de multiplicación en un arreglo en memoria global del dispositivo que llamaremos $\underline{x}_{temp}$. Podemos calcular el tamaño de dicho vector temporal sabiendo los colores mediante la siguiente expresión:

$$sizeof(\underline{x}_{temp}) = \left\lceil \frac{\left\lceil \frac{n}{\mathbf{slice\_size}} \right\rceil}{colors} \right\rceil \cdot \mathbf{slice\_size} \cdot 8 \tag{4.1}$$

Luego de la ejecución de los kernels de multiplicación debemos ejecutar un kernel de reducción, que escribirá los valores de $\underline{x}_{temp}$ al vector $\underline{x}$. Este kernel tiene una duración aproximadamente igual a la suma de los kernels de multiplicación para cada color y será aún mayor en los casos en que se utiliza el vector $\underline{b}$.

Por todo lo anteriormente presentado, concluimos que para optimizar el método partido debemos contar o bien con una arquitectura GPU/CPU diferente o explorar la programación multi-GPU. De todas maneras predecimos que golpearemos el límite del ancho de banda del puerto PCI Express antes de tener que afrontar cualquier otro inconveniente.

## 4.4. Librería *libcudasles*

Como sección final de este capítulo presentaremos la librería *libcudasles* (CUDA Sparse Linear Equation Solver). Esta librería surgió como consecuencia de nuestro estudio para acelerar los operadores de model checking probabilístico. La razón por la cual ubicamos nuestro código en una librería aparte es porque creemos que la cantidad de aplicaciones que podrían estar siendo actualmente limitadas en capacidad de cómputo y que utilizan un esquema de codificación

similar al nuestro amerita el hecho. Otra razón es la inexistencia de una librería especifica para resolver sistemas lineales con múltiples matrices simultáneamente en GPU.

Actualmente esta librería implementa el método Pseudo Gauss-Seidel Over-relaxed normal y partido. Con distintas heurísticas para acomodar su ejecución en un solo dispositivo. Creemos que si la misma prueba ser de utilidad no pasará mucho tiempo antes que veamos contribuciones que agreguen otros algoritmos y la capacidad de hacer uso de múltiples GPUs.

### 4.4.1. Interfaz

Esta librería esta pensada para ser enlazada estáticamente en la compilación de nuestro programa. Su API presenta solo un punto de entrada que es la llamada a una función C llamada **libcudasles_solve**. Antes de mostrar su signatura vamos a definir dos estructuras de datos que serán necesarias en la misma

#### 4.4.1.1. COOSparseMatrix

Esta es nuestra representación de matrices ralas COO presentada en el Listado 4.1. El parámetro $n$ indica la cantidad de filas que ocupa (la altura) y $width$ la cantidad de columnas (el ancho). Los arreglos $non\_zeros$, $rows$ y $cols$ son los que contienen las coordenadas de los valores no nulos, y su longitud esta dada por $nnz$.

**Listing 4.1:** Estructura de Matriz Rala COO

```
typedef struct COOSparseMatrix
{
        int n;
        int nnz;
        int width;

        double *non_zeros;
        unsigned int *rows;
        unsigned int *cols;
} COOSparseMatrix;
```

#### 4.4.1.2. COOSparseMatrixInst

Como ya mencionamos anteriormente, los MTBDDs guardan una sola copia por cada elemento equivalente en la imagen de la función que representan. Por lo que en nuestra codificación híbrida se guardaran un conjunto de matrices ralas mucho menor al numero real de matrices ralas.

Con la estructura **COOSparseMatrixInst** debemos especificar las distintas instancias de una matriz rala dentro de la matriz original. Esto sería un paso anterior a calcular los trabajos, tarea que recae sobre la librería. Podemos ver los campos de esta estructura en el Listado 4.2. Allí *sm_idx* es el índice dentro del arreglo de matrices al cual hace referencia, y *row_offset* junto con *col_offset* indican la posición dentro de la matriz original de esta instancia.

**Listing 4.2:** Estructura de Instancia de Matriz Rala

```
typedef struct {
        int sm_idx;

        int row_offset;
        int col_offset;
} COOSparseMatrixInst;
```

#### 4.4.1.3. libcudasles_solve

Ahora sí estamos en condiciones de presentar nuestra interfaz. Podemos ver la signatura de nuestra función en el Listado 4.3.

Los argumentos que toma son: $n$, la cantidad de filas de nuestra matriz; *soln*, $b$ y *diags* representan a los vectores $\underline{x}$, $\underline{b}$ y $\underline{diags}$ respectivamente; *sparse_matrices* es el arreglo que contiene las distintas matrices ralas; *instances* es el arreglo que contiene las distintas instancias de las matrices ralas; *omega* es nuestro parámetro $\omega$ en la sobre-relajación, *term_crit* nos indicará si el criterio de terminación es absoluto o relativo y *max_iters* indica el número máximo de iteraciones que debemos realizar.

**Listing 4.3:** Estructura de Instancia de Matriz Rala

```
extern int libcudasles_solve(int n,
                             double *soln,
                             double *b,
                             double *diags,
                             COOSparseMatrix **sparse_matrices,
                             int sparse_matrices_n,
                             COOSparseMatrixInst *instances,
                             int instances_n,
                             double omega,
                             int term_crit,
                             int max_iters);
```

# Capítulo 5

# Resultados

En este último capítulo presentaremos los resultados de nuestra implementacion en distintos casos de estudio. Los modelos y propiedades que hemos utilizado para obtener los resultados experimentales forman parte del conjunto de benchmarks que provee PRISM en su sitio web http://www.prismmodelchecker.org/benchmarks/. Hemos utilizado los modelos DTMC, CTMC, y las propiedades de los mismos que para su verificación requieren la solución de sistemas de ecuaciones lineales. Hemos dejado de lado las propiedades que requerían menos de 5 segundos para ser verificadas por nuestra implementación.

## 5.1. Análisis de Performance

La estructura de nuestro algoritmo se puede dividir en dos etapas: una de inicialización o de setup donde se crean las estructuras de datos necesarias y otra de multiplicación. Solo nos hemos dedicado a paralelizar esta segunda parte, ya que por lo general es la mas pesada computacionalmente y la que presenta mayor paralelismo de datos. Sin embargo hemos implementado esta segunda etapa para que corra en el CPU para comparar el el performance de nuestra implementación en GPU.

Ocasionalmente mencionaremos también las diferencias entre el algoritmo pseudo Gauss-Seidel ya implementado dentro del motor híbrido de PRISM. Si bien este algoritmo es diferente, es la única otra implementación disponible de un

algoritmo pseudo Gauss-Seidel.

Las mediciones de tiempo se llevaron a cabo utilizando la función `times(struct tms *)` de la librería `"sys/times.h"`. Evaluamos un total de 31 casos y para cada caso obtuvimos los valores de las corridas en CPU, la versión directa en GPU y la versión partida en GPU. Para cada caso también variamos el coloreo entre 2 y 8 inclusive. Por último, para cada caso corrimos la versión pseudo Gauss-Seidel de PRISM. Podemos encontrar las especificaciones del entorno de prueba utilizado para correr los experimentos en la Figura 5.2.

Podemos expresar el tiempo total empleado por el algoritmo como $T_{psor} = T_{setup} + T_{iters}$ donde $T_{setup}$ es el tiempo que se demoró en las tareas de inicialización y $T_{iters}$ es el tiempo que se demoró en resolver el sistema de ecuaciones. El valor de $T_{setup}$ no cambiará para nuestras implementaciones CPU y GPU. Por lo que para obtener el speedup solo nos interesará el tiempo que se demoró en resolver el sistema de ecuaciones, que es la parte paralelizada del mismo. Una vez dicho esto estamos en condiciones de expresar nuestra fórmula de speedup como:

$$\text{Speedup} = \frac{T_{\text{iters CPU}}}{T_{\text{iters GPU}}}$$

**Figura 5.1:** Fórmula de Speedup

| **CPU** | |
|---|---|
| Procesador | Intel(R) Core(TM) i7 CPU 950 a 3.07GHz |
| Memoria | 16 GB DDR3 @ 1333 MHz |
| **GPU** | |
| Procesador | Kepler GK110 a 706MHz |
| Memoria | 5 GB DDR5 |
| Interfaz | PCI Express 2.0 |

**Figura 5.2:** Entorno de Prueba

## 5.2. Casos de Estudio

En esta sección enumeraremos los distintos modelos utilizados junto con una breve descripción de los mismos. Presentamos en la Figura 5.3 la morfología de las distintas matrices de transiciones.

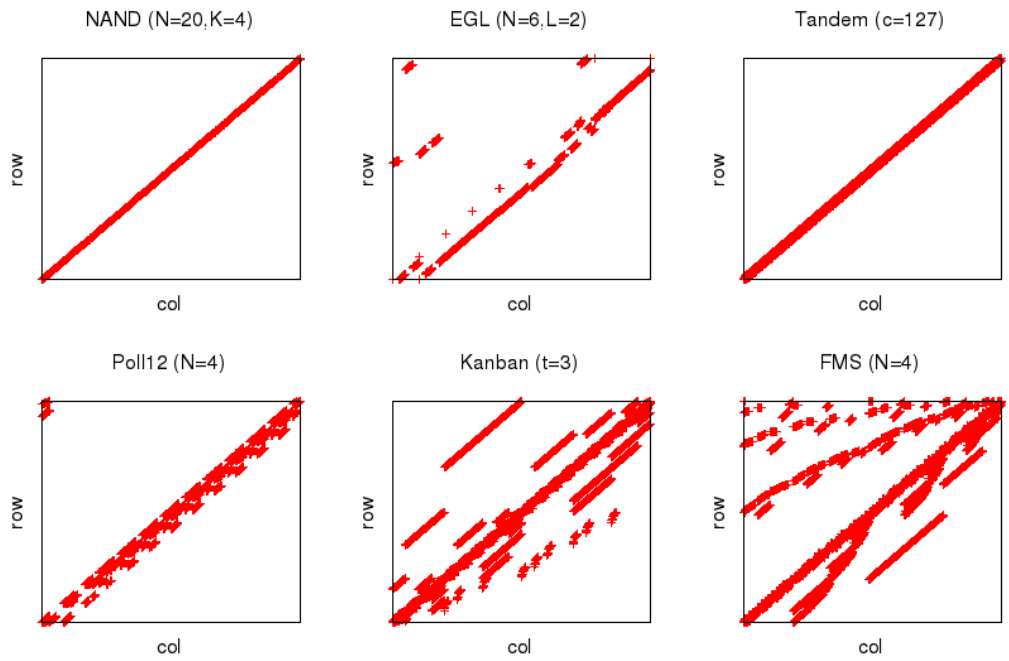



**Figura 5.3:** Matrices de transición de estados de los modelos analizados

Para cada propiedad verificada agregaremos una tabla de valores que nos ayudaran a visualizar las características de nuestro algoritmo. A continuación presentamos una breve explicación de las cabeceras de dichas tablas:

- **Configuración**: Contiene todos los parámetros variables de la propiedad y del modelo así como también la cantidad de filas de la matriz de transición (N) y el tamaño en memoria de las estructuras de datos necesarias para nuestro algoritmo.
- **Iteraciones**: Posee dos columnas $\text{PSOR}_{old}$ y $\text{PSOR}_{new}$, que indican la cantidad de iteraciones que requeridas en la implementación de pseudo Gauss-Seidel de PRISM y la implementación de *libcudasles* respectivamente.

- **Tiempo**: Posee dos columnas CPU y GPU que indican el tiempo de corrida $T_{iters}$ de las implementaciones de nuestro algoritmo para CPU y GPU respectivamente. Estos tiempos corresponden a una corrida con *color* igual a la columna **Color**.
- **Speedup**: En esta columna agregamos el resultado de la fórmula 5.1 del color mas rápido.
- **Color**: Indica cual fue el coloreo mas eficiente de todos los corridos.

Estas tablas se encontrarán acompañadas de dos gráficos: uno que mostrará una gráfica perteneciente a la columna **Tiempo**; y otro que mostrará como varían las iteraciones en base al color escogido, agregando también las iteraciones para el algoritmo pseudo Gauss-Seidel PRISM bajo el nombre CPU.

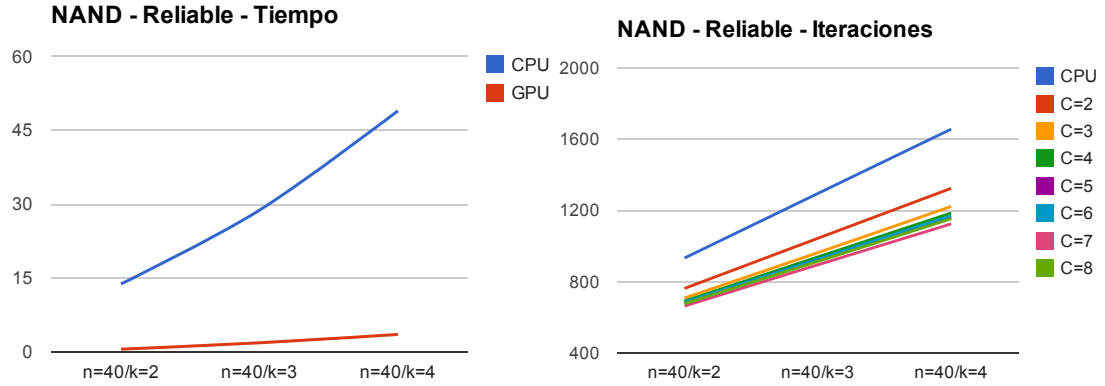
### 5.2.1. NAND Multiplexing

Esta es una técnica para construir elementos de computación confiables a partir de dispositivos no confiables. A medida que la escala de fabricación de circuitos integrados se hace mas pequeña, la presencia de fallas físicas deja de ser un factor despreciable. Es por esta razón que la necesidad de nuevas arquitecturas tolerantes a fallas es muy importante. El estudio de esta técnica se realizo en [NPKS05] y tomaremos una propiedad de este estudio.

Para este modelo verificaremos una propiedad que expresa que la probabilidad de que menos del 10 % de las output del sistema sean erróneas se cumpla, es decir que el sistema sea confiable.

| Configuración | | | | Iteraciones | | Tiempo (seg) | | | |
|---|---|---|---|---|---|---|---|---|---|
| n | k | N | Tamaño | $PSOR_{old}$ | $PSOR_{new}$ | CPU | GPU | Speedup | Color |
| 40 | 2 | 2003082 | 50.1 MB | 932 | 761 | 13.75 | 0.5 | 28 | 2 |
| 40 | 3 | 3001302 | 72.9 MB | 1295 | 1043 | 28.73 | 1.79 | 16 | 2 |
| 40 | 4 | 3999522 | 95.8 MB | 1657 | 1324 | 48.92 | 3.49 | 14 | 2 |

**Figura 5.4:** Tabla de resultados del modelo NAND




**(a)** Tiempo C=2 **(b)** Iteraciones

**Figura 5.5:** En 5.5a podemos ver la comparación entre los tiempos de corrida secuenciales y paralelos para nuestro algoritmo cuando el color escogido es igual a 2; en 5.5b podemos observar como varían las iteraciones con respecto al color, en este caso CPU hace referencia al algoritmo GS original de PRISM.

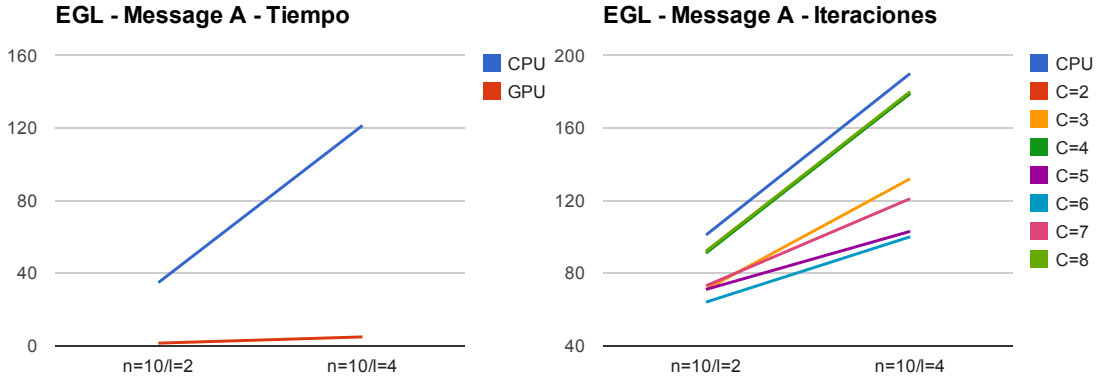
### 5.2.2. EGL Probabilistic Contract Signing

Este es un protocolo presentado en [EGL85] que resuelve el problema de intercambiar datos entre varios participantes cuando estos se desconfían entre si. *Fair exchange* es la clase de problemas al intercambiar datos en los cuales se garantiza que todos los participantes obtienen lo que desean o ninguno lo hace. *Contract signing* es un caso particular del intercambio justo, en la cual los participantes intercambian compromisos a un contrato. Los signing protocols son una parte esencial de la infraestructura e-commerce ya que garantizan la firma digital de un contrato. Podemos encontrar el estudio de su verificación con PRISM en [NS06].

Para este caso, calcularemos el valor esperado de mensajes que A necesita para conocer un par de secretos de B dado que B ya conoce un par de secretos de A.

| Configuración | | | | Iteraciones | | Tiempo (seg) | | | |
|---|---|---|---|---|---|---|---|---|---|
| n | l | N | Tamaño | $PSOR_{old}$ | $PSOR_{new}$ | CPU | GPU | Speedup | Color |
| 10 | 2 | $66\times10^6$ | 1.5 GB | 101 | 64 | 34.75 | 1.39 | 25 | 6 |
| 10 | 4 | $149\times10^6$ | 3.4 GB | 190 | 100 | 121.34 | 4.83 | 25 | 6 |

**Figura 5.6:** Tabla de resultados del modelo EGL PCS




**(a)** Tiempo C=2 **(b)** Iteraciones

**Figura 5.7:** En 5.7a vemos una comparación entre los tiempos CPU y GPU con *color* = 2; en 5.7b observamos como varían las iteraciones con el color.

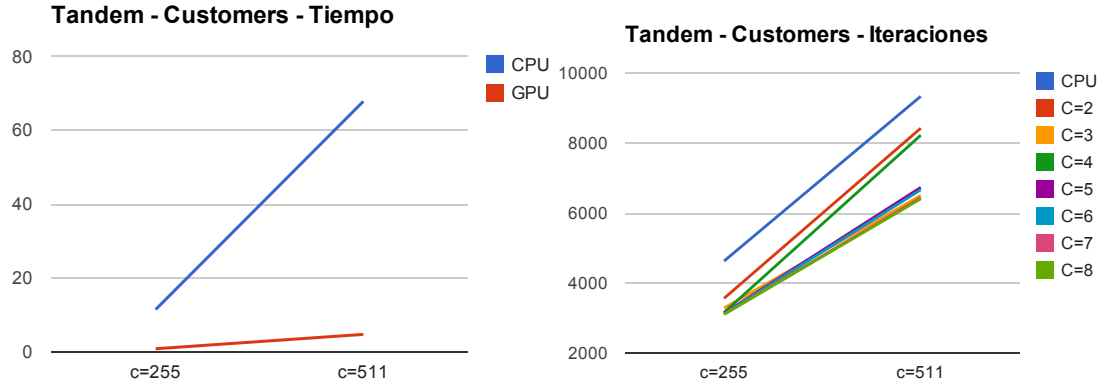
### 5.2.3. Tandem Queue

Las colas tándem o en línea sirven para modelar un servicio que esta conformado por un grupo ordenado de colas, podemos observar un estudio de su modelado en [HMKS99]. Cuando un cliente ingresa al sistema debe recorrer todas las colas formando en línea antes de considerarse 'servidos'. Las colas están presentes en nuestra vida diaria por ejemplo cuando hacemos streaming de un video, cuando pedimos una página web, etc. Hay varias preguntas interesantes que podemos hacer sobre estos modelos como: cual es el tiempo de espera promedio, cual es el throughput del sistema, cual es el valor promedio de clientes esperando en la cola y saber si el sistema es estable bajo una distribución de clientes simulada, entre otras preguntas.

En este caso, calcularemos el valor esperado de la cantidad de clientes en la red en un momento dado en una corrida infinita.

| Configuración | | | Iteraciones | | Tiempo (seg) | | | |
|---|---|---|---|---|---|---|---|---|
| c | N | Tamaño | $PSOR_{old}$ | $PSOR_{new}$ | CPU | GPU | Speedup | Color |
| 255 | 130816 | 2.2 MB | 4631 | 3290 | 11.4 | 0.78 | 14.6 | 3 |
| 511 | 523776 | 8.4 MB | 9342 | 6507 | 67.77 | 4.67 | 14.5 | 3 |

**Figura 5.8:** Tabla de resultados del modelo Tandem Queue




**(a)** Tiempo C=3 **(b)** Iteraciones

**Figura 5.9:** En 5.9a vemos una comparación entre los tiempos CPU y GPU con *color* = 3; en 5.9b observamos como varían las iteraciones con el color.

### 5.2.4. Cyclic Server Polling System

Este es otro modelo de colas, basado en [IT90], en el cual se modela un servidor que posee varias colas finitas, donde llegan los clientes. El servidor va revisando (*polling*) las colas y si hay un cliente esperando, la servirá. De lo contrario continuará revisando las colas en orden cíclico.

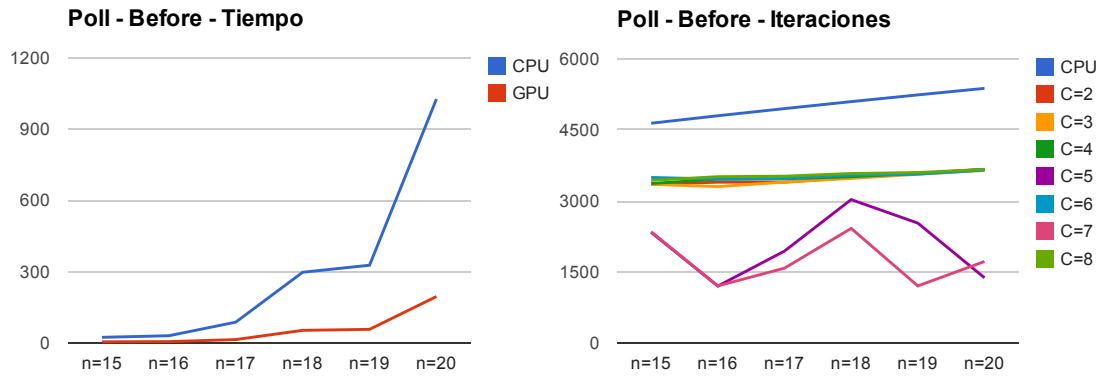
Para este modelo analizaremos dos propiedades relevantes: la probabilidad de que en una corrida infinita la estación 1 este esperando ser servida (*s1*) y la probabilidad de que la estación 1 sea atendida antes que la estación 2 (*before*). Este resultó ser el único modelo que no presenta un comportamiento predecible en el crecimiento de las iteraciones al aumentar los parámetros del modelo.

| Configuración | | | Iteraciones | | Tiempo (seg) | | | |
|---|---|---|---|---|---|---|---|---|
| n | N | Tamaño | $PSOR_{old}$ | $PSOR_{new}$ | CPU | GPU | Speedup | Color |
| 15 | 737280 | 18.3 MB | 4639 | 2351 | 24.34 | 5.47 | 4.4 | 7 |
| 16 | 737280 | 37.2 MB | 4797 | 1209 | 31.02 | 6.28 | 4.9 | 7 |
| 17 | 1572864 | 77.9 MB | 4948 | 1583 | 88.12 | 14.91 | 5.9 | 7 |
| 18 | 1572864 | 163.7 MB | 5095 | 2421 | 297.86 | 53.66 | 5.5 | 7 |
| 19 | 3342336 | 343.9 MB | 5237 | 1208 | 327.43 | 57.86 | 5.6 | 7 |
| 20 | 3342336 | 722.4 MB | 5375 | 1723 | 1025.86 | 195.78 | 5.3 | 7 |

**Figura 5.10:** Tabla de resultados del modelo CSPS para la propiedad 'Before'

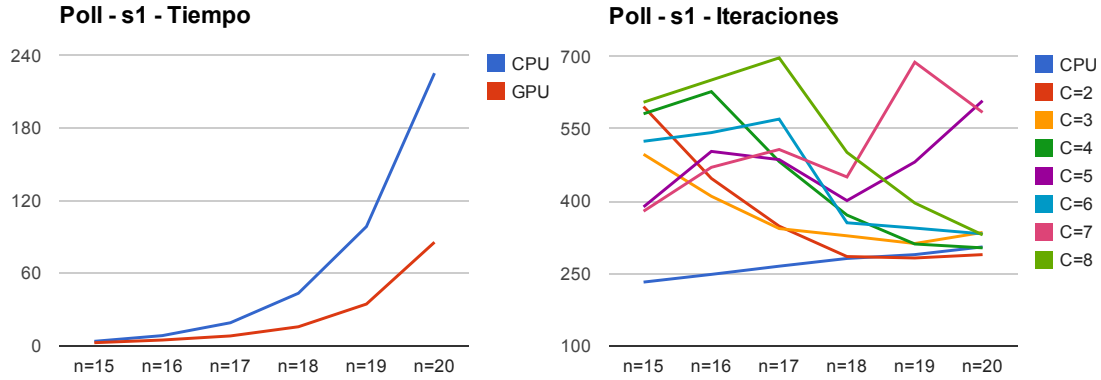
| Configuración | | | Iteraciones | | Tiempo (seg) | | | |
|---|---|---|---|---|---|---|---|---|
| n | N | Tamaño | $PSOR_{old}$ | $PSOR_{new}$ | CPU | GPU | Speedup | Color |
| 15 | 7077888 | 12 MB | 232 | 388 | 3.59 | 2.45 | 1.4 | 5 |
| 16 | 7077888 | 24.8 MB | 248 | 503 | 8.31 | 4.66 | 1.7 | 5 |
| 17 | $14.9 \times 10^6$ | 51.9 MB | 265 | 486 | 18.94 | 8.04 | 2.3 | 5 |
| 18 | $14.9 \times 10^6$ | 109 MB | 281 | 401 | 43.33 | 15.63 | 2.7 | 5 |
| 19 | $31 \times 10^6$ | 229.1 MB | 289 | 481 | 98.48 | 34.4 | 2.8 | 5 |
| 20 | $31 \times 10^6$ | 481.3 MB | 305 | 608 | 225.24 | 85.49 | 2.6 | 5 |

**Figura 5.11:** Tabla de resultados del modelo CSPS para la propiedad 's1'




**(a)** Tiempo C=7 **(b)** Iteraciones

**Figura 5.12:** En 5.12a vemos una comparación entre los tiempos CPU y GPU con *color* = 7; en 5.12b observamos como varían las iteraciones con el color.




**(a)** Tiempo C=5 **(b)** Iteraciones

**Figura 5.13:** En 5.13a vemos una comparación entre los tiempos CPU y GPU con *color* = 5; en 5.13b observamos como varían las iteraciones con el color.

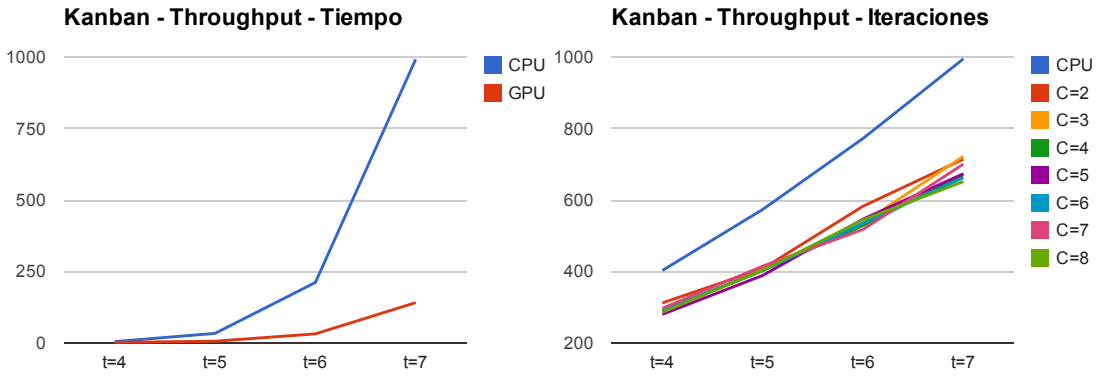
### 5.2.5. Kanban Manufacturing System

Kanban es un sistema de scheduling para lean manufacturing y producción just-in-time. En el mismo, la provisión o producción de una fábrica se determina en base a la demanda actual de los clientes. Esto permite responder progresivamente a los cambios en la demanda y así poder tolerar cambios bruscos de la misma. Este sistema se basa en el envío de señales (*tokens* en PRISM) a lo largo de la cadena de producción para indicar que hay faltante o ausencia de un elemento. Fue creado por la empresa Toyota y desde su implementación ha sido adaptado para ser aplicado en diversas áreas (management, lean software development, etc).

Este modelo está basado en el sistema presentado en [CT96]. En este caso calcularemos el valor esperado del throughput del sistema, variando la cantidad de tokens del mismo.

| Configuración | | | Tiempo (seg) | | Iteraciones | | | |
|---|---|---|---|---|---|---|---|---|
| n | N | Tamaño | $PSOR_{old}$ | $PSOR_{new}$ | CPU | GPU | Speedup | Color |
| 4 | 454475 | 7.9 MB | 402 | 285 | 3.25 | 0.54 | 6 | 8 |
| 5 | 2546432 | 40.2 MB | 573 | 402 | 32.01 | 4.62 | 6.9 | 8 |
| 6 | 11.2 $\times 10^6$ | 173.5 MB | 772 | 544 | 210.28 | 30.23 | 6.9 | 8 |
| 7 | 41.6 $\times 10^6$ | 637.2 MB | 995 | 651 | 992.19 | 139.75 | 7 | 8 |

**Figura 5.14:** Tabla de resultados del modelo Kanban




**(a)** Tiempo C=8 **(b)** Iteraciones

**Figura 5.15:** En 5.15a vemos una comparación entre los tiempos CPU y GPU con *color* = 8; en 5.15b observamos como varían las iteraciones con el color.

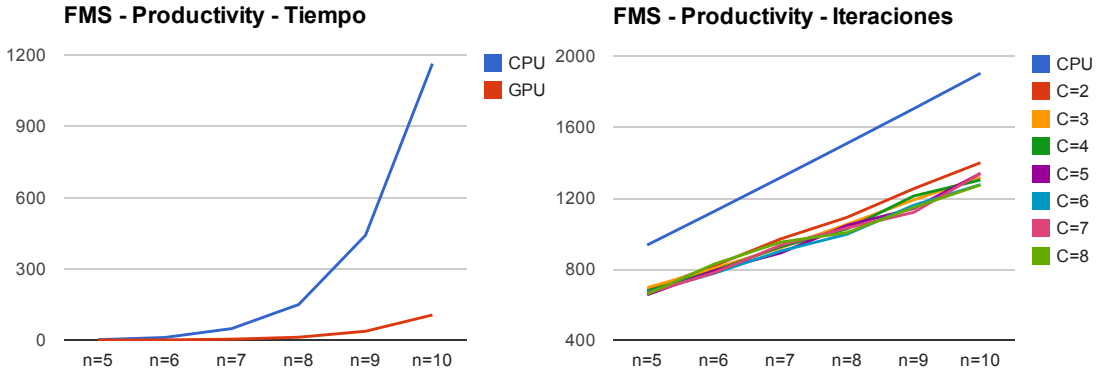
### 5.2.6. Flexible Manufacturing System

Este es otro sistema de manufactura flexible 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, máquinas de inspección) en distintos segmentos de producción interconectados que se envían mensajes entre si. En nuestro caso se modela la interacción de los nodos en este sistema para calcular su throughput. Nuestro modelo esta basado en la presentación hecha en [CT93].

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

| Configuración | | | Iteraciones | | Tiempo (seg) | | | |
|---|---|---|---|---|---|---|---|---|
| n | N | Tamaño | $PSOR_{old}$ | $PSOR_{new}$ | CPU | GPU | Speedup | Color |
| 5 | 152712 | 6.2 | 936 | 660 | 2.39 | 0.58 | 4.1 | 8 |
| 6 | 537768 | 13.7 | 1124 | 828 | 11.61 | 1.51 | 7.6 | 8 |
| 7 | 1639440 | 33 | 1315 | 952 | 49.02 | 4.54 | 10.8 | 8 |
| 8 | 4459455 | 81.1 | 1508 | 1006 | 150 | 12.65 | 11.8 | 8 |
| 9 | 11058190 | 187.7 | 1703 | 1148 | 441.4 | 37.8 | 11.6 | 8 |
| 10 | 25397658 | 412.4 | 1902 | 1274 | 1162.61 | 106.2 | 10.9 | 8 |

**Figura 5.16:** Tabla de resultados del modelo FMS




**(a)** Tiempo C=8 **(b)** Iteraciones

**Figura 5.17:** En 5.17a vemos una comparación entre los tiempos CPU y GPU con *color* = 8; en 5.15b observamos como varían las iteraciones con el color.

## 5.3. Método Directo

Como podemos observar en nuestros casos de estudio, nuestro algoritmo nos provee de una reducción sustancial de las iteraciones con respecto al algoritmo de PRISM. En algunos modelos esta reducción ha llegado a ser de hasta un 67 %. Si bien la cantidad de operaciones realizadas en cada iteración es el mismo para ambos algoritmos nuestra implementación en CPU no recorre el MTBDD, esto reduce el costo de cada iteración. Por lo que cuando hablamos de un speedup de 10x en nuestro algoritmo, contrastando la implementación CPU con la implementacion GPU, significa que obtuvimos un speedup aún mayor con respecto a

la implementación original de PRISM. Cabe notar que la reducción de iteraciones es igual para las implementaciones CPU y GPU de nuestro algoritmo.

El caso mas extremo de los que observamos es el perteneciente al modelo FMS de la Sección 5.2.6 en donde el speedup de nuestra implementación en GPU con respecto al algoritmo de PRISM llegó a 73x, logrando reducir el tiempo necesario para verificar la propiedad en el caso con el parámetro $n$=10 de 9110 segundos a solo 123.6 segundos. Mientras, el promedio de speedup en nuestros casos de estudio con respecto a la implementación de PRISM es de 26x con una desviación estándar de 20x. Esta desviación es tan grande debido a que para nuestra implementación en GPU el modelo CSPS de la sección 5.2.4 requiere de mas iteraciones para converger a una solución que la implementación de PRISM, por lo que el speedup obtenido es bastante pequeño. Esto se debe a que la cantidad de iteraciones depende del color que elijamos y de la morfología de la matriz de transiciones.

La razón por la cual nuestro algoritmo disminuye la cantidad de iteraciones con respecto a la implementación de PRISM es que logramos explotar más las relaciones entre las filas de la matriz al no realizar un barrido secuencial. Si en lugar de utilizar los grupos de colores realizáramos las multiplicaciones sobre clusters secuenciales de slices, no lograríamos explotar el hecho de que nuestras matrices tienen una morfología diagonal. Esta morfología implica que existe una interrelación entre las slices cercanas, lo que genera mejores aproximaciones al separar sus cálculos. Adicionalmente estamos partiendo las filas en slices de tamaño mas pequeños que los bloques de la implementación de PRISM lo que también aumenta la cantidad de valores nuevos que serán utilizados en la multiplicación matriz-vector.

Podemos estimar exitosamente cual será el color que resultará en menos iteraciones haciendo un pequeño preprocesamiento de la matriz de transiciones. Lo que hacemos es contar cuantas operaciones punto a punto garantizamos que utilizarán un valor calculado en la misma iteración. Esto nos da una idea de cuan grande es la proporción de operaciones que aprovecharán las dependencias entre filas para generar mejores aproximaciones. Luego es seguro escoger el color que haya resultado en un número mas alto de dichas transiciones.

Un detalle que vale la pena mencionar es que al aumentar el color también

aumentamos la cantidad de kernels que se ejecutan por iteración, y cada ejecución acarrea un overhead que debemos pagar. Finalmente, la arquitectura CUDA sobre la que corremos nos impone otro límite para el color. Esto es así dado que cada vez que aumentamos el color disminuimos la cantidad de bloques que se crearán para cada kernel de la iteración, eventualmente esa cantidad de bloques harán que no se logre una ocupación total de la placa provocando un decaimiento en el performance de nuestro algoritmo.

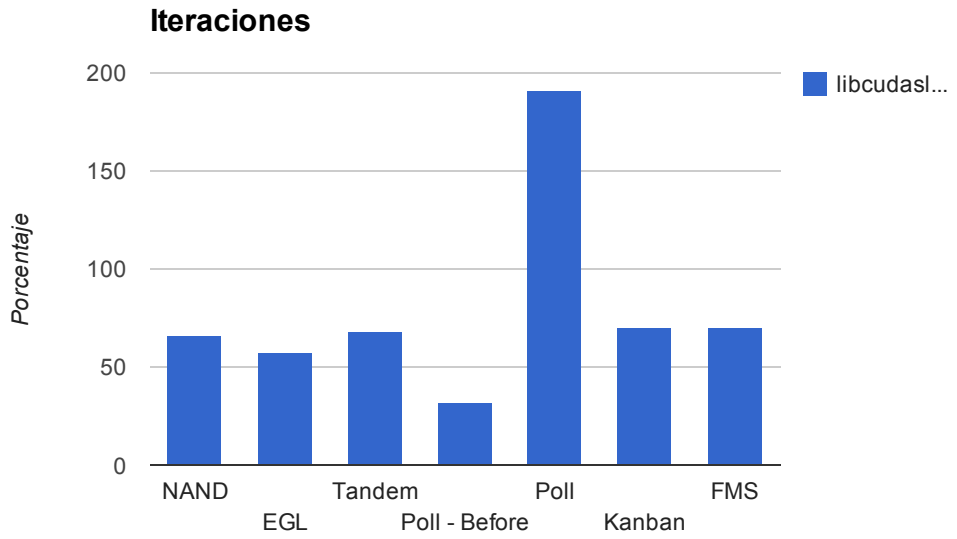



**Figura 5.18:** Proporción de iteraciones realizadas por libcudasles con respecto a la implementación de PRISM.

Hemos agregado las Figuras 5.18 y 5.19 que muestran graficos comparando el speedup y las iteraciones de nuestra implementación con la del motor híbrido de PRISM. En 5.18 podemos observar como varían las iteraciones ubicando en el eje vertical la proporción de iteraciones con respecto a PRISM, dado que los valores varían mucho en escala. En 5.19 podemos observar como es que influye esta reducción de iteraciones en algunos modelos para obtener un mayor speedup, pero también podemos observar que no es el único factor determinante del speedup como en el caso de FMS.

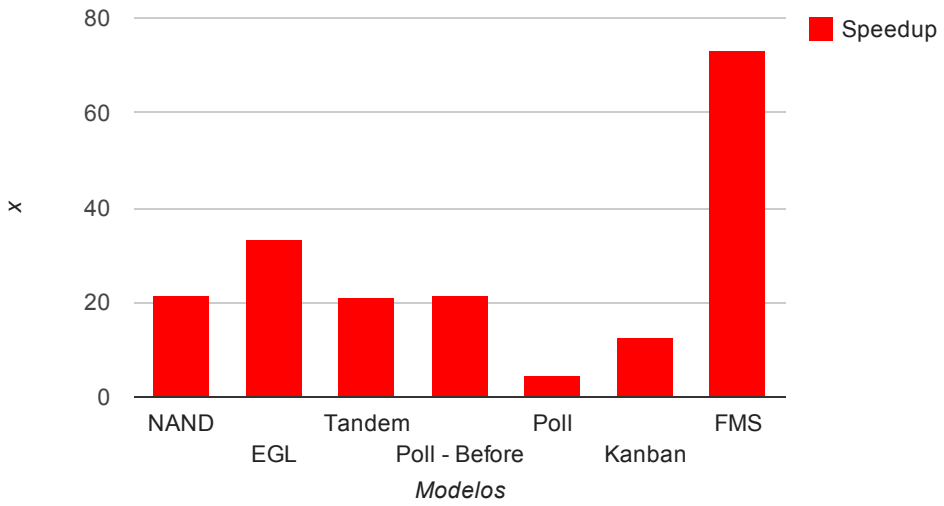



**Figura 5.19:** Speedup obtenido en los distintos modelos utilizando libcudasles.

## 5.4. Método Partido

Analizaremos a continuación dos ejemplos para el método partido. Los casos que utilizaremos serán el modelo Kanban con su propiedad *Throughput* en la Figura 5.20 y el modelo Cyclic Server Polling System con su propiedad *Before* en la Figura 5.21.

Dado que en el kernel partido estamos utilizando UVA, toda escritura al vector **x** y toda lectura de los vectores **diags** y **b** incurren en una demora adicional debido a que se utilizará el puerto PCI Express para transferir los datos desde y hacia la memoria del CPU. La velocidad de transferencia del puerto PCI Express 2.0 utilizado en nuestro ambiente de prueba es de 8 GBytes/s. De acuerdo a los resultados del profiler *nvprof* estamos utilizando alrededor de 5.8 GB/s, lo cual no es muy alejado del limite teórico del puerto. Lamentablemente no contamos con un host con PCI Express 3.0 para poder correr nuestras pruebas, pero estimamos que el desempeño de nuestro algoritmo mejoraría en por lo menos 1.5x en los casos donde no se utiliza el vector **b** y 1.6x en los casos que si es utilizado.

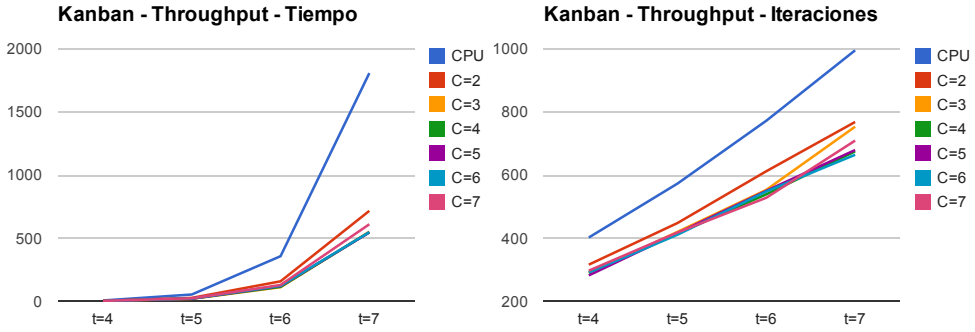



**Figura 5.20:** Metodo Partido con Kanban

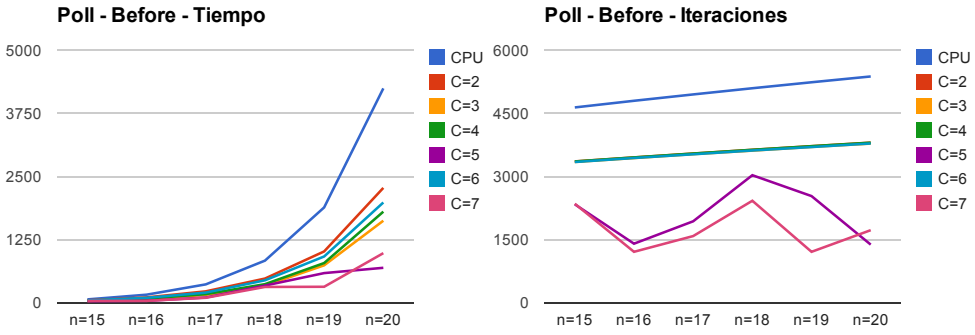



**Figura 5.21:** Metodo Partido con Cyclic Server Polling System

La verificación de la propiedad utilizada en el modelo Kanban no requiere del vector $\underline{\mathbf{b}}$, a diferencia de la propiedad del modelo CSPS.

Otra diferencia con respecto al metodo directo reside en que debemos ejecutar los kernels de multiplicación necesarios y el kernel de reducción para cada color. Este overhead de ejecución de kernels aumenta a medida que aumentamos los colores, por lo que utilizaremos el menor color que permita que $\underline{x}_{temp}$ y las otras estructuras de datos quepan en la memoria del dispositivo.

## 5.5. Consumo de Memoria

El consumo de memoria de nuestro algoritmo es similar al de la implementación de PRISM. Si bien el formato de las matrices ralas utilizadas por nuestra implementación no es el mismo, la diferencia en tamaño es tan pequeña que podemos despreciarla en nuestro análisis. De hecho la única diferencia radica en las estructuras de datos requeridas para almacenar los jobs.

En la Figura 5.22 podemos observar una situación que se cumple para todos nuestros casos de estudio. Los datos indican que a medida que aumentamos el tamaño de nuestros modelos la proporción de memoria dedicada a las estructuras de datos jobs y job_starts decrece.

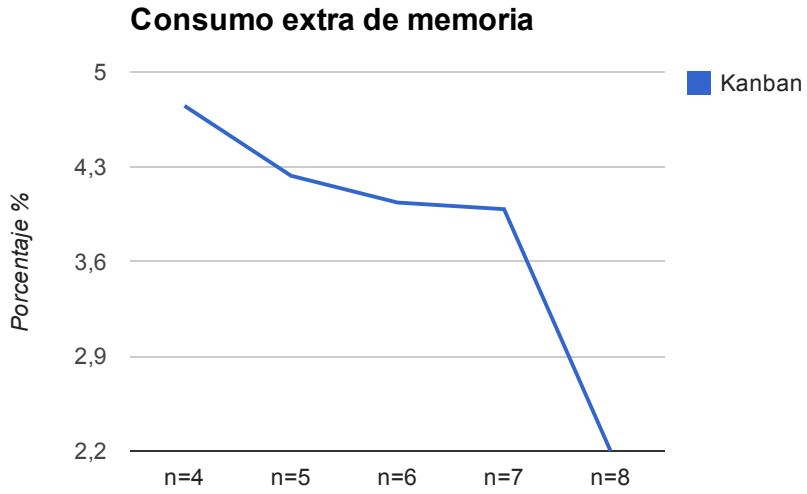



**Figura 5.22:** Consumo extra de memoria debido a jobs y job_starts para el modelo Kanban

En cuanto al metodo partido, el consumo extra de memoria es considerable. Para el metodo directo no requerimos de un vector para almacenar los productos intermedios, ya que se almacenan en memoria compartida hasta que finalizamos de calcular un slice. Pero para el metodo partido necesitamos almacenar estos productos intermedios en memoria global del dispositivo antes de escribirlos en el vector solución al finalizar los distintos kernels de multiplicación. El tamaño extra de memoria dependerá únicamente de la cantidad de filas de la matriz y del color como mostramos en la Figura 4.1. A modo de ejemplo tenemos en la Figura 5.23 un gráfico donde podemos ver como varía el tamaño del vector requerido con respecto a los colores para el modelo Kanban.

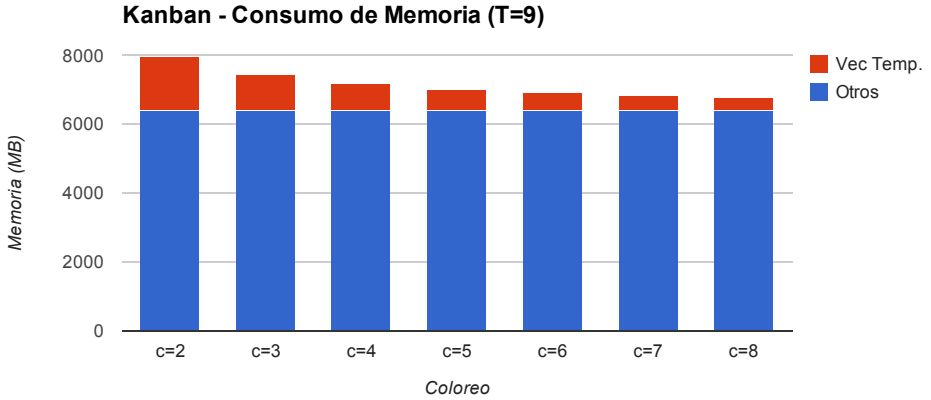



**Figura 5.23:** Consumo de memoria del vector temporal para el modelo Kanban

# Capítulo 6

# Conclusiones

Hemos presentado en este trabajo un algoritmo para la resolución de sistemas de ecuaciones lineales para ser aplicados en la verificación de propiedades de modelos probabilistas que corre en los dispositivos de arquitectura CUDA de Nvida. Este trabajo estuvo motivado por el creciente estudio de dichos dispositivos para resolver problemas que presentan paralelismo de datos y alto requerimiento computacional como también en la necesidad de acelerar los model checkers actuales para poder verificar modelos mas grandes que los que actualmente estamos limitados a modelar debido a la impracticidad de la verificación de propiedades sobre los mismos.

Trabajos similares han sido estudiados en [Tea13], [WB12] y [Lag11], dejando en claro que es un campo de estudio de importancia.

En el camino nos vimos limitados por la cantidad de memoria disponible en estos dispositivos, que impedía que aumentaramos el tamaño de los modelos a verificar, sin embargo pudimos sortear este obstaculo por medio de nuestro método partido. Hay que tener en cuenta que la memoria de estos dispositivos se duplica cada 2 años, lo que significa que el presente trabajo seguirá siendo relevante a medida que estas aumenten. Por otro lado se esta viendo en la industria la aparición de arquitecturas hibridas que estan compuestas por la fusión de CPU y GPU en un mismo chip como la del Sistema de Arquitectura Heterogenea (HSA) de ATI y la microarquitectura de los chips Haswell de Intel. Esto significa que en un futuro no muy lejano la CPU y el GPU compartirán el mismo espacio de memoria, lo que haría que nuestro método partido sea irrelevante y al mismo

tiempo que nuestro método directo sea mas prometedor.

Los resultados de performance obtenidos por nuestra implementación dejaron en claro su conveniencia, inclusive en la implementación CPU, sobre los algoritmos actuales utilizados por el model checker PRISM. La mayor limitación que hemos encontrado es el ancho de banda a memoria global del dispositivo, esta limitación es común en la mayoría de los programas GPU. Previendo esto, Nvidia mejorará el ancho de banda a memoria global que actualmente es de 250GB/s en la Tesla K20X a 1 TB/s con sus nuevos chips Volta agregando memoria en el mismo chip del GPU. Sin embargo hemos obtenido mas speedup del que esperabamos obtener en un principio, lo que deja en evidencia que el futuro de la arquitectura GPU es muy prometedor.

# Referencias


[CT93] G. Ciardo and K. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993. 78

[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. 77

[EGL85] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, 1985. 73

[HMKS99] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In B. Plateau, W. Stewart, and M. Silva, editors, *Proc. 3rd International Workshop on Numerical Solution of Markov Chains (NSMC'99)*, pages 188–207. Prensas Universitarias de Zaragoza, 1999. 74

[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. 75

[Lag11] Pablo Dal Lago. Paralelización de algoritmos para vericación simbólica de modelos probabilísticos. 2011. 86

[MS92] K. L. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In N. Suzuki, editor, *Shared Memory Multiprocessing*. MIT Press, 1992. 5

[NPKS05] Gethin Norman, David Parker, Marta Kwiatkowska, and Eep Shukla. Evaluating the reliability of nand multiplexing with prism. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24:2005, 2005. 72

[NS06] G. Norman and V. Shmatikov. Analysis of probabilistic contract signing. *Journal of Computer Security*, 14(6):561–589, 2006. 73

[Par02] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002. 8, 21, 22, 49, 50

[Tea13] Matías Tealdi. Paralelización de algoritmos en gpu para verificación simbólica de modelos probabilísticos. 2013. 86

[WB12] A. Wijs and D. Bosnacki. Improving gpu sparse matrix-vector multiplication for probabilistic model checking. *19th International SPIN Workshop on Model Checking of Software (SPIN'12)*, 2012. 86