Sara Tavares Martins

Licenciada em Ciências da Engenharia Eletrotécnica e de Computadores

Computação paralela utilizando GPU na análise de redes de Petri IOPT

Dissertação para obtenção do Grau de Mestre em Engenharia Eletrotécnica e de Computadores

Orientador: Luís Filipe dos Santos Gomes, Prof. Doutor, FCT/UNL

Júris:

Presidente: Doutor João Francisco Alves Martins – FCT/UNL

Vogais: Doutor Luís Filipe dos Santos Gomes – FCT/UNL

Doutor Filipe de Carvalho Moutinho – FCT/UNL

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2019

Computação paralela utilizando GPU na análise de redes de Petri IOPT

“Não to mandei eu? Esforça-te e tem bom ânimo; não te atemorizes, nem te espantes, porque o Senhor teu Deus está contigo, por onde quer que andares”

Josué 1:9

Agradecimentos

Quero agradecer ao professor Luís Gomes, pelo apoio e orientação que me deu ao longo deste trabalho, sem a qual não teria sido possível concluir.
Um agradecimento a Carolina Lagartinho Oliveira, pelo total apoio que me deste durante o desenvolvimento da aplicação, e pela paciência e dedicação que tiveste comigo durante todo trabalho.
Quero agradecer a professora Anikó Costa pelo tempo disponibilizado durante as férias, para certificar que o desktop estaria sempre a trabalhar para que eu pudesse ter acesso remoto para dar a continuação ao trabalho durante o mês de agosto.
À professora Teresa Cavão e os demais que fazem parte da FACIT, pelo apoio que me tem durante todo o meu percurso académico.
Aos professores de mestrado pela dedicação e profissionalismo que mostraram ao longo deste ano.
Quero agradecer a todos os meus amigos da faculdade, pois sem eles não seria possível completar esta etapa, e pelos bons tempos que me proporcionaram, por todas as amizades sinceras que tive ao logo desses anos.
Um agradecimento a toda minha família, porque sem eles não seria possível a realização desta tarefa, pela paciência que tiveram comigo, e por serem compreensíveis por várias vezes que não pude estar presente por causa de um trabalho ou teste.
Um agradecimento especial para os meus pais, pela motivação e dedicação que me deram para nunca desistir.
E por fim quero agradecer a todos que fizeram parte da minha vida ao longo destes anos.

Resumo

O principal objetivo desta dissertação é melhorar o tempo de execução na construção do espaço de estados associado a um modelo de rede de Petri *Input-Output Place-Transition* (IOPT), utilizando computação paralela numa *Graphics Processing Unit* (GPU) instalada no computador com um servidor de IOPT-Tools em execução, permitindo o processamento descrito.

Os modelos de sistema de controlo desenvolvidos em Rede de Petri (RdP) podem ser muito complexos, o que pode tornar de difícil compreensão o seu comportamento. Devido à variedade e à dimensão das redes, os sistemas desenvolvidos em RdP podem apresentar um grafo associado de espaço de estados com muitos nós e arcos, tornando-se um problema sobre o ponto de vista computacional quando se pretende realizar a verificação das propriedades do modelo. Isto porque, na construção do grafo do espaço de estados pode ocorrer uma explosão do número de estados, ou seja, o grafo pode ser tão grande que dificulta a procura e análise de todos os estados que o modelo pode alcançar. Com a utilização da GPU pode-se contribuir para mitigar este problema, aumentando o desempenho no processamento da construção do espaço de estados.

O algoritmo implementado para o processamento da construção do espaço de estados utilizando GPU é adaptação do código gerado automaticamente pela plataforma IOTP-Tools. Para executar o algoritmo é usada a *Compute Unified Device Architecture* (CUDA) da NVidia.

A CUDA permite executar o algoritmo em Central Processing Unit (CPU) e *Graphics Processing Unit* (GPU). A parte sequencial do algoritmo é executada na CPU e a parte do processamento intensivo, ou seja, o tratamento dos estados não processados é executada na GPU.

Palavra Chave: GPU, GPGPU, CUDA, Redes de Petri, Rede IOPT, IOPT-Tools, Grafo do Espaço de Estados.

Abstract

The main goal of this dissertation is to improve the execution time in the construction of state-space associated with an Input-Output Place Transition (IOPT) Petri net models, using parallel computing in a GPU (Graphics Processing Units) installed on the computer with an IOPT-Tools server running, allowing the processing described.
The control systems models developed in Petri net can be very complex, which can make it difficult to understand their behavior. Due to the variety and size of the nets the systems developed in Petri Net may present an associated state-space graph with many nodes and arcs, becoming a problem from the computational point of view when one wants to check model properties. This is because, in the construction of state-space graph an explosion in the number of states may occur, that is, the graph can be so great that it makes it difficult to search and analyze all states that the models can achieve. Using the GPU can contribute to mitigate this problem, increasing the performance in processing state-space construction.
The algorithm implemented for processing the state-space construction using GPU is an adaptation of the IOPT-Tools platform automatically generated code. To run the algorithm, Nvidia´s Compute Unified Device Architecture (CUDA) is used.
CUDA allows one to run the algorithm on Central Processing Units (CPU) and Graphics Processing Units (GPU). The sequential part of algorithm runs on the CPU and intensive processing parts, that is, the treatments of the unprocessed states is performed on the GPU.

Keywords: GPU, GPGPU, CUDA, Petri net, IOPT Nets, IOPT-Tools, State-Space Graph.

Conteúdo

Índice de figuras

Índice de tabelas

Índice dos gráficos

Lista de abreviaturas

CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random- Access Memory
GALS	Globalmente Assíncrono Localmente Síncrono
GDDR	Graphics Double Data Rate
GPC	GPU Processing Clusters
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
IOPT	Input Output Place Transition
PNML	Petri Net Markup Language
RdP	Redes de Petri
SDRAM	Synchronous DRAM
SFU	Special Function Unit
SM	Streaming Multiprocessor
SP	Streaming Processor

XML	Extensible Markup Language
XSL	Extensible Stylesheet Language

Lista de símbolos

M_0	Marcação inicial
M_{conj}	Conjuntos de marcação finitos
∅	Conjuntos vazio
∈	Pertence A
∩	Interseção
∪	União
⊆	É igual A ou está contido
M’	Nova Marcação
t	Transição habilitada
ω	Conjunto infinitos de marcações

1

1 Introdução

Neste capítulo apresenta-se a importância da utilização da *Graphics Processing Unit* (GPU) na verificação dos modelos de sistemas com Redes de Petri (RdP), assim como os objetivos e a estrutura desta dissertação.

1.1 Motivação e enquadramento do trabalho

Em 1962, Carl Adam Petri, durante a sua dissertação de doutoramento, “Kommunikation mit Automaten” “(Communication with Automata)” na universidade de Bonn na Alemanha, criou uma ferramenta de modulação gráfica matemática, que recebeu o nome de Redes de Petri (RdP) em sua homenagem [1][2], e que se tornou cada vez mais utilizado na modelação de processos económicos, científicos, nas industrias entre outros sistemas operacionais [3].

As Redes de Petri (RdP) são conhecida como uma ferramenta que modela sistemas utilizando processos paralelos discretos. São utilizadas cada vez mais como linguagem de programação para construir projetos de diversas áreas nomeadamente no ramo das ciências e engenharias [3], aplicando uma linguagem gráfica, sendo que uma delas são as das classes *IOPT (Input-Output Place-Transition)* para simulação de um sistema.

A tecnologia evoluiu bastante, e com o aumento de complexidade dos sistemas de controlo, tem se tornado cada vez mais difícil de modular e simular um sistema através da linguagem de RdP. Essa evolução fez a RdP enfrentar várias limitações, com isso

surgiram várias soluções, sendo uma delas a utilização da GPU, com o objetivo de acelerar a simulação ou verificação das propriedades do modelo.

A computação gráfica tem evoluído muito em GPU *multicore.* Com o passar do tempo aumentou a sua capacidade computacional, com isso facilita a sua rápida execução de sistemas que funcionam paralelamente.

Atualmente, várias aplicações ou ferramentas foram introduzidas para permitir o desenvolvimento e execução de *General Purpose Computations* em uma GPU (GPGPU), uma delas é a NVIDIA CUDA (*Compute Unified Device Architecture*) [4]. A arquitetura CUDA possui vários *processor cores* responsáveis pelo processamento de dados. No modelo de computação usa-se uma CPU e uma GPU juntas para o processamento de dados, em que a parte sequencial é executada na CPU e a parte do uso intensivo de computação é acelerada pela GPU [5].

1.2 Objetivos

O principal objetivo desta dissertação é melhorar o tempo da execução na construção do espaço de estados associados a um modelo de rede de Petri IOPT utilizando computação paralela numa GPU instalada no computador com um Servidor das IOPT-Tools em execução, permitindo o processamento descrito. Com um modelo de rede de Petri IOPT aplica se uma técnica para verificar o seu comportamento recorrendo à construção do espaço de estados, ou seja, a partir de uma marcação qualquer, irá ser construído um espaço de estados que corresponde aos vários estados do sistema. Este modelo é implementado com base nas ferramentas IOPT-Tools. Para isso, será necessário ter um processador, isto é, uma placa GPU que realizará esta tarefa, com apoio específico da hardware para alcançar o referido processamento.

1.3 Estrutura da dissertação

Esta dissertação encontra-se dividida em sete capítulos. No primeiro capítulo apresenta-se uma breve introdução sobre a realização deste trabalho bem como os seus principais objetivos.

No segundo capítulo são abordadas as definições geral e matemática de uma RdP assim como as propriedades da mesma. Também se apresentam classes de RdP com o foco principal na classe IOPT, sendo esta, utilizada neste trabalho. Ainda se

apresenta a plataforma utilizada para modelar os sistemas de controladores, na qual é utilizada a classe IOPT para a construção do espaço de estados da mesma. Por último, é introduzida a abordagem utilizando *General Purpose Graphics Processing Uni*t (GPGPU) com o propósito de melhorar o tempo da construção do espaço de estados. Aliada à GPGPU, é utilizada a plataforma *Compute Unified Device Architecture* (CUDA), com o intuito de executar a construção do espaço de estados utilizando a GPU.

No terceiro capítulo são apresentadas todas as ferramentas e recursos utilizados no decorrer deste trabalho. Neste sentido é abordada a geração de código para geração do espaço de estado através da plataforma *IOPT-Tools* em que se apresenta o algoritmo da construção do espaço de estados e a placa GPU – GeForce Titan V.

No quarto capítulo é feita a implementação do algoritmo supracitado na plataforma CUDA, descrevendo o processamento paralelo da construção do espaço de estados.

No quinto capítulo são apresentados os resultados dos modelos utilizados no algoritmo que foi apresentado no capítulo anterior. Também descreve detalhadamente os modelos e é feita uma análise sobre os resultados obtidos.

No sexto capítulo são apresentadas as principais conclusões e sugestões para trabalhos futuros.

No último capítulo são expostas todas as referências utilizadas para o desenvolvimento desta dissertação.

2

2 Revisão bibliográfica

Neste capítulo serão abordados alguns conceitos teóricos, necessários para uma melhor compreensão e preparação para o desenvolvimento do trabalho realizado na presente dissertação, bem como as tecnologias que irão ser utilizadas.

2.1 Redes de Petri - RdP

2.1.1 Definição

Uma RdP é representada através de um grafo bipartido definindo a orientação sobre um processo dinâmico. O grafo é constituído por elementos geométricos representando os nós, como círculos que representam condições e retângulos que representam eventos [6].

2.1.2 Componentes da Rede de Petri: Lugares, Transições e Arcos

Uma RdP é uma estrutura com dois tipos de nós, nomeadamente lugares e transições, como se encontra na Figura 2.1.a), e Figura 2.1.b) respetivamente. Um lugar graficamente é representado por círculos ou elipses, as transições são representadas graficamente por barras, quadrados ou retângulos. Lugares e transições são

conectados por arcos, representados graficamente por arcos dirigidos (terminados com uma seta) como ilustra a Figura 2.1.c).

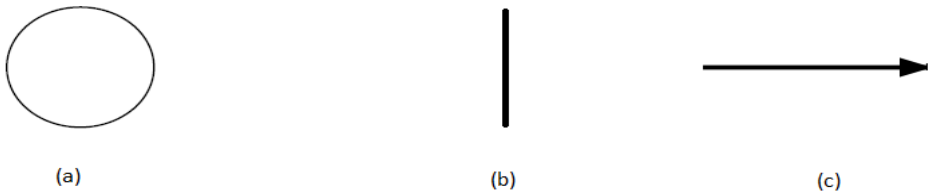


Figura 2.1 a) Um Lugar, b) uma Transição e c) um arco

Um lugar é um componente passivo, ou seja, ele pode armazenar ou acumular marcas. Contrariamente a um lugar, a transição é sempre um componente ativo, ou seja, pode ser utilizado para modelar situações como produzir, consumir, transportar e alterar. Relativamente ao arco, este é um componente que mostra uma relação abstrata entre lugares e transições [2].

Definição 1: (de [7],[8],[9] adaptado em [10]) *RdP é um estrutura* $N = (P, T, F, M_0)$ *onde P é o conjunto de* m *lugares, T é um conjunto de* n *transições, F:*$(P \times T) \cup (T \times P) \rightarrow \mathbb{N}$*, é a relação de fluxo ponderada* $M_0: P \rightarrow \mathbb{N}$ *é a marcação inicial. A matriz C:* $(T \times P) \rightarrow \mathbb{Z}$ *definida por* $C(p,t) = F(p,t) - F(p,t)$*, é a matriz incidência da rede.* $Pres: P \times T \rightarrow \mathbb{N}$, $Post: P \times T \rightarrow \mathbb{N}$ *são matrizes de pré e pós incidência que especificam os arcos direcionados de transição para lugares e é representado como matriz* $m \times n$.

2.1.3 Marcação

Como foi referido na definição 1, uma RdP é constituída por quatro elemento em que o quarto elemento é a marcação.

As marcas são associadas aos lugares e são representadas por pontos pretos ou inteiros positivos representando o seu número. A distribuição das marcas nos lugares de uma rede é denotada por marcação que corresponde a um estado do sistema. Sendo a sua distribuição inicial denominada por marcação inicial [2][11].

Cada lugar pode ser ou não marcado. Diz-se que um lugar é seguro quando contém apenas 0 ou 1 marca, e se todos os lugares forem seguros a rede será segura. E isso é uma das propriedades da RdP que mais adiante será aprofundada.

2.1.4 Disparo de transição

Como foi referido anteriormente, uma RdP é constituída por lugares, transições, arcos e marcações.

A marcação é criada e removida de lugares de acordo com o disparo de transições. Para isso é estabelecida uma condição de evento, que depende de uma função de sinal de entrada e de saída. A transição é disparada quando forem satisfeitas as pré-condições (todos os lugares de entradas estarem marcados) e as pós-condições (todos os lugares de saídas não estarem marcados) (Figura 2.2.a) e como consequências as marcas nos lugares de entrada são destruídas ou removidas e criadas marcas nos lugares de saída (Figura 2.2.b) [12]. Para as classes de redes de Petri mais comuns em engenharia, as pós-condições não são consideradas na habilitação de uma transição.

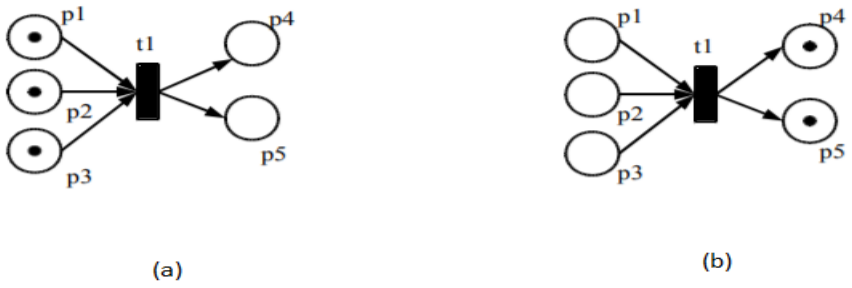


Figura 2.2 Rede de Petri com marcação inicial (a), Rede de Petri após o disparo de t1 (b)

Como mostra a Figura 2.2.a) temos três lugares p1, p2 e p3 e uma transição t1 que está habilitada. A Figura 2.2.b) mostra a situação após o disparo de t1; pode-se ver que foram consumidas três marcas na entrada de t1 e produzidas duas marcas na saída. Isto mostra uma situação de operação *atómica* [13].

2.1.5 Propriedades

Os modelos expressos em RdP deparam-se com várias dificuldades de análise devido a complexidades computacionais, que crescem rapidamente se a dimensão dos sistemas aumentar. Isso impede de ter uma visão mais clara sobre o comportamento do sistema. Sendo assim a análise estrutural de RdP pode ser obtida de várias maneiras dependendo da complexidade do problema [14].
A rede apresenta determinadas propriedades independentemente da marcação inicial, mas são caracterizadas de acordo com a estrutura da rede. Diz-se que uma rede é [14] [15] :

- **Conservativa,** quando o número das marcas que circulam na rede se mantém igual a marcação inicial, ou seja, a quantidade de marcas é invariante após o disparo de qualquer transição.
- **Alcançável ou Acessível,** quando a partir da marcação inicial, por uma sequência de disparo for possível chegar a essa marcação.
- **Reversível,** se a marcação do estado inicial for sempre alcançável.
- **Simplesmente limitada,** se o número das marcas em cada lugar não ultrapassar um número finito *k* para qualquer marcação alcançável.
- **Estruturalmente limitada,** se for limitada por qualquer marcação finita.
- ***Morta,*** quando todas as transições são mortas, ou seja, que nunca se disparam.
- **Viva,** quando todas a transições são vivas.

2.2 Classes de Redes de Petri Autónomas e Não-Autónomas

As redes de Petri dividem-se em dois grandes grupos: RdP Autónomas e RdP Não-Autónomas. As redes Autónomas incluem as componentes do modelo RdP, ou seja, possuem a mesma estrutura dos lugares, transições e arcos, enquanto que as redes Não-Autónomas são redes que se apresentam características que se adaptam ao modelo de sistemas de eventos discretos como a modelação de dependências temporais e sinais externos [16].

As classes de RdP autónomas são caracterizadas por níveis que são: primeiro, segundo e terceiro nível. Os dois primeiros níveis normalmente são designados por RdP de baixo-nível enquanto que o terceiro é designado por RdP de alto-nível. As RdP de baixo-nível são caracterizadas por lugares com zero, uma ou mais marcas sem estrutura associada, onde os lugares representam condições ou contadores, enquanto que as RdP de alto-nível são caracterizadas por lugares que podem conter marcas com estruturas associadas (cores) e tem como objetivo obter modelos mais compactos para sistemas mais complexos [16].

Existem vantagens de utilização de RdP, apresentadas nas seguintes linhas [17].

- Facilita a modelação e a compreensão do sistema usando uma notação gráfica e matemática.
- Simplifica a validação, análise e simulação do sistema de controlo durante a fase de implementação.
- Facilita a deteção de conflitos e situações inesperadas.

2.3 Redes de Petri IOPT (Input-Output Place Transition)

As redes de Petri IOPT derivam da classe de RdP Não-Autónoma conhecida como Lugar-Transição, que modela o controlador e o ambiente. A interação é feita através da ocorrência de eventos que condicionam o disparo das transições. A construção do modelo permite que o modelador especifique a sua conexão entre o controlador e o ambiente, sendo este visto como um conjunto de eventos e sinais ativos de entrada e saída. Os sinais e eventos de entrada definem o estado de entrada do sistema, que aciona os controladores em cada passo de execução. Esses sinais podem ter valores lógicos e valores inteiros positivos e os eventos podem ter valores booleanos [18][19].

As redes de Petri IOPT tem algumas características que permitem a especificação de um conjunto de prioridades nas transições, de forma a permitir a resolução de conflitos. As transições envolvidas numa situação de conflito serão disparadas em função de uma prioridade em relação às outras envolvidas no mesmo conflito, caso existam marcas suficientes para garantir os disparos. Os disparos de transições podem emitir eventos de saída, e também podem atualizar os sinais de saída. Os eventos de saída estão associados às transições [18].

As redes Petri IOPT também permitem a atribuição de pesos aos arcos.

Nos próximos parágrafos são apresentadas as definições mais detalhadamente sobre rede IOPT utilizada em [19].

Definição 2(Interface do sistema): A interface do sistema de controlador de uma rede IOPT é um tuplo $ICS = (IS, IE, OS, OE)$ satisfazendo os seguintes requisitos:

1) IS é um conjunto finito de sinais de entrada.
2) IE é um conjunto finito de eventos de entrada.
3) OS é um conjunto finito de sinais de saída.
4) OE é um conjunto finito de eventos de saída.
5) $IS \cap IE \cap OS \cap OE = \emptyset$

Definição 3 (Estado da entrada do Sistema): Dado uma interface *ICS = (IS, IE, OS, OE*) com um sistema controlador (*Definição 2*), um estado de entrada do sistema é dado pelo par *SIS = (ISB, IEB*) satisfazendo os seguintes requisitos:

1) *ISB* é um conjunto finito de sinal de entrada*:* $ISB \subseteq IS \times \mathbb{N}$*.*
2) *IEB* é um conjunto finito de evento de entrada*:* $IEB \subseteq IE \times \mathbb{B}$*.*

Definição 4 (Rede IOPT): Dado um controlador com uma interface *ICS = (IS, IE, OS, OE),* uma rede IOPT é um tuplo *N= (P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc),* satisfazendo os seguintes requisitos*:*

1) *P* é um conjunto finitos de lugares.
2) *T* é um conjunto finitos de transições (disjuntos do P).
3) *A* é um conjunto de arcos, de tal modo que $A \subseteq ((P \times T) \cup (T \times P))$*.*
4) *TA* é um conjunto de arcos teste, de tal modo que $TA \subseteq (P \times T)$*.*
5) *M* é a função marcação*:* $M: P \longrightarrow \mathbb{N}_0$*.*
6) *weight:* $A \longrightarrow \mathbb{N}_0$*.*
7) *weightTest:* $TA \longrightarrow \mathbb{N}_0$*.*
8) *priority* é uma função parcial de inteiros não negativos aplicada a transição: prioridade*:* $TA \longrightarrow \mathbb{N}_0$*.*
9) *isg* é um conjunto de expressões booleanas, associadas aos sinais de entrada, e que são aplicadas como condições de habilitação a transição (onde todas as variáveis são sinais de entradas)*:* $isg : T \longrightarrow BE$*, onde* $\forall eb \in isg(T), Var(eb) \subseteq IS$*.*

10) *ie* é um conjunto de eventos de entrada, gerados a partir de sinais de entrada, e que representam condições de habilitação para transições: $ie : T \longrightarrow IE$.
11) *oe* é um conjunto de eventos de saída, e com uma dependência de transição de forma a serem gerados $oe : T \longrightarrow OE$.
12) *osc* é um conjunto de sinal de saída, ativados em função de uma série de condições associadas à marcação dos lugares: $osc : P \longrightarrow P(RULES)$, onde $RULES \subseteq (BES \times OS \times \mathbb{N}_0), \subseteq BE \quad e \quad \forall e \in BES, Var(e) \subseteq ML \quad com \quad ML$ o conjuntos de identidades para cada marcação do lugar depois da execução de um passo: cada marcação de lugar tem um identificador associado, que é usado quando executado o código gerado.

Pode se ainda realçar que a transição depende das prioridades e dos eventos de entrada (*ie*) e os eventos de saída (*oe*) aplicando às condições dos sinais externos (*isg*), em que estes podem ser alterados com base nos disparos das transições (eventos de saída (*oe*)) ou no final de cada passo de execução, com base nas marcações. Sempre que uma transição é habilitada e a condição externa é verdadeira (*ie* e *isg*), a transição é disparada.
Um passo de execução é o período entre dois *tics,* que é definido por um relógio global externo.

2.4 IOPT-Tools

IOPT-Tools é ambiente que desenvolve sistema integrado utilizando ferramentas que suportam um desenvolvimento de sistemas embutidos, aplicativos de automação industrial e outros sistemas, incorporando o desenvolvimento e testes de controladores. Os sistemas são especificados através da classe de redes IOPT, que é uma classe alargada e fornece recursos de entrada e de saída, necessários para comunicar com o mundo externo, permitindo a leitura de sensores, manipulação de atuadores. Também eles comunicam-se com outros sistemas e implementam interface com o utilizador. O editor das redes IOPT permite a utilização das várias transições, lugares, arcos, sinais e eventos de entrada e de saída, assim como outras características e propriedades como deteção de erros, geração de código entre outros como será visto em seguida, baseando na referência principal [20].

2.4.1 Lugar, Transição e Arco

Como referido anteriormente, os lugares, as transições e os arcos são representados nas IOPT-Tools por modelos geométricos, como se representa na Figura 2.3.

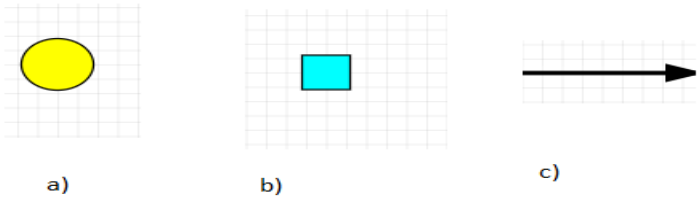


Figura 2.3 Lugar (a), Transição (b) e Arco (c)

Esses tipos de componentes também apresentam diversas propriedades em IOPT-Tools, que serão apresentadas em seguida.

2.4.1.1 Lugares

Além do identificador do lugar que é único, existem campos que contém o nome, uma marcação inicial e um número máximo de marca atribuído. Existe um limite de marcas nos lugares durante a execução da rede, ou seja, quando se dá a execução, o lugar terá uma marcação inicial que não pode ultrapassar o número máximo das marcas. Esses valores são apenas uma indicação para os geradores do código alocarem os elementos da memória para armazenar marcas, e podem ser calculado usando as ferramentas de cálculo do espaço de estados. O domínio temporal para RdP de sistemas Globalmente Assíncrono, Localmente Síncrono (GALS) é um campo que permite adicionar comentários. Finalmente o campo de ação de saída pode ser utilizado para definir os estados dos sinais de saída. Uma ação é definida por um nome de sinal, uma expressão de valor e uma condição. Uma expressão de saída só é avaliada quando um lugar é marcado e se a condição for verdadeira caso contrário, o valor de sinal de saída será um valor por *default* (Figura *2.4*).

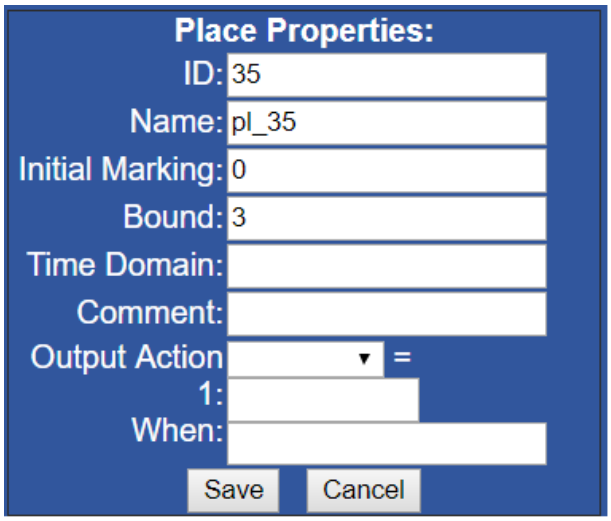


Figura 2.4 Propriedades dos Lugares

2.4.1.2 Transição

Da mesma maneira que os Lugares, as transições também têm o seu próprio identificador, o seu próprio nome, domínio temporal e campos para adicionar comentários. Para além disso, a transição tem uma prioridade, que é usada para resolver os possíveis conflitos em que várias transições disputam uma mesma marca, onde o menor número corresponde a maior prioridade. O disparo da transição pode ser inibido com condições de guarda e eventos de sinais de entrada, em que as condições são expressões lógica, que podem ser sinais de entrada e de saída. Os eventos e ações de saída podem ser associados às transições, ou seja, quando uma transição é disparada os eventos de saída são disparados (Figura 2.5).

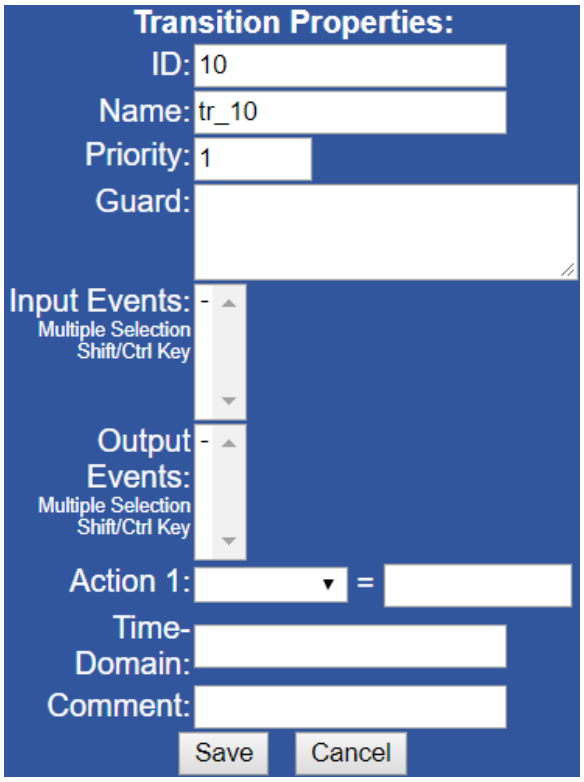


Figura 2.5 Propriedade das Transições

2.4.1.3 Arcos

Da mesma maneira que os lugares e as transições, os arcos também contém um identificador único. Para além disso existem mais dois tipos de propriedades, tipo de arco e a inscrição. Existem dois tipos de arcos, o normal e o teste ou canal. Relativamente a inscrição, o valor indica o número das marcas: no caso de arcos de entrada, o número da marca é necessário para disparar uma transição e para o caso do arco de saída o número da marca é adicionada no lugar de saída. O arco de teste só pode ser utilizado como arco de entrada de transição (inicia em um lugar e termina em uma transição), em que a(s) marca(s) presente(s) no lugar de entrada não será(ão) consumida(s) aquando do disparo da transição (Figura 2.6).

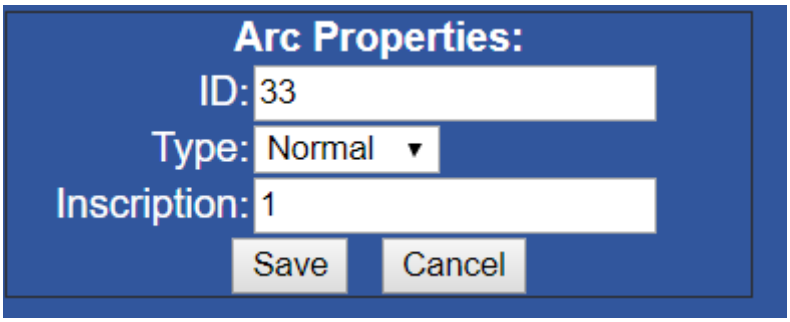


Figura 2.6 Propriedade dos Arcos

2.4.1.4 Sinais

O sinal também tem as suas propriedades, que é o nome com um modo de entrada ou saída, com um tipo, com um valor *default* e com um intervalo mínimo/máximo associado. O nome do sinal é o identificador usado internamente nos modelos IOPT para definir expressões matemáticas, também são usados como uma ferramenta para gerar código. O tipo de sinal pode ser representado por booleanos ou por números inteiros. O modo do sinal é definido quando o sinal é criado e o valor *default* se aplica somente aos sinais de saída (Figura 2.7).

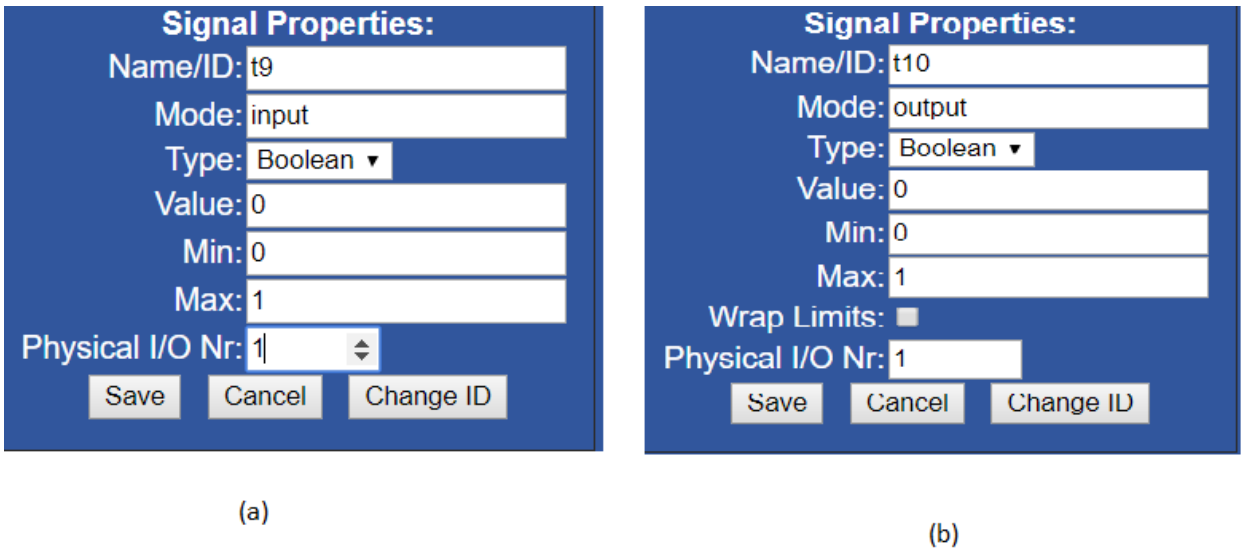


Figura 2.7 Propriedades de sinal: (a) sinal de entrada, (b) sinal de saída

2.4.1.5 Eventos

Assim como sinais, os eventos também têm um identificador único e dois modos e possuem as mesmas propriedades que os sinais, ou seja, são definidos quando os eventos são criados. Os eventos de entrada e de saída podem ser autónomos ou associados a um sinal. Os eventos autónomos são usados para implementação compostas por múltiplos modelos, em que um evento de saída de um componente é diretamente associada a uma outra componente como eventos de entrada, garantindo a propagação de eventos na mesma etapa de execução (Figura 2.8).

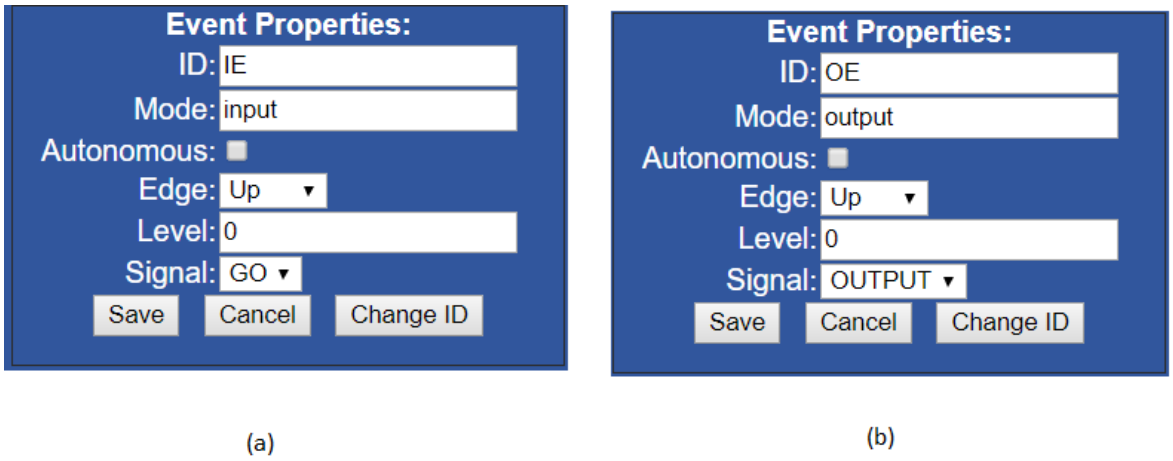


Figura 2.8 Propriedades de Eventos, (a) eventos de entrada, (b) eventos de saída

2.4.1.6 Editor de expressões

Finalmente, existe também um editor de expressões, que serve para construção das *guards.* O editor serve para criar e editar expressões, utilizando operadores, operações aritméticas, comparadores, lógicos e sub-expressões (*open* e *close* de *sub-expressão*, *NOT*) (Figura 2.9).

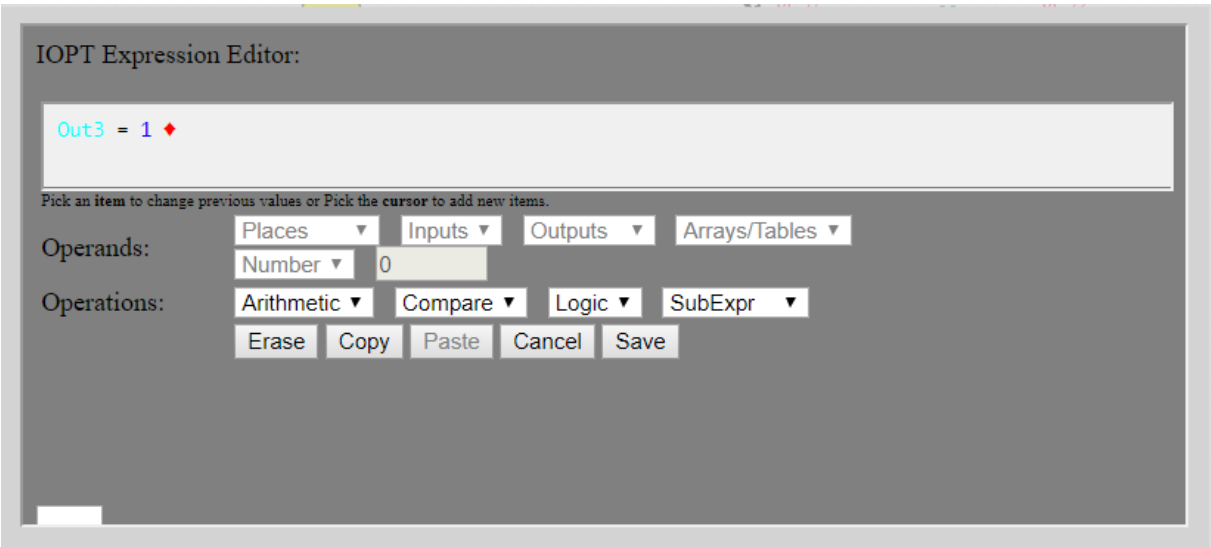


Figura 2.9: Editor de Expressão IOPT-Tools

2.5 Construção do espaço de estados de uma RdP

Existe várias técnicas possíveis para a construção da árvore, em que duas delas é mais usada: árvore de acessibilidade (ou alcançabilidade) e árvore de ocorrências. Nesta dissertação será abordada somente a técnica de árvore de acessibilidade, apresentada nos próximos parágrafos.

Dada uma marcação inicial, a arvore será contruída com base nas sequências de disparo de transições, gerando novas marcações para as transições ativadas. Seguindo o mesmo raciocínio do princípio, a partir de cada nova marcação geram mais marcações até que é encontrado uma marcação igual a que inicialmente tinha ou quando, nenhuma transição esteja habilitada. Uma marcação representa um nó na árvore e cada arco representa um disparo de uma transição que gera novas marcações. Caso a árvore seja infinita (presença de lugares não limitados) é introduzida um símbolo especial ω que pode ser visto como um conjunto infinito de marcações, sendo assim possível construir uma árvore reduzida (Figura 2.10) [21].

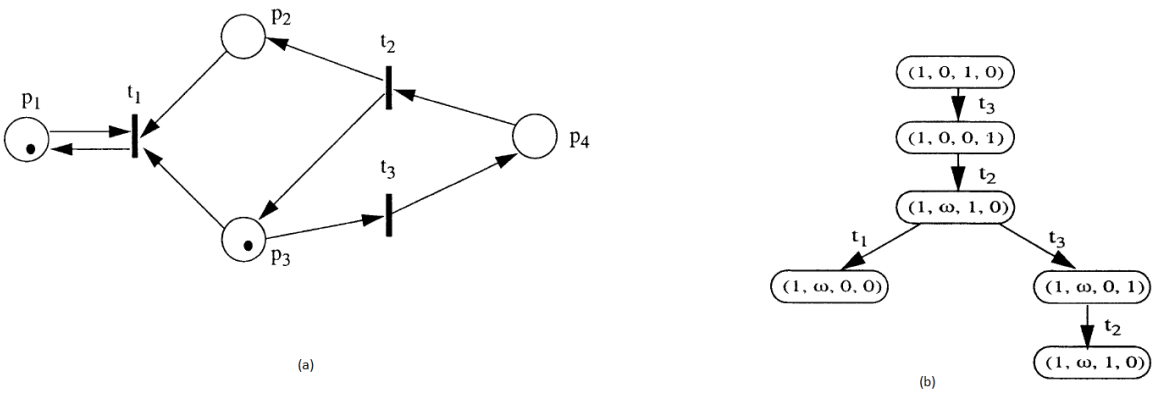


Figura 2.10: (a) Rede Petri; (b) árvore de acessibilidade [21].

Na árvore de acessibilidade, o nó com uma entrada ω representa um conjunto de marcações infinitos quando são considerados todos os valores possíveis de ω.

A referência [21] apresenta um algoritmo de como é construído a árvore de acessibilidade que é apresentado seguidamente.

(1) A marcação inicial M_0;

cria (M_0);

$M_{conj} = \emptyset$; (M_{conj} é um conjunto de marcações finito)

(2) Quando deixa de ser o nó da marcação atual M

repetir enquanto há transição {

se ($M_{conj} = \emptyset$)

então {cria (nova marcação M')

nó(M) →t →nó (M')

M_{conj} = {M'}

}

se não {*repetir* para cada $M_k \in M_{conj}$

se (M' = M_k)

então nó(M) →t →nó (M_k)

se (M' = M_0)

então nó(M) →t →nó (M_0)

}

} repetir até não ter mais transição habilitada.

A ideia básica do algoritmo é, com uma marcação inicial M_0, com uma marcação atual M e com uma transição t disparada sobre M. Construi-se a árvore em termos da marcação resultante M'. Se M' é igual a uma certa marcação na árvore de acessibilidade então desenha se um arco de M para a marcação existente. Se M' é diferente de qualquer outra marcação existente, então M' é uma nova marcação e um arco de M para M' é introduzido na árvore. Quando não há mais nenhuma transição habilitada o procedimento é encerrado.

2.5.1 Geração de espaço de estados da rede IOPT

A rede IOPT tem um papel importante na implementação de sistemas de controladores hardware e de software. Para criar esses controladores, deve-se conhecer a informação sobre um determinado conjunto de propriedades do sistema, que é obtida através do espaço de estados [22]. A classe de rede IOPT utiliza a semântica de execução *Maximal-Step* para a geração do espaço de estados [22][23]. Para o processamento de modelos de redes de IOPT, as ferramentas de geração do espaço de estados utiliza a estratégia de geração automática de código. Oferece também uma maior velocidade de geração do espaço de estados, melhorando a capacidade de analisar sistemas mais complexos [22].

A semântica de execução de *Maximal-Step*, significa que todas as transições habilitadas em um instante específico são disparadas numa próxima etapa de evolução. Com isso, garante-se uma operação coerente e determinística, um recurso responsável para geração automática de código de software e síntese de controladores em hardware. A operação determinística também requer a resolução de conflitos entre transições, atingindo-as com atribuição das prioridades das transições e arcos de teste [22].

O estado da rede IOPT é um vetor que inclui a marcação e todos os valores de sinais de saída influenciados pelos eventos de saída, isto porque o sistema memoriza os valores desses sinais [22].

2.5.1.1 Grafo do espaço de estados

O grafo do espaço de estados é composto por um conjunto de nós representando todos os estados da rede IOPT alcançáveis e um conjunto de arcos conectando cada nó aos respetivos filhos. Os nós filhos são todos os nós descendentes, originados de um único nó pai através do disparo de uma ou mais transições durante um passo de execução. Nós e arcos, contém anotações, isto é, representam vetores de estados e lista de transições disparadas de nó pai para originar cada nó filho [22].
Nas próximas alíneas são apresentadas definições mais detalhada do espaço de estados de uma rede de IOPT, baseadas em [22].

Definição 5: POS e OES são respetivamente *place output signal* e *output-event signals,* tal como:

1) POS é um subconjunto finitos de sinais de saída OS (definição 2)
2) OES é um subconjunto finitos de sinais de saída OS
3) $POS \cap OES = \emptyset$

Definição 6: Um vetor de estados da rede IOPT é um tuplo $SSV = (M, OESv)$ onde:

1) M é um vector de marcação de lugar (M da definição 4).
2) $OESv$ é um vetor de valores OES

Definição 7: TS é um conjunto de transições tal com $TS \subseteq T$ (definição 4).

Definição 8: dado uma rede IOPT (definição 4), o grafo do espaço de estados de uma rede IOPT é definido por um tuplo $SS = (sN, sA, nsv, at)$ satisfazendo os seguintes requisitos:

1) sN é um conjunto finitos de nós chamados estados.
2) sA é um conjunto finitos de arcos conectado nós, que satisfaz a seguinte condição $sA \subseteq sN \times sN$.
3) nsv é um estado de sistema de função parcial aplicando anotações de vector de estados para nós $nsv: nN \rightarrow SSV$.
4) at é um conjunto de função parcial de transição aplicando anotações de conjuntos de transições para arcos $at: A \rightarrow TS$.

Na Figura 2.11 encontra-se representado um modelo de rede Petri não-autónoma, com três sinais de entrada IS1, IS2 e IS3 e todas as transições possuem expressões

de *guard* que condicionam o disparo de transição de acordo com o estado do respetivo sinal de entrada.

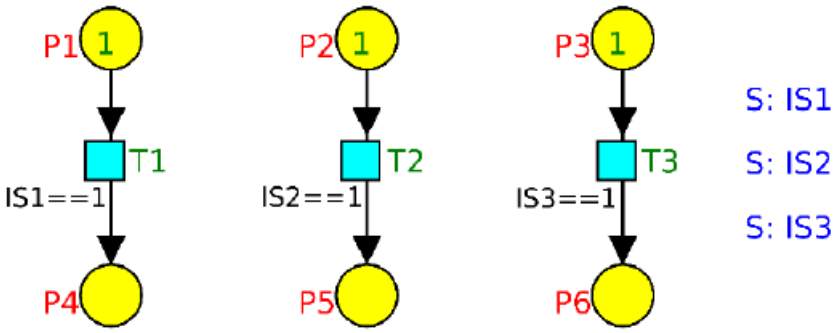


Figura 2.11: Modelo de rede IOPT [22].

O grafo do espaço de estados representado na Figura 2.12 do modelo de rede de IOPT (Figura 2.11) foi criado utilizando a semântica de *Maximal-Step*, o que cria uma complexidade do espaço de estados devido a aumento do número de arcos.

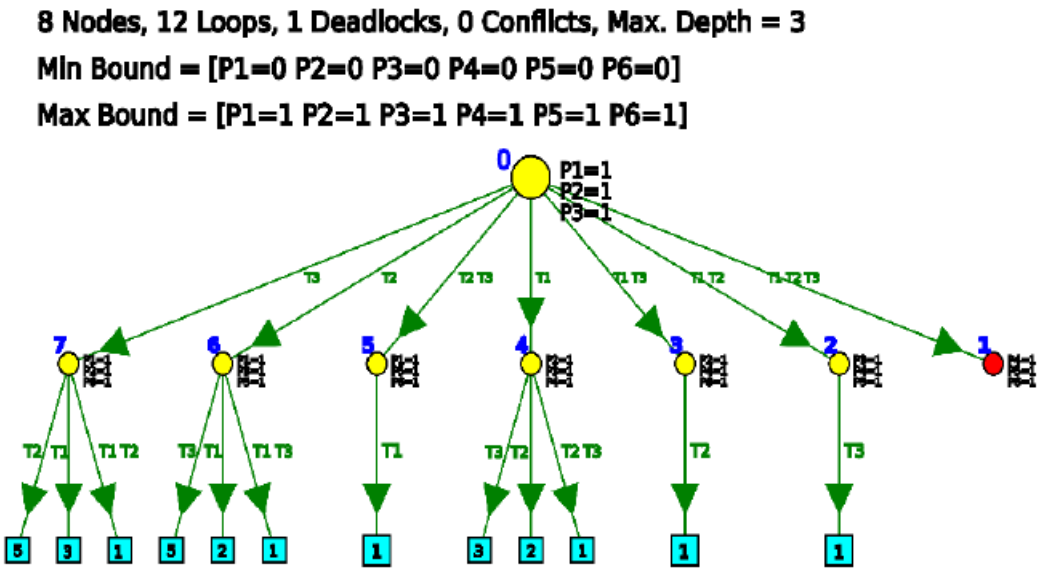


Figura 2.12: Gráfico do espaço de estados de rede IOPT [22].

A semântica *maximal-step* ainda é derivado das seguintes regras [22]:

Teorema 1: Dado um nó do espaço de estados, onde um subconjunto de N transições habilitadas TS sem nenhum conflito, o número C de nós filhos será dado por $C = 2^N - 1$.

Corolário 1: Dado um nó do espaço de estados, onde o subconjunto de N transições habilitadas TS são habilitados, se o número de nós filho C for menor que $2^N - 1$ *então o subconjunto de transição TS tem conflitos.*

2.6 General Purpose Graphics Processing Unit (GPGPU)

Nesta secção é apresentado um breve resumo sobre a hardware que vai ser utilizado nesta dissertação nomeadamente a *Graphics Processing Unit* (GPU), explicitando a sua evolução, tecnológica assim como as suas possíveis arquiteturas, constituições e funcionamento.

2.6.1 Graphics Processing Unit (GPU)

Atualmente o mercado de computação gráfica tem tido um crescimento significativo [24], evoluindo desde um Central Processing Unit (CPU) com um *core* (núcleo) para *CPUs multicore* capazes de realizar vários cálculos ao mesmo tempo [3], ou seja, processadores *manycore*, *multithread* com enorme potência computacional e com uma elevada largura de banda de memória [24], até GPU, tendo este ultimo na sua natureza paralela facilidade de execução rápida de cálculos de dados paralelos, ultrapassando muitos CPUs [4]. Esta evolução pode ser vista no gráfico da Figura 2.13.

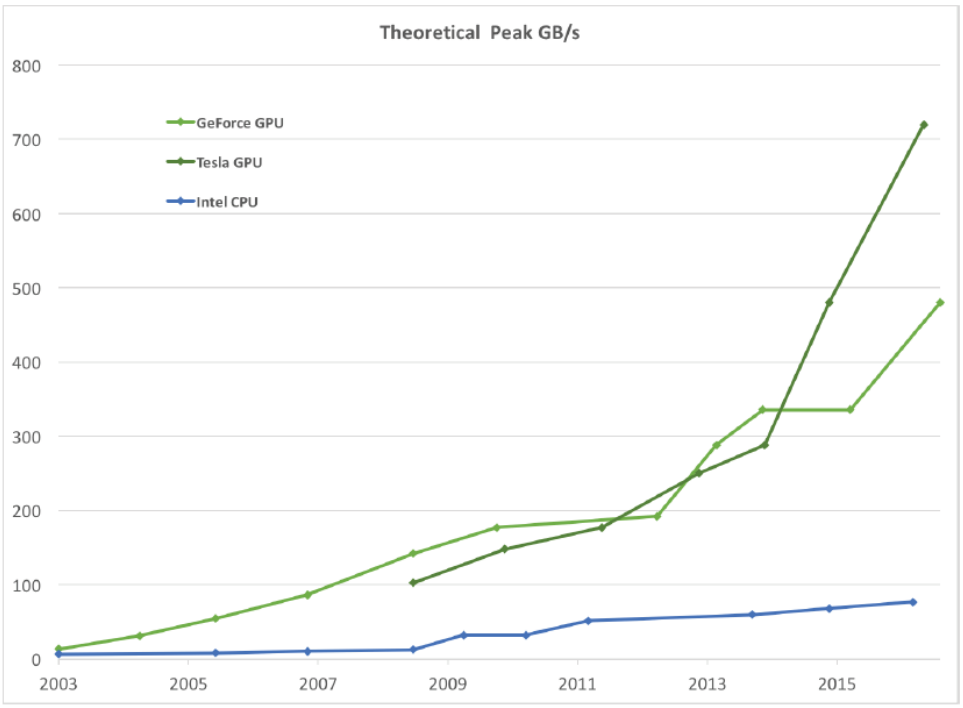


Figura 2.13 Largura de Banda de memória para CPU e GPU [24].

A razão por detrás da discrepância entre a capacidade de GPU e CPU é que GPU é projetada de forma a ter mais transístores dedicados para o processamento, em vez de armazenamentos de dados em cache e controlo de fluxo, isto é, ela é especializada em computação intensiva, altamente paralela, exatamente para o que é a renderização gráfica [3][24][25].

A Figura *2.14* mostra a diferença entra CPU e GPU e os seus componentes internos.

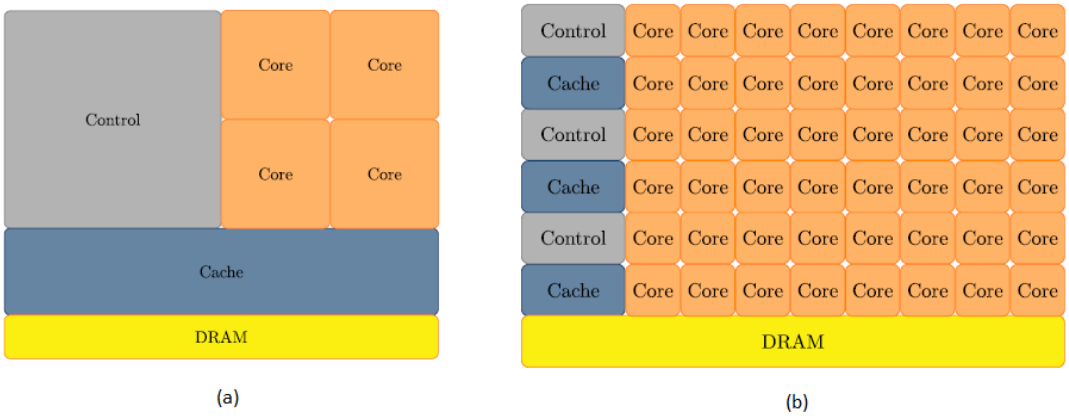


Figura 2.14*: Esquemático típica: (a) CPU, (b) GPU [3].*

Na arquitetura da GPU os núcleos de processamento são organizados em linhas com pequenas quantidades de caches e de hardware de controlo. As GPUs modernas suportam mais operações vetoriais e matriciais, transmitem rapidamente os dados e possuem muitas memórias locais, sendo utilizados como placas de vídeos em computadores modernos [26].

Segundo a referência [3], a GPU não é um componente obrigatório para a constituição do computador, mas sim a sua função era criar e processar apenas os gráficos da criação de filmes animados com vastas matrizes e ajudar na aceleração da produção. Elas são equipamentos computacionais de alta potência e enorme largura de banda que tem traduzido várias acelerações para grande variedade de aplicações [27], também atuam como um co-processador, baseado em matrizes de multiprocessadores de *Streaming Multiprocessors* (*SMs*) que possuem um design diferente dos núcleos de CPU, suportam o paralelismo a nível de instrução e são constituído por pequenos cache. Cada SM consiste em vários *Streaming Processor* (*SPs*), responsáveis pelas operações aritméticas e operações matemáticas, um ou mais programadores *warp* e um descodificador independente, para executar diferentes instruções [4].

A principal funcionalidade da GPU é resolver problemas como cálculos de dados paralelos, executando-as em altas intensidades aritméticas [24].

As GPUs têm ainda o uso de uma nova geração GPGPU (*General Purpose Graphics Processing Units*) que visam o processo mais geral, completo e intensivo, fornecendo um conjunto de operações que trabalham em dados arbitrários. Um GPGPU contém vários SMs que podem executar várias *threads* ao mesmo temp SMs estão implementadas com cache e unidades de controle que são compartilhados por *threads* internos [27].

2.6.2 Compute Unified Device Architecture (CUDA)

Muitos dos softwares que processam e executam grande quantidade de dados em longos períodos de tempo nos computadores, são projetados para implementar fenómenos físicos do mundo real. Este tipo de software tem capacidade de processar imagens e vídeos instantâneos de um mundo físico, em que diferentes partes de uma imagem capturam eventos físicos simultâneos e independentes, baseados em

paralelismos de dados. Esta execução de paralelismo de dados é uma das propriedades do software, por meio da qual muitas operações aritméticas podem ser executadas, no qual é utilizado um dispositivo CUDA para acelerar a sua execução [28].

CUDA é uma plataforma de computação paralela e uma API (*Application Programming Interface*) que permite que um dispositivo GPU seja utilizada para programar dados em paralelos. Também pode ser usada para resolver muitos problemas computacionais complexos de maneira mais eficiente do que uma CPU [3][24]. Ela é projetada pela NVidia e vem com um ambiente de software que permite aos programadores usar o C como linguagem de programação (também inclui as seguintes linguagens de programação como: *C++, Fortan, Java, Python, Warppers, DirectCompute, OpenCL, OpenACC, OpenMP*, encontra se mais informação em [28]), que é fácil de usar e acessível para todos aqueles que codificam a referida linguagem [24].

A CUDA permite que os programadores reservam memórias de GPU e executem *Kernel* e *threads* em paralelos [24][27].

A Figura 2.15 mostra a arquitetura de uma GPU compatível com a CUDA. É organizado em uma matriz altamente encadeada de *Streaming Multiprocessor* (SM)., em que dois ou mais SMs formam um *block*.

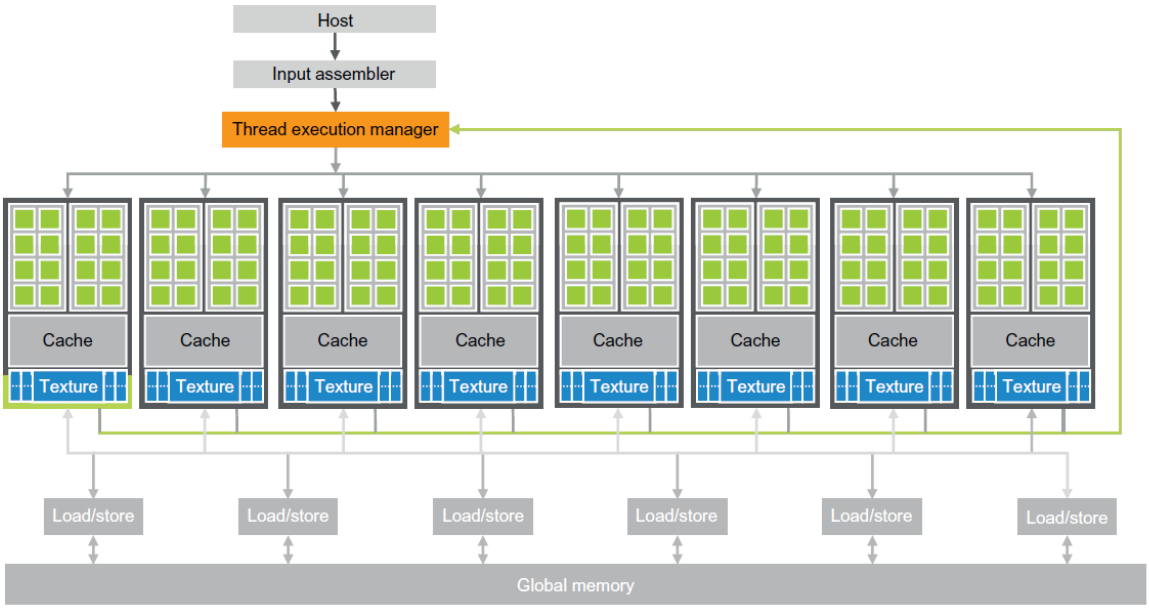


Figura 2.15: Arquitetura de uma GPU compatível com a CUDA [28].

A CUDA é constituída por uma memória global, uma memória constante e uma memória de textura estando todas incluídas na memória física, enquanto que a memória partilhada se encontra dentro de SMs. A memória local e de registo são utilizadas apenas para as *threads* [27]. As *threads* podem aceder dados de vários espaços de memórias durante a sua execução, como ilustra a Figura 2.16. Cada *thread* possui sua própria memória local. Cada *thread block* possui uma memória partilhada que é visível para todas as *threads blocks* com o mesmo tempo útil, em que todas as *threads* tem acesso à memoria global [24].

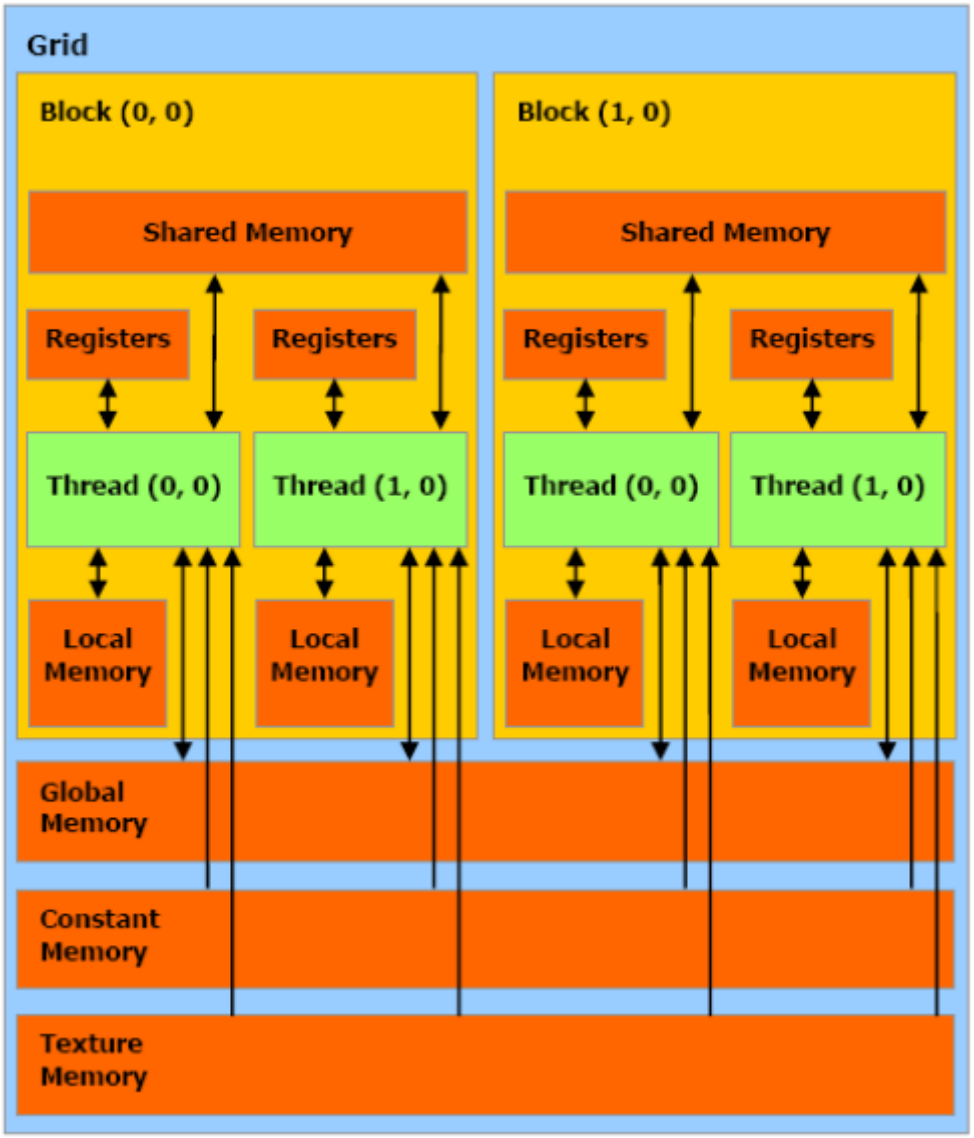


Figura 2.16: Hierarquia de memória CUDA [5].

A CUDA possui uma estrutura hierárquicas das *threads* (Figura 2.17) que reflete a arquitetura da hardware. Cada *Kernel* projetado é manipulado por um grande *thread grid* que consiste em um vetor ou matriz de *threads blocks*, onde os *blocks* contém uma matriz *thread* alocada [24].

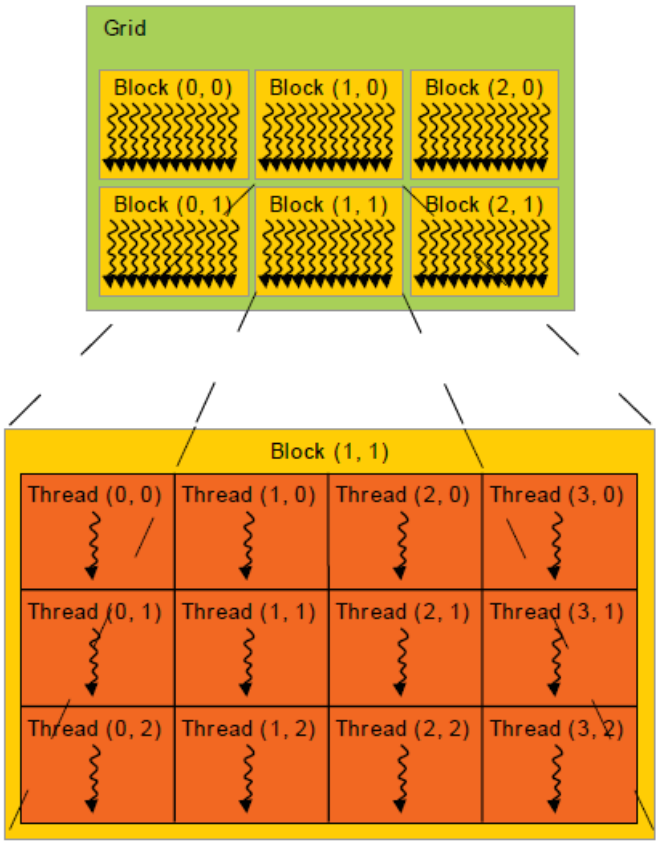


Figura 2.17: Grid de Threads blocks [24].

O limite máximo para o número de threads por *blocks* é de 1024, uma vez que todas as threads de um *blocks* consistem no mesmo *processor core* e devem partilhar os recursos de memórias limitados desse *core*. Um *kernel* pode ser executado por vários *blocks* de threads, de modo que número total das *threads* sejam iguais ao número de *threads* por *blocks* multiplicado pelo número de *blocks*. Os *blocks* são organizados em uma *thread grid* unidimensional, bidimensional ou tridimensional [24].

2.6.2.1 Modelo de programação CUDA

O modelo de programação de CUDA assume que uma *thread* é executada em um *device* separado fisicamente que opera como um processador para um *host* que executa o programa C [24], como mostra a Figura 2.18.

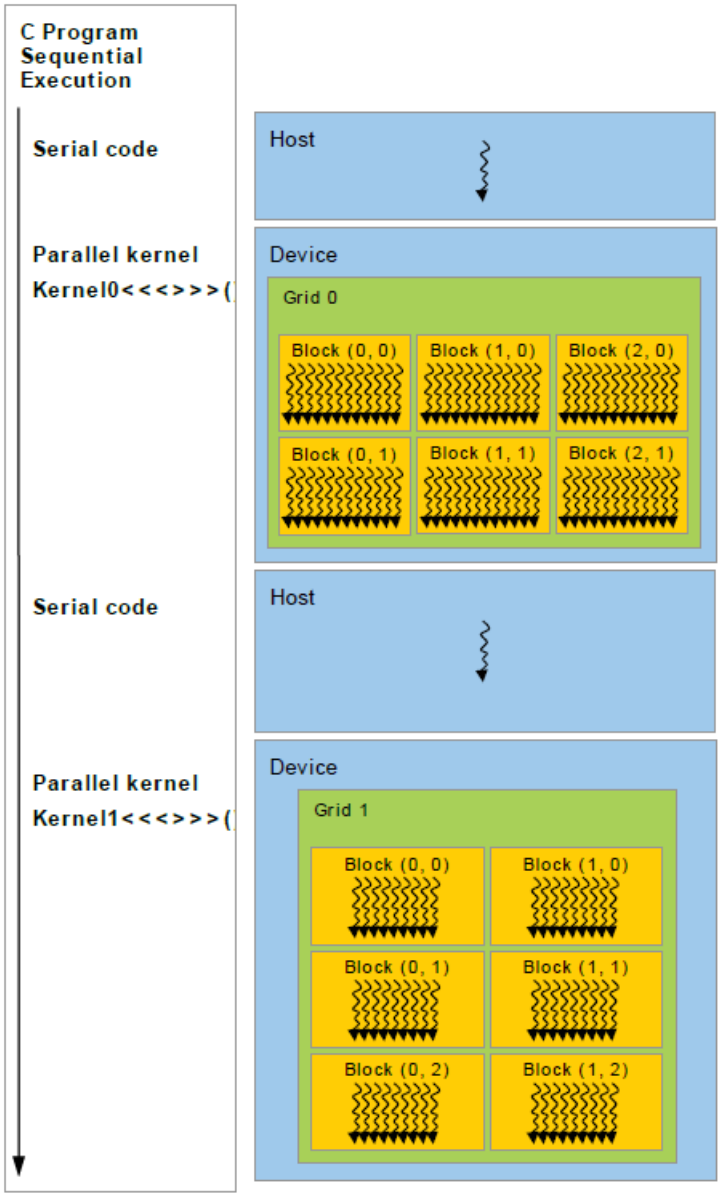


Figura 2.18: Estrutura do modelo de programação em CUDA [24].

O *host* e o *device* tem os seus próprios espaços de memórias separadas em *Dynamic Random Access Memory* (DRAM), conhecidos como os *host memory e device memory*, respetivamente. Um programa em CUDA faz a gestão dos espaços de memória global, constante e textura visíveis aos *kernels,* isso inclui alocação e

libertação da memória do *device*, bem como transferência de dados entre a memória *host* e a do *device*. No *host* são executados o *serial code* enquanto que no *device* é executado o *parallel code* [24].

3

3 Métodos para a construção do espaço de estados

Neste capítulo são abordados todos os procedimentos para a realização desta dissertação, nomeadamente, os recursos utilizados desde a geração automática de código C para a construção do espaço de estados no ambiente IOPT-Tools até a fase dedicada à sua construção através de uma GPU.

3.1 Ferramentas (Recursos)

3.1.1 Ficheiro PNML de uma Rede IOPT

Os modelos da rede de Petri IOPT gerados na plataforma IOPT-Tools são representados utilizando Petri Net Markup Language (PNML) [29]. Para geração do espaço de estados de uma rede, recorre-se ao modelo PNML, como ficheiros de leitura. Neste tipo de ficheiro encontra-se toda informação sobre a rede, incluindo um conjunto de marcações inicias, de sinais e de eventos de entrada e de saída e as expressões de condições de *guard* associadas ao disparo das transições.

De acordo com a informação encontrada em [30], o PNML foi projetado como um formato de representação de rede de Petri baseado em XML (Extensible Markup

Language), permitindo interoperabilidade entre ferramentas e plataformas que trabalham com diferentes formato de redes de Petri.

Como foi referido anteriormente, o ficheiro PNML de uma rede de Petri IOPT contém toda a informação da rede referente a entrada e saída de um modelo. Assim sendo, na Figura 3.1 se encontra representado um exemplo de um ficheiro PNML da plataforma IOPT-Tools.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
<Snoopy revision="0" version="0">
   <pnml>
     <net id="1" name="pla_tran" type="IOPT">
       <input>
          <signal id="GO" type="boolean" value="0" gpio_nr="0"></signal>
          <event id="Ev_in" edge="up" level="0" signal="GO"></event>
       </input>
      <output>
       <signal id="Out" type="boolean" value="0" gpio_nr="0" min="0" max="1" wrap="0"></signal>
       <event id="Ev_out" edge="up" level="0" signal="Out"></event>
     </output>
     <variable/>
     <place id="2">
       <name><text>p1</text></name>
       <initialMarking> <text>1</text></initialMarking>
       <bound> <text>3</text></bound>
       <signalOutputActions></signalOutputActions>
     </place>
     <transition id="3">
       <name><text>tr_3</text></name>
       <priority>1</priority>
       <signalInputGuards>
         <concreteSyntax language="iopt"><text>GO = 1</text>
          <expression><operand type="input-signal" idRef="GO" seq="1"/>
             <operation operator="equal" seq="2"><operand type="literal" value="1" seq="3"/></operation>
          </expression></concreteSyntax>
       </signalInputGuards>
       <inputEvents></inputEvents>
     </transition>
     <arc>
       <type>normal</type>
       <inscription><value>1</value></inscription>
     </arc>
   </pnml>
</Snoopy>
```

Figura 3.1: Exemplo de um ficheiro PNML utilizada na plataforma IOPT-Tools.

As redes de classe IOPT são conhecidas como rede de Petri Não-Autónoma Lugar-Transição. Como se pode verificar na Figura 3.1, o nó *Place* (Lugar) encontra-se associado a um *id name*, a um *initial marking* (marcação inicial), a um *bound* (que determina o limite das marcas num determinado lugar) e expressões de sinais de saídas associadas as ações de saída. O nó *Transition* (Transição) encontra-se associado a um *id name*, a uma prioridade, a uma condição de *guards* que define as expressões associadas aos sinais de entrada e de saída. Nos arcos encontra-se as informações associada a tipo de arcos e inscrição ou peso que indica o número das marcas.

Todas essas informações são usadas para a geração automática de código para a construção dos espaços de estados da rede Petri IOPT.

3.1.2 Linguagem de Programação C

Na década de 1970, Dennis Ritchie inventou a linguagem C, que era utilizado para o uso do sistema operacional UNIX. Esta linguagem é considerada uma das linguagens de baixo nível mais utilizadas, ou seja, a sua forma de compilação e execução está mais próxima da linguagem máquina, semelhante a linguagem Assembly. Ela permite a manipulação de bits, bytes e endereços de memória do dispositivo sendo também é muito utilizada para programação de *Hardware*. A linguagem C é uma linguagem estruturada, ou seja, os códigos são separados em dados e funções (ou sub-rotinas) que utilizam variáveis locais. Com o uso das variáveis locais é possível escrever funções de forma que os eventos que ocorrem dentro delas não causam efeitos inesperados noutras partes do programa. Os códigos comunicam por meio de parâmetros de entrada e de saída das funções.
O algoritmo da construção do espaço de estados gerados na plataforma IOPT-Tools foi implementado utilizando a linguagem de programação C. As funções C geradas foram adaptadas para serem executadas em GPU. Para a execução das funções é utilizada a CUDA. A CUDA é uma plataforma que permite usar a linguagem C utilizando a GPU.

3.1.3 A arquitetura da NVidia GeForce Titan V

No capítulo anterior desta dissertação, referem-se alguns conceitos sobre a arquitetura das GPUs com suporte CUDA. Uma GPU é organizada em vector de Streaming Multiprocessors (SMs) com várias *threads. C*ada SMs tem um número de *Streaming Processors* (SPs) que partilham a lógica de controle e a cache de instrução. Cada GPU [28] vem com gigabytes de *Graphics Double Data Rate* (GDDR), *Synchronous* DRAM (SDRAM) conhecida como memória global. O GDDR e SDRAMs diferem das DRAMs de sistema na placa mãe da CPU, por serem essencialmente a memória de buffer usadas para gráficos, como armazenar imagens de vídeos e informações de texturas de renderizaçao 3D.

Para a aceleração da construção do espaço de estados de uma rede IOPT, foi utilizada a placa GPU da NVidia GeForce Titan V. Segundo a NVida [31][32][33], GPU tem uma arquitetura parecida com a GPU Volta. A Titan V vem com um chip de 21,1 bilhões de transístores de uma área de matriz de 815 mm^2, ainda se encontra 5120 *CUDA cores* divido em seis GPU Processing *Clusters (GPC),* com uma frequência de 1200 MHz, cada GPC contém sete *Texture Processing Clusters* (TPCs) e 14 Streaming Multiprocessors (SMs). Cada SM é fragmentado em quatro blocos de processamento, e cada bloco consiste em dois *tensor* core como se encontra na Figura 3.2. Esta placa GPU vem com 12 GB de memória, com 850 MHz de frequência de memória, com uma interface de 3072 bits de velocidade de 1,7Gbps (Gigabits por segundos) HBM2. Com todas essas especificações, a largura de banda chega a 652,8 Gb/s.

Figura 3.2: Arquitetura Titan V [34].

3.2 Utilização da ferramenta da construção do espaço de estados do ambiente IOPT-Tools

No capítulo anterior foi referido que as redes IOPT implementam sistemas de controladores hardware e de software. Esses sistemas de controlo são implementados utilizando ferramentas disponíveis na plataforma Web IOPT-Tools [35]. As IOPT-Tools fornecem uma editora gráfica para descrever o formalismo de rede de Petri IOPT, um simulador de sistema, um gerador do espaço de estados e ferramentas que executam a geração automática de código C e VHDL para implementação do sistema especificado.

As IOPT-Tools disponibilizam programas de software para executar a construção do espaço de estados de redes IOPT a partir do modelo de sistemas implementados por rede de Petri IOPT. Os modelos implementados fornecem um conjunto de informações sobre rede como os sinais e evento de entrada declarada nas *guards* associadas as transições e conjuntos de marcações iniciais. Os eventos de saídas ocorrem quando as *guards* associadas às transições são verdadeiras e com isso são habilitadas as transições, os sinais de saídas são os resultados das expressões de saída associadas aos lugares. Com isso, o estado do sistema é composto por dois vetores: um de marcação; e outro de sinais de eventos de saída, que contém os valores memorizados de todos os sinais associados aos eventos de saída.

O servidor IOPT-Tools fornece um gerador automático de código do espaço de estados que são disponibilizado aos seus utilizadores. A geração automática do código foi implementada usando um conjunto de transformações XSL (*Extensible Stylesheet Language*) [36], que leem o ficheiro do modelo do PNML original e criam o código C. O código resultante contém funções para implementar as regras semânticas do modelo, um algoritmo da construção do espaço de estados, uma tabela de *hash* para armazenar a *database* do espaço de estados e o código ficheiro I/O (Input/Output) que armazena o grafo do espaço de estados resultante dentro de um ficheiro XML hierárquico [37]. A Figura 3.3 mostra a ideia por detrás da geração automática do espaço de estados.

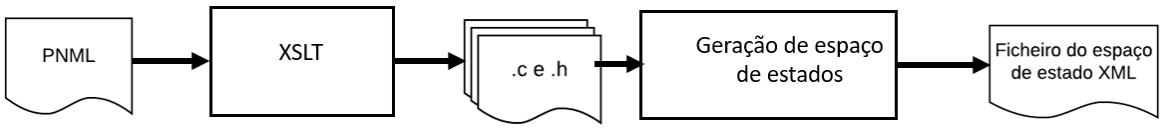


Figura 3.3: Geração do espaço de estados utilizando modelos IOPT.

O ficheiro C resultante que implementa o algoritmo da construção do espaço de estados são: *net_type.h*; *net_functions.c*; *ss_types.h*; *ss_exec_step.c*; *ss_hashtable.c; ss_compat.c* e *ss_main.c*, onde:

- *net_types.h* contém a lista dos lugares, dos sinais e eventos de entrada, dos sinais e eventos de saída e das transições.
- *net_functions.c,* contém as funções relacionadas aos lugares, as transições, e aos sinais e eventos de entrada e de saída.
- *ss_types.h* contém a lista dos elementos que constituem um estado, tais como id do estado, filhos, irmãos, links, conjuntos de marcação, conjuntos das transições, e conjuntos de sinais de eventos de saída.
- *ss_exec_step.c* contém funções que são responsáveis pelo cálculo de nós filhos.
- *ss_hashtable.c* contém funções que calculam o número de estados, número de índice de estados, números de deadlocks, números de conflitos, números de estados não processados de uma rede. Também é responsável pelo armazenamento de estados na tabela *hash.*
- *ss_compat.c* contém funções responsáveis pela contagem de todos os disparos das transições.
- *ss_main.c* com todas as funções necessárias para a construção do espaço de estados.

Durante o cálculo do espaço de estados são armazenadas as contagens de todos disparos das transições, as informações sobre a rede, como os *bounds* de lugares, *deadlocks* e conflitos entre as transições que foram habilitadas simultaneamente.
Na Figura 3.4 encontra-se representado o fluxograma que representa o algoritmo da construção do espaço de estados da rede de Petri IOPT.

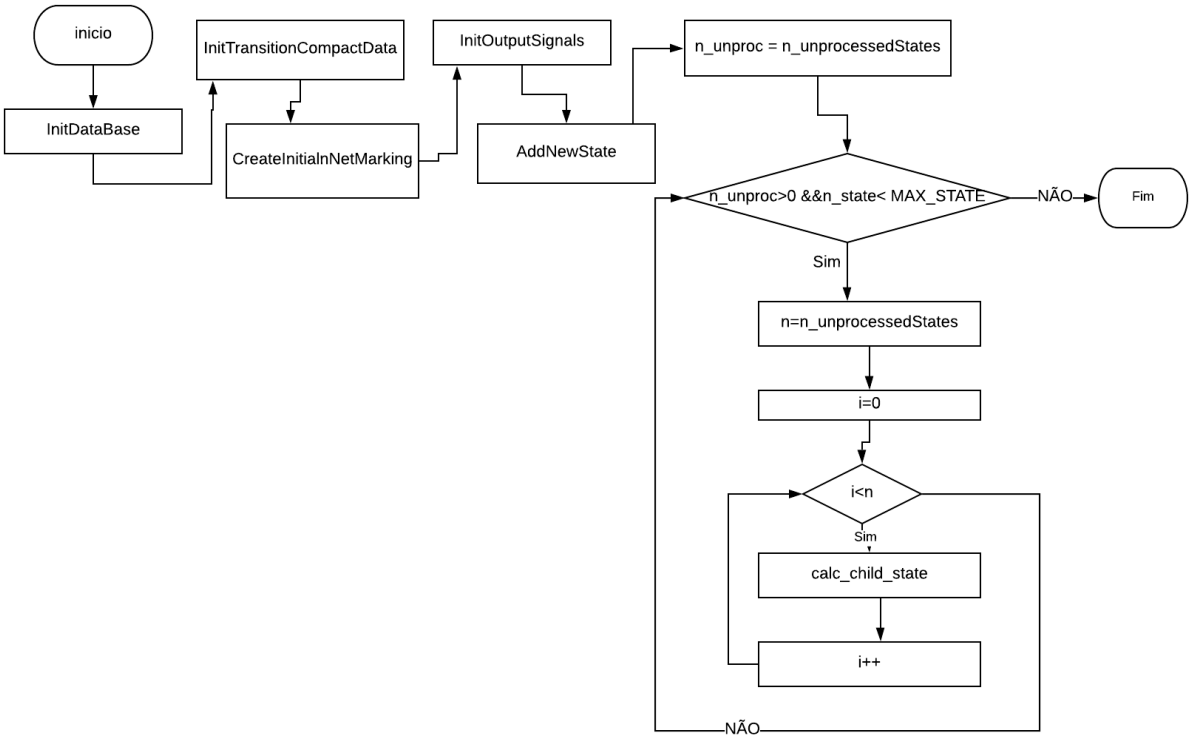


Figura 3.4: Fluxograma que representa a construção do espaço de estados da rede IOPT.

O algoritmo começa com a inicialização de *Database*, ou seja, reserva um espaço de memória para os estados da rede. Depois passa para a inicialização de todas as transições associadas aos sinais e eventos e em seguida pela criação de vetor de estado através da marcação inicial e de todos os valores de sinais de evento de saída. Após a criação do vetor do estado, é adicionado o nó do estado inicial à *Database* correspondente a marcação inicial e outputs de estados. Depois passa para um ciclo de criação de espaço de estados, que continuará até que todos os estados não processados sejam tratados, ou até que o grafo do espaço de estados atinja o número máximo de estados (“80000000”), que é possível calcular. Dentro do primeiro *loop* existe um segundo *loop* responsável a avaliar a cada índice de estado não processados e calcular os seus respetivos filhos, que tornarão os novos estados não processados, o processo *calc_child_states* que é responsável para os calcular.

O processo *calc_child_states* é uma função, que é executado para cada vetor de estado de entrada, ela calcula todos os nós filhos correspondentes as transições ativadas. Os nós filhos são armazenados numa tabela *hash* que armazena base de dados do espaço de estados.

Como resultado, o grafo do espaço de estados é composto por um conjunto de nós, representando todos os estados da rede, e conjunto de arcos conectando cada nó aos respetivos filhos (um exemplo típico se encontra representada na Figura 2.11 e Figura 2.12).

4

4 Adaptação do algoritmo para a construção do espaço de estados com recursos a GPU

Neste capítulo é apresentada a implementação do algoritmo da construção do espaço de estados dos modelos da rede Petri IOPT em GPU (*Graphics Processing Units*), e como é efetuado o seu processamento paralelo em CUDA (*Compute Unified Device Architecture*).

No capítulo da introdução foi apresentado que o principal objetivo desta dissertação é melhorar o tempo de processamento da construção do espaço de estados dos modelos da rede IOPT na GPU. Os modelos de sistema de controlo desenvolvidos em RdP são muito complexos, e com isto torna se difícil compreender melhor o seu comportamento. Devido à variedade e à dimensão das redes, os sistemas desenvolvidos em RdP podem apresentar um grafo do espaço de estados com muitos nós e arcos, e isso torna-se um problema sobre o ponto de vista computacional quando se pretende realizar a verificação das propriedades do modelo. Isto porque, no grafo do espaço de estados há uma explosão de número de estados, ou seja, o grafo pode ser tão grande que dificulta a procura e análise de todos os estados que o modelo pode alcançar. Com utilização da GPU pretende se resolver este problema, aumentando o desempenho no processamento da construção do espaço de estados.

Para a implementação do algoritmo da construção do espaço de estados em GPU, foi utilizado um conjunto de funções C, resultante da geração do espaço de estado do

modelo no ambiente IOPT-Tools, adaptadas de forma a obter o processamento paralelo em CUDA

4.1 Modelo de processamento em CUDA

Para realização da construção do espaço de estados na GPU, foi considerado o uso da CUDA Toolkit da NVIIDIA. A CUDA Toolkit oferece um conjunto de ferramentas que permitem desenvolver e executar programas em CPU e em GPU seguindo um modelo de processamento heterogêneo. A CPU é conhecida como *host* e ela é responsável pela execução do programa. Encontra-se fisicamente separada da GPU, que é conhecida como *device* e é responsável pelo processamento das *threads* [24],[5]. Como a CUDA permite usar algumas linguagens de programação como C/C++, assim facilita a adaptação do código C disponível no gerador do espaço de estados da IOPT-Tools.

A Figura 4.1, mostra a estrutura completa do processamento paralelo em CUDA usado para a construção do espaço de estados em GPU.

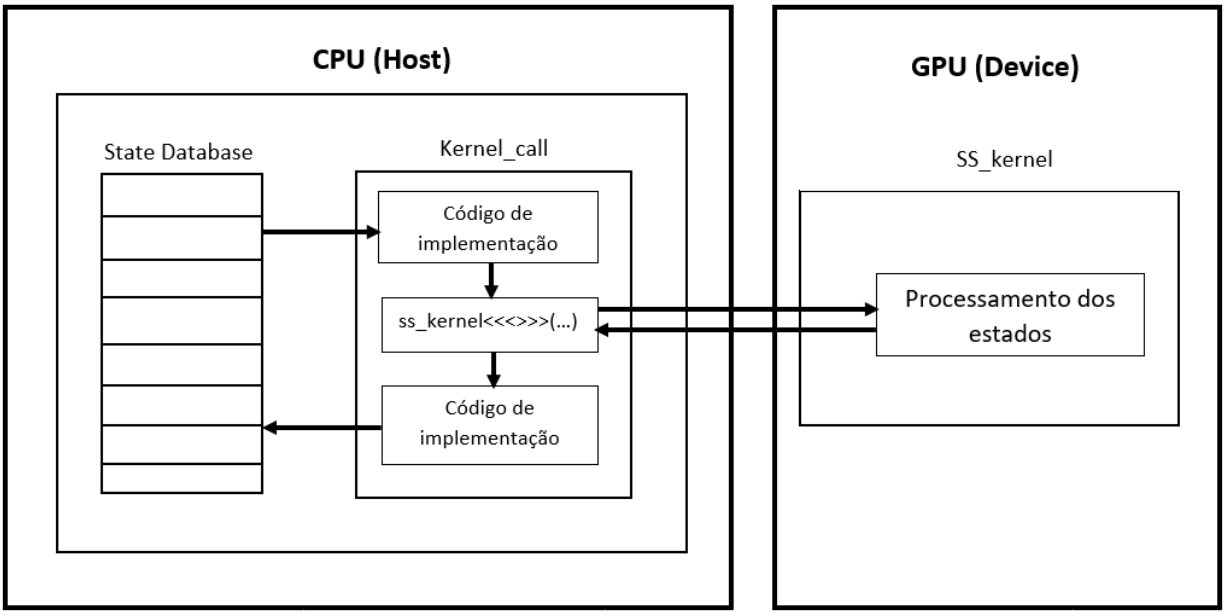


Figura 4.1: Processamento paralelo da construção do espaço de estados utilizando CUDA.

No capítulo 3 foi apresentado um fluxograma que representa a construção do espaço de estados. A construção do espaço de estados começa com a inicialização da *Database*, que contém o vetor de estados com valores de marcação inicial e valores de todos os sinais e evento de saída. Sendo assim o procedimento de processamento, inicia com a busca do número de estados não processados a *Database* ao correspondente nó do estado. O número de estados não processado é considerado igual ao número de *threads* que vão ser executados em GPU. A função *kernel_call* implementa o código para as sequências de *threads* e faz o lançamento do *kernel* na GPU para o processamento paralelo. Depois de processada as *threads* na GPU, o número de estados é enviado de volta para a *Database* para fazer a atualização destes estados tornando-os em estados processado. Os resultados enviados à *Database* são adicionados à parte inferior do vetor estado não processado. Esse procedimento é cíclico e termina quando não existirem mais estados a processar ou atingir o número máximo de estados.

4.1.1 Arquitetura de CUDA no processamento da construção do espaço de estados

Como se pode ver na estrutura de processamento da construção do espaço de estados (Figura 4.1), a CUDA assume que o sistema é composto por um *host* e um *device*, cada um com a sua própria memória.

A implementação do código na CPU (host) consiste em alocar e libertar a memória no *device* e na transferência de dados entre a memória do *host* e do *device*. Para isso, a CUDA fornece algumas funções como: *cudaMalloc(), cudaFree(), cudaMemcpy()* [24]. Essas funções são semelhantes aos *malloc*(), *free*(), e *memcpy*() do C. Ou seja, a sequência para realizar um processamento na GPU, corresponde a alocar um espaço de memória *device*, utilizando *cudaMalloc()*; realizar a transferência de dados para a memória device, usando *cudaMemcpy()* com o parâmetro *cudaMemcpyHostToDevice;* carregar o programa GPU e executar (lançamento de *kernel*), isto é, uma maneira de CPU passar o controle para GPU, ainda é especificado o número de *blocks* e *threads*; depois do processamento no *device* é efetuado uma cópia do resultado final para *host*, utilizando *cudaMemcpy()*, com o parâmetro *cudaMemcpyDeviceToHost:* e por ultimo, a libertação da memória *device* utilizando

cudaFree(). Esses são os procedimentos básicos utlizados num programa CUDA, que foram utilizados como a base para o processamento da construção do espaço de estados na GPU.

Um código *kernel* é responsável para o processamento das *threads* na memória *device*, sendo que a função é definida utilizando um qualificador __*global*__*.* Para além desse qualificador, existem outros que determinam se uma função é executada no *host* ou no *device*. Esses qualificadores foram utilizados na adaptação de funções do código para a construção do espaço de estados, pois precisam ser usadas no *kernel*. A função pode ser declarada com __*host*__, __*global*__, e __*device*_*.* A função declarada com o qualificador do tipo:

- __*host*__ é chamada e executada somente no *host*;
- __*global*__ é chamada no *host* e executada no *device*;
- __*device*__ é chamada e executada somente no *device*.

Depois da chamada de *Kernel* é feita uma sincronização do *device*, sendo esta uma forma de garantir que o *kernel* terminou o processamento atual. A sincronização é feita através de funções *cudaDeviceSyncronize()*.Esta função fica a aguardar até que todas as *threads* lançadas pelo *kernel* terminem o seu processamento [24].

Relativamente a gestão de memórias e de execução de *kernel*, pode-se notar que ambos dependem das *threads* logo, é preciso saber programá-las. Quando se trata de executar todas as *threads* no mesmo *kernel*, a execução deve ser feita através de um índice único que é calculado com o índice de *thread* identificado por *threadIdx*, a dimensão de *block* identificado por *blockDim* e índice de *block* identificado por *blockIdx.*

As *threads* dentro de um *block* podem cooperar partilhando dados através de algumas *Shared Memory*, e sincronizando sua execução para coordenar a comunicação e acessos à memória utilizando a função __*syncthreads().* A função __*syncthreads().* atua como uma barreira na qual todas as *threads* no *block* devem aguardar antes de poder continuar com a comunicação e acesso a memória [24].

4.2 Algoritmo implementado

De acordo como foi referido, para o desenvolvimento desta dissertação foi utilizado as funções já implementadas da geração do espaço de estados de um modelo IOPT-Tools, que foram adaptadas de forma a obter funções que funcionasse em paralelo na GPU utilizando CUDA. Nos próximos parágrafos é dado mais ênfase a essas funções.

Para adaptar o código da geração do grafo do espaço de estados foi utilizado o algoritmo representado na Figura 3.4. Neste algoritmo é apresentada um loop que executa a função *calc_child_state*, que é responsável para calcular os nós filhos para cada análise de estado não processados. Ela foi substituída pela função *kernel_call* representada na Figura 4.2, que será uma função que executará o paralelismo do processamento da construção do espaço de estados.

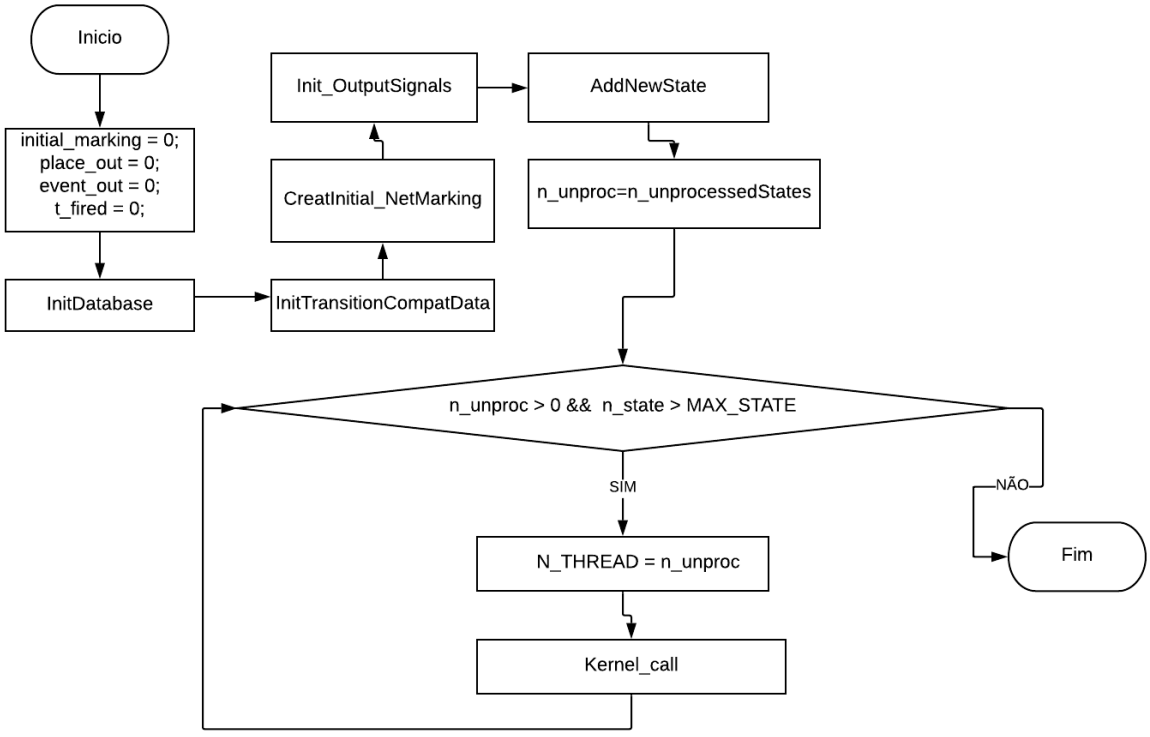


Figura 4.2:Fluxograma que representa o algoritmo principal sobre o processamento da construção do espaço de estados na GPU.

Na Figura 4.2, encontra representado o algoritmo principal responsável pelo processamento do espaço de estados dos modelos da rede Petri IOPT. O algoritmo começa com a inicialização de *Database*, ou seja, é reservada um espaço da memória

para os vetores de estados da rede. Os processos *InitTransitionCompatData, CreatInitial_NetMarking e Init_OutputSignals* são as funções que são responsáveis pela criação dos elementos de um vetor de estado, ou seja, inicializam todas as transições associadas aos sinais e eventos de saída, e criam o vetor de estado correspondente a marcação inicial da rede. Após a criação do vetor de estado, o nó que representa o estado inicial é adicionado à *Database*

Em seguida, dá-se início ao ciclo de preparação para o processamento paralelo dos estados não processados que permite a construção do espaço de estados. O número das *threads* a processar será igual ao número de estados não processados, que é o parâmetro de entrada para a função representado no processo *Kernel_call*. O ciclo é executado enquanto existirem estados para processar e o número de estados não atingir o máximo.

Na Figura 4.3, encontra se representado o algoritmo da função *Kernel_call*. Esta função é responsável pela gestão da memória do *device*, pela transferência de dados entre *host* e *device* e pelo lançamento de *kernel*.

A função começa em alocar a memória para as variáveis *devices* que é utilizados para o processamento, através de *cudaMalloc()*. As variáveis *devices* são: o número de estados não processados, a lista de estados não processados e o número de estados não processados mais os links, ou seja, número de nós filhos encontrados no ciclo anterior. Em seguida é efetuada a cópia dos valores das variáveis definidas no *host* para as variáveis *device* através de função *cudaMemcpy()*, e também é inicializada a variável que guarda o número dos nós filhos encontrados no ciclo em que está a ser executado através da função *cudaMemset()*, com o valor sempre inicializado a zero.

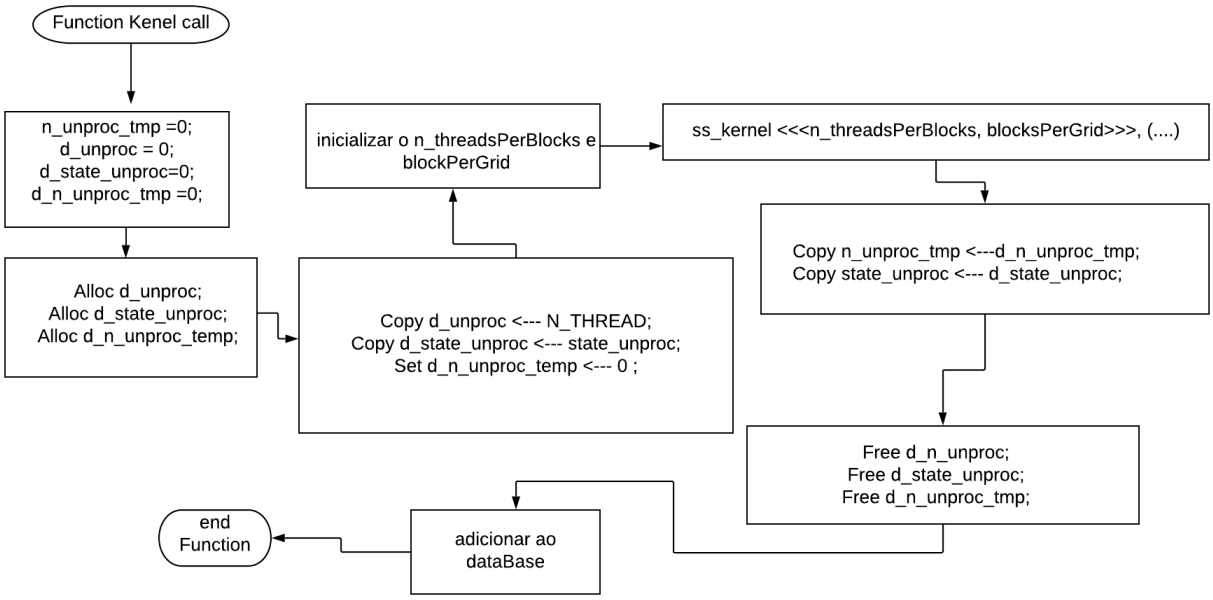


Figura 4.3: O algoritmo que descreve a função Kernel_call utilizado para o processamento da construção do espaço de estados na GPU.

Depois de alocar a memória e transferir os dados para o *device*, o passo seguinte foi o cálculo do número de *blocks* por *grid* e *threads* por *block* que caracterizam o *kernel* a ser lançado.

O número de estados não processados pode ser um número pequeno para o CPU, mas grande para ter *blocks* suficientes para manter as GPUs em execução mais rápidas. Sendo assim foi feita uma análise do grafo do espaço de estados de acordo com o número máximo de estados não processados. Foi então atribuído um valor constante para o número de *threads* por *blocks*, sendo um valor próximo do valor máximo, dependendo do tamanho do modelo da rede. Para isso foi escolhido um máximo de 3 blocks por grid, embora seja um caso que poderá apresentar um melhor ou pior desempenho de GPU. Se existir N estados não processados, precisa-se apenas de N *threads* para processar os estados. Portanto neste caso, precisa-se do menor múltiplo de n_*threadsPerBlocks* (número de Threads por blocks) e *blocksPerGrid* (Blocks por Grid) que é maior ou igual a N, para isso foi preciso saber quantos *blocks* por *grid* seria necessário, sendo assim foi obtido da seguinte forma:

```
const int blocksPerGrid = imin(3,(N + n_threadsPerBlocks - 1) / n_threadsPerBlocks );
```
(1).

Portanto o número de *block* lançado ser 3 ou, (N + n_threadsPerBlocks - 1) / n_threadsPerBlocks), depende do que for menor.

Em seguida é feito o lançamento do *kernel*:

```
ss_kernel<<<blocksPerGrid, n_threadsPerBlocks>>>                    (2)
                    (d_n_unproc, d_state_unproc, d_n_unproc_temp);
```

Na Figura 4.4, encontra se representado o algoritmo que executa a função *ss_kernel*. A função começa por executar o cálculo de *thread Index*, dado pela seguinte formula:

```
unsigned int index = threadIdx.x + (blockIdx.x * blockDim.x);      (3)
```

O *thread index* trabalha em um *grid* unidimensional. O *index* faz com que o código *kernel* seja executado várias vezes, enquanto tiver estados não processados para serem analisados. Cada *index* é atribuído a um *thread,* que adquirirá uma lista de estado. Em seguida é executada a função *Calc_ChildrenState* que calcula todos os nós filhos correspondentes as transições ativadas, associada ao vetor do estado de entrada. Os nós filhos são adicionados à parte inferior do vetor de estado não processados. E por último a função *__syncthread()* garantirá que cada *thread block* tenha concluído as instruções anteriores.

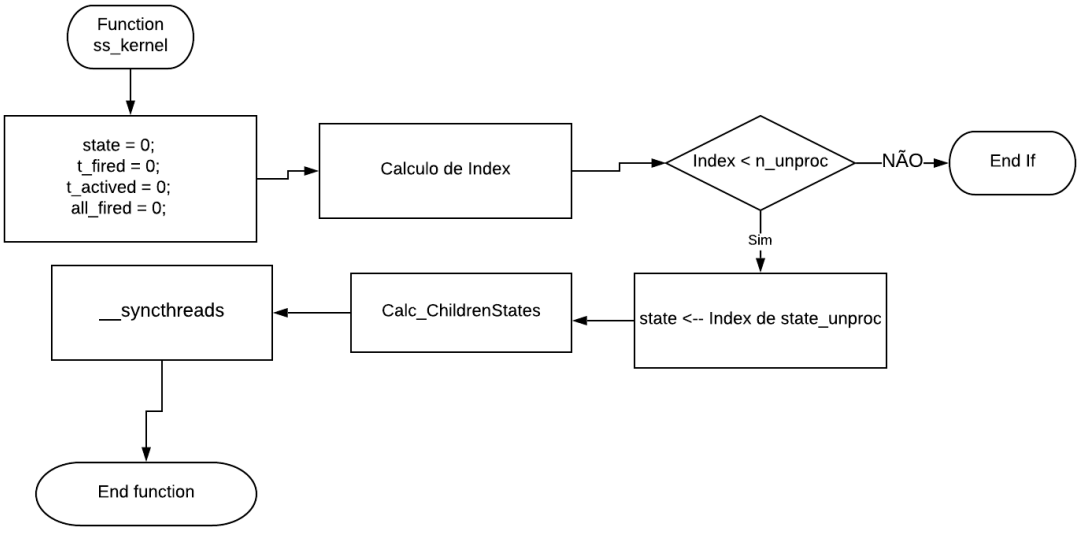


Figura 4.4: O algoritmo que descreve a função ss_kernel utilizado para o processamento do espaço de estados na GPU.

Depois que o *kernel* terminar é feito a cópia dos valores do *device* para *host* através de *cudaMemcpy()*. Os valores de saída copiados para o *host* são usados para atualizar a *Database* e posteriormente, forneçe o novo conjunto de nós a serem analisados no próximo ciclo. Assim quando o *kernel* termina, a análise da *Database* continua a ser executada no CPU.

5

5 Resultados

Neste capítulo são apresentados alguns exemplos dos modelos utilizados, suas características e os resultados obtidos no processamento da construção do espaço de estados na GPU. Para validar os resultados da aplicação implementada, foram utilizados alguns modelos de redes de Petri IOPT em que alguns destes encontram-se publicados em [38].

O *hardware* utilizado para validação dos resultados, é uma CPU e uma GPU, com as características descritas na Tabela 5.1. A CPU e a GPU comunicam-se através de um bus PCI-Express x16 Gen3, apresenta um total de 32 GB/s de largura de banda bidirecional, ou seja, oferece 16 GB/s da largura de banda em cada direção. Esta PCI transfere dados a uma velocidade de 8 GT/s (GigaTransfers per second).

Para a execução da construção do espaço de estados foi utilizado a CUDA Toolkit, que nos permite ter um ambiente de programação heterogéneo onde é possível utilizar os dois processadores num mesmo programa.

Tabela 5.1: Característica do CPU e GPU utilizados durante a validação dos resultados

	CPU	**GPU**
Processadores	Intel® Core™ i5-4690K	TITAN V
Frequência de memória	3.50 GHz	850 MHZ
Memória	8.00 GB	12.00 GB
Números de cores	4	5120
Interface de Memória	64 bits	3072 bits

5.1 Modelos

Nesta secção são apresentados os modelos que foram utilizados no teste da aplicação desenvolvida para a construção do espaço de estados.
Para o teste foram consideradas algumas características dos modelos de redes Petri IOPT, tais como, o número de lugares, número das transições, os sinais e eventos de entrada e de saída, o número de ciclos, o número de nós (estados mais os links).
Para isso, foram escolhidos alguns modelos usados para o teste e validação dos resultados, nos quais foi usada a nova estratégia para a construção dos espaços estados na GPU.

5.1.1 Park2in1out

O exemplo do modelo representado na Figura 5.1 encontra-se disponível em [35], tendo sido já utilizado em algumas aplicações [39]. Este modelo modela um controlador de um parque de estacionamento com duas entradas, uma saída e com capacidade para um veículo.
O modelo representado é composto por onze lugares, quatro deles com *tokens* a representar a marcação inicial, nove transições e um total de doze sinais e eventos de entrada e três sinais de saída. O grafo do espaço de estados deste modelo obteve-se em nove ciclos, apresentando duzentos e sessenta e nove nós (cinquenta e quatro estados mais duzentos e quinze *links*), e um máximo de treze estados não processados.

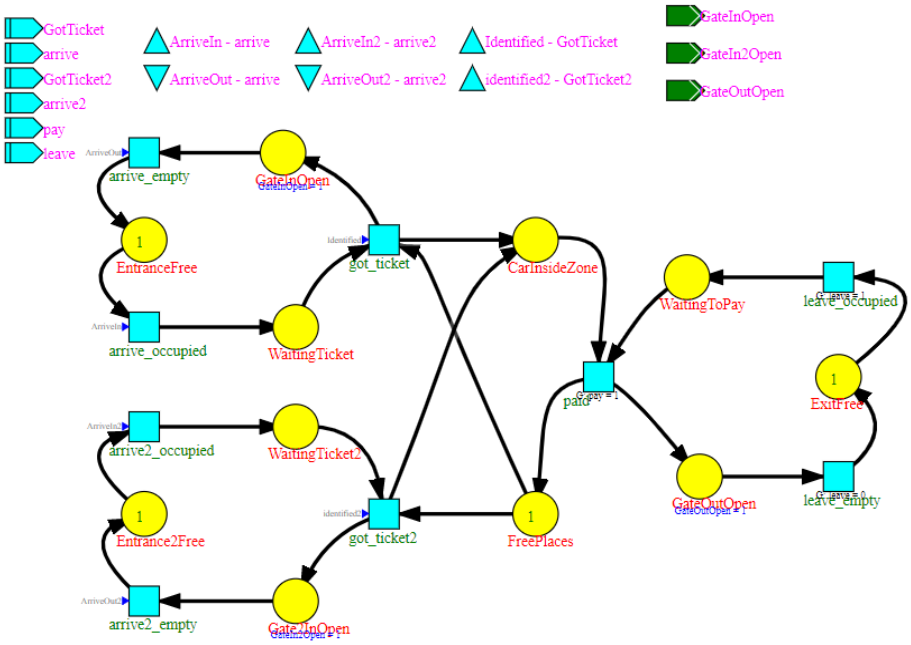


Figura 5.1: Modelo Park2in1out.pnml

Para a construção do espaço de estados deste modelo na GPU, foram considerados alguns elementos importantes da rede, tais como o número de nós (estados mais links), e o número dos estados não processados em cada nó. Isto porque, esses elementos influenciarão no cálculo da reserva de memória no *device* e na determinação de números de *threads* por *block*. No entanto, foram utilizados 75 mais *n_estado_nao_processados* (número de estados não processados) para o tamanho de espaços de memória no *device,* para as variáveis que guardam a lista dos estados não processados. Foi considerado este tamanho, porque o modelo apresenta um máximo de 75 nós, que corresponde ao número de nós filhos que foram calculados com base nos estados não processados enviados para a GPU. Também foram consideradas 5 *threads* por *block*, devido ao número máximo de estados não processados que o modelo apresenta no seu grafo do espaço de estados. O número de *blocks* considerado foi um máximo de 3, isto porque o modelo apresenta um máximo de treze estado não processados e será necessário de um total de treze *threads* para processar na GPU, sendo que para manipular as *threads* foi considerado que o múltiplo de *blocks* por *grid* e *threads* por *block* seja igual ou superior a números de estados não processados enviados para o *kernel*.

O tamanho de espaço que foi utilizado para alocar a memória *device* e a determinação de *threads* por *blocks* e de *blocks* por *grid* fornece o tempo de execução na GPU.

5.1.2 ICIT13_quad_encoder

O segundo modelo escolhido representado na Figura 5.2, encontra-se publicado em [35]. Este modelo modela um controlador responsável pela contagem de pulsos produzidos por um codificador presente na placa FPGA [38].

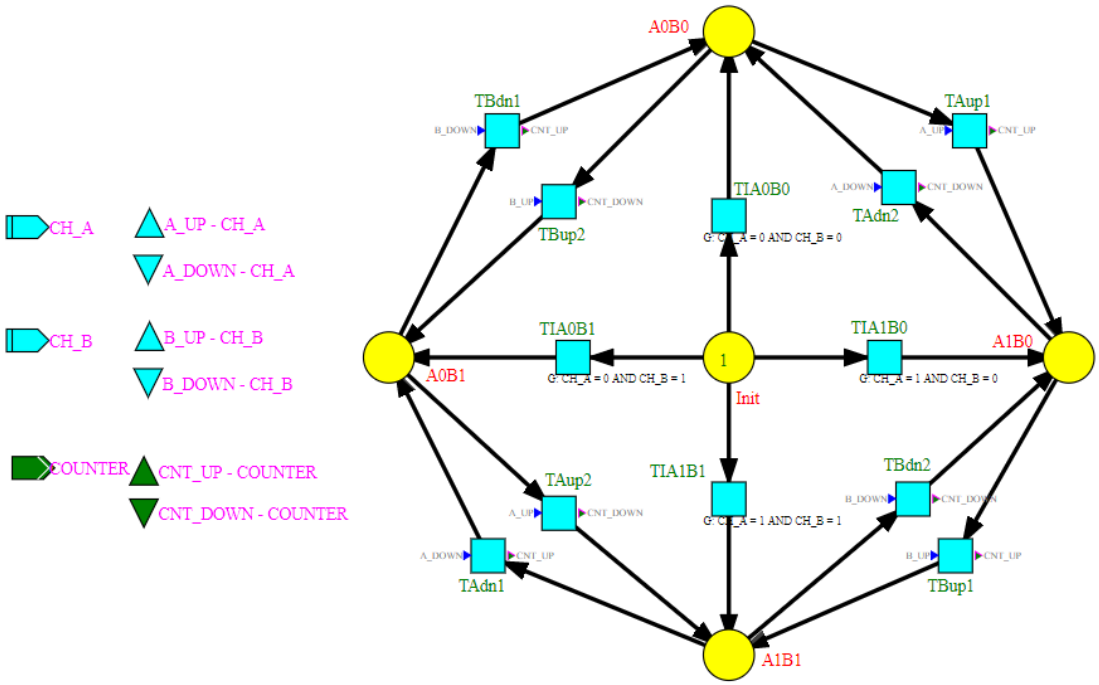


Figura 5.2: Modelo ICIT13_quad_encoder.pnml

O modelo é constituído por cinco lugares, doze transições, dois sinais e quatro eventos de entrada, um sinal e dois eventos de saída.

O grafo do espaço de estados deste modelo obteve-se em cento e trinta ciclos, apresentando dois mil e quarenta e cinco nós (mil e vinte e cinco estados e mil e vinte links) e um máximo de 8 estados não processados.

Para a construção do espaço de estados deste modelo na GPU, foi utilizado o mesmo raciocínio que o modelo anteriormente representado, sendo o que os diferencia é o cálculo de tamanho de espaço de memória e a determinação do número de *threads* por *blocks*. No entanto foi considerado 16 mais o *n_estado_nao_processados* para o tamanho de espaços de memória no *device* para as variáveis que guardam a lista dos estados não processados e 3 *threads* por *block*.

Para além dos modelos já mencionados, também foram utilizados outros modelos, que se encontram representados na Tabela 5.2, e as suas respetivas características.

Tabela 5.2: Modelos Utilizados na análise do algoritmo de construção de espaço de estados na GPU.

Modelo	Lugares	Transição	Input	Output	Ciclos	Estados	Links	n_nós (State+links)	n_max_estados_nao_processados
FMMS13_car3	12	8	8	6	5	16	13	29	7
Park2in1out	11	9	12	3	9	54	215	269	13
Park2levels	18	24	16	10	11	135	1225	1360	26
Concrete_mixer2_6x	25	20	12	8	33	676	2162	2838	34
Concrete_mixer_6xA	13	11	11	6	56	110	108	218	3
ICIT13_quad_encoder	5	12	6	3	130	1025	1020	2045	8
ICIT13_pwm_gen	4	6	3	5	1025	4096	12287	16383	5
Concrete_mixer	10	10	11	6	9208	9216	18412	27628	3

5.2 Análise dos resultados

Para o teste de análise foi considerado o tempo de execução que a GPU gasta para calcular todos os estados que compõe um grafo do espaço de estados de um modelo. O tempo gasto pela GPU depende do tamanho de espaço de memória *device* que é alocado pelo *host*, dos dados transferidos entre a memória do *host* e a memória do *device* e da determinação de números de *threads* por *block* a ser usado num *kernel*. A alocação do tamanho de espaço de memória *device*, foi baseada no número máximo de nós filhos que é determinado num grafo do espaço de estados, o número de *threads* por *block* foi baseada no número máximo de estados não processados que o grafo do espaço de estados pode apresentar. Com isso, pode-se dizer que o tamanho do grafo do espaço de estado teve a influência sobre os resultados obtido.

No Gráfico 5.1 encontra-se representado o tempo gasto em GPU na construção do espaço de estados utilizando os modelos de rede Petri IOPT.

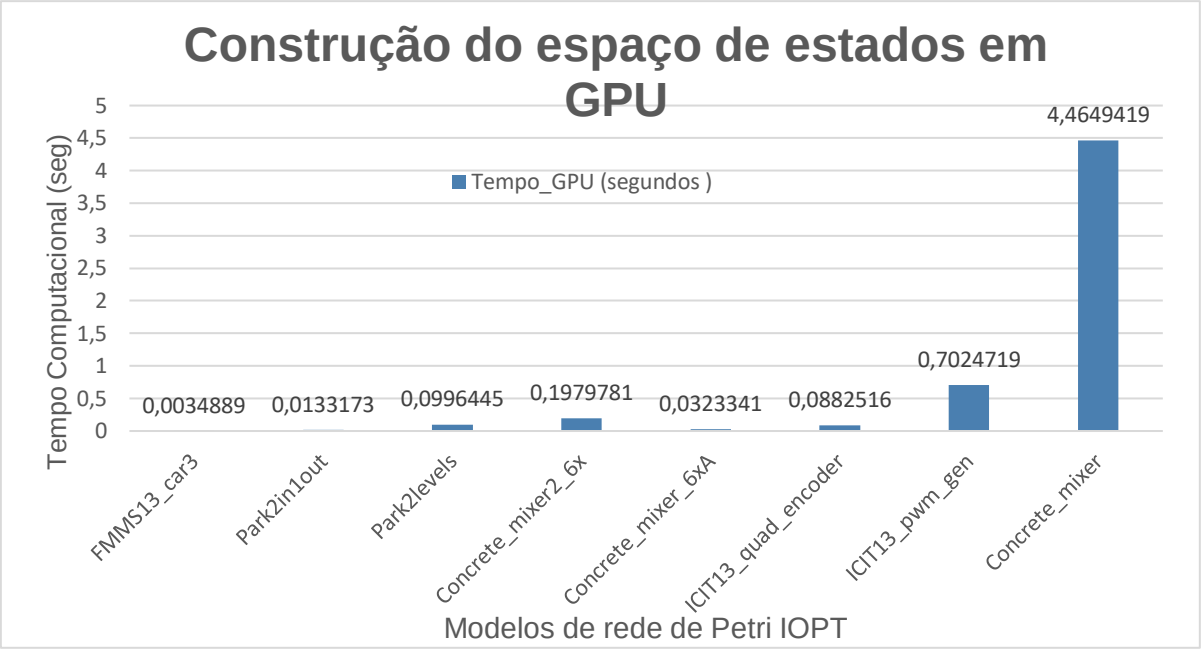


Gráfico 5.1: Representação do tempo obtido na construção do espaço de estados em GPU dos modelos de rede de Petri IOPT.

Os modelos encontram representados por ordem crescente dos ciclos que estão representados na Tabela 5.2. Pode-se verificar que há modelos que apresentam poucos ciclos com muitos estados, poucos ciclos com poucos estados, muitos ciclos com muitos estados e muitos ciclos com poucos estados. Dos resultados obtidos, os modelos de poucos ciclos com muitos estados e muitos ciclos com muitos estados, são os que levaram mais tempo para terminar de analisar os estados não processado e calcular os seus nós filhos.

A aplicação implementada usa uma *thread* para processar um estado não processados, ou seja, cada *thread* calcula os estados filhos de um nó. Se em um ciclo houver poucos estados não processados, só serão usados algumas *threads*, não sendo considerado o número máximo de *threads* ativas em cada *block*. Isso faz com que a GPU gasta menos tempo em processar as *threads*.

Analisando os modelos *concrete_mixer2_6x* e *concrete_mixer_6xA* verifica-se que o primeiro modelo apresenta poucos ciclos em relação ao segundo modelo, mas o primeiro apresenta dez vezes mais estados do que o segundo. Isso fez com que demorasse mais tempo para processar os estados na GPU. O modelo *concrete_mixer2_6x* apresenta um máximo de trinta e quatro estados não processados, sendo usado um máximo de trinta e quatro *threads* em paralelos, enquanto que o segundo é usado um máximo de três *threads* ativas em paralelos.

Com isso pode-se dizer que quanto mais estados não processados um nó tiver, mais *threads* em paralelo será enviado para *kernel*, isso faz com que a GPU leva mais tempo para terminar a execução do espaço de estados.

Para analisar o desempenho computacional, foi preciso saber o tempo que a GPU levou para calcular todos os estados, bem como o tempo gasto na CPU. O tempo gasto na GPU e na CPU encontra se representada no Gráfico 5.2, através deste podemos ver facilmente que o tempo em GPU foi menor que o tempo em CPU.

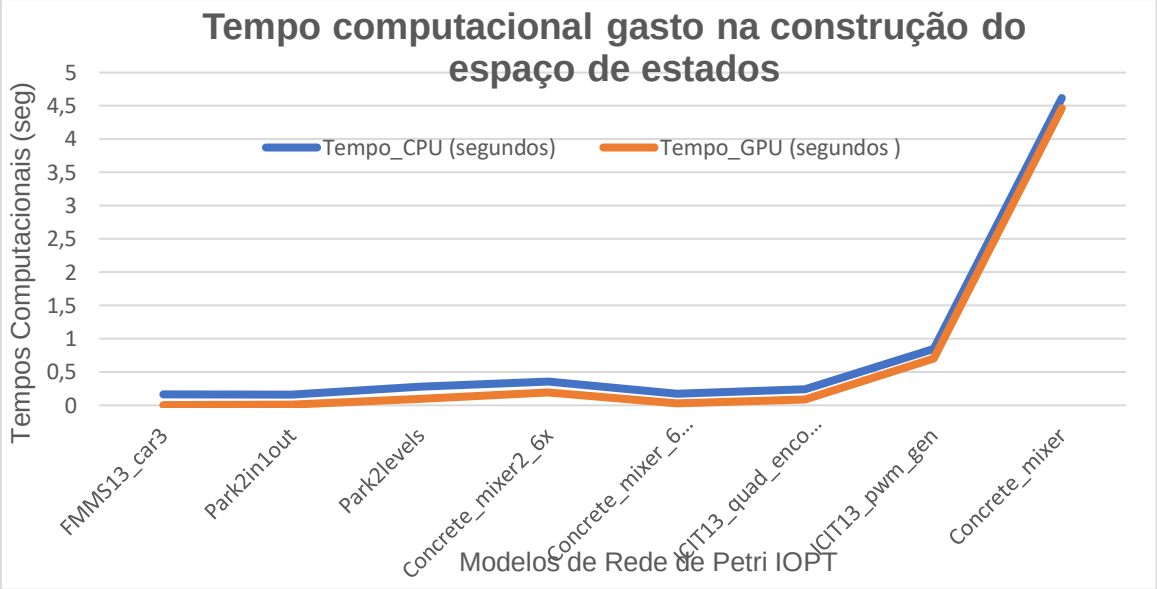


Gráfico 5.2:Comparação do tempo gasto na construção do espaço de estados entre CPU e GPU.

Através do tempo obtido pela CPU e GPU, foi obtido o desempenho em percentagem da GPU, representada no Gráfico 5.3.

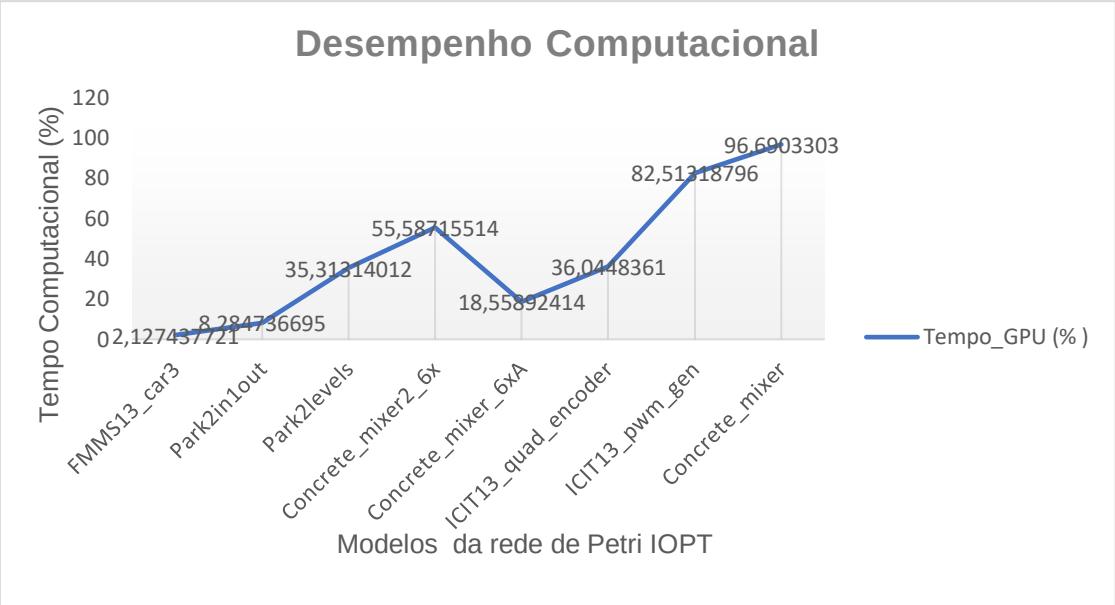


Gráfico 5.3: Desempenho gasto pela GPU na construção do espaço de estados.

O desempenho obtido é a razão entre o tempo gasto em GPU e em CPU, ou seja, o tempo que o *kernel* leva para executar uma função como um recurso critico.
Do Gráfico 5.2, podemos observar que o tempo em GPU foi sempre menor que em CPU, mesmo sabendo que a GPU é que realiza as tarefas mais intensa. Com isso apresentou um bom desempenho em termos dos resultados obtidos.

5.2.1 Validação do resultado no modelo Park2in1out e ICIT13_quad_encoder

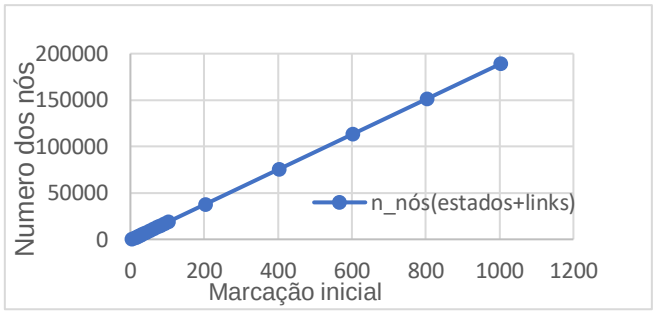
Para além dos elementos já mencionados, a marcação inicial presente nos modelos afetará a dimensão do grafo do espaço de estados. O tamanho do grafo do espaço de estados é expresso pelo número de ciclos e nós exibidos. Com isso, foram escolhidos os dois modelos representados na Figura 5.1 (park2in1out) e na Figura 5.2 (ICIT13_quad_encoder).
O modelo park2in1out, é um controlador que modela um parque de estacionamento com duas entradas, uma saída e um lugar livre de estacionamento. O objetivo é o de ao aumentar o número nos lugares livres de estacionamento (que determinam a capacidade do parque) concluir sobre o comportamento do grafo representando o espaço de estados, utilizando a GPU.

Na Tabela 5.3 encontra se os valores que determina a capacidade do parque utilizado como marcação inicial no lugar (*place*) que corresponde a lugares livres e o tamanho do grafo do espaço de estados que corresponde a cada marcação inicial.

Tabela 5.3: Marcação inicial correspondente a capacidade do parque e o tamanho do grafo do espaço de estados.

Park2in1out				
Marcação inicial (Capacidade do parque)	**ciclos**	**Estados**	**links**	**n_nós(estados+links)**
1	9	54	215	269
10	20	297	1674	1971
15	26	432	2465	2897
20	35	567	3294	3861
25	41	702	4085	4787
30	50	837	4914	5751
40	65	1107	6534	7641
50	80	1377	8154	9531
60	95	1647	9774	11421
70	110	1917	11394	13311
80	125	2187	13014	15201
90	140	2457	14634	17091
100	155	2727	16254	18981
200	305	5427	32454	37881
400	605	10827	64854	75681
600	905	16227	97254	113481
800	1205	21627	129654	151281
1000	1505	27027	162054	189081

Pode-se ver no Gráfico 5.4 que a partir de duzentos lugares livres de estacionamento, ocorreu um aumento significativo da dimensão do grafo do espaço de estados, tanto no número do ciclo quanto no número dos nós.



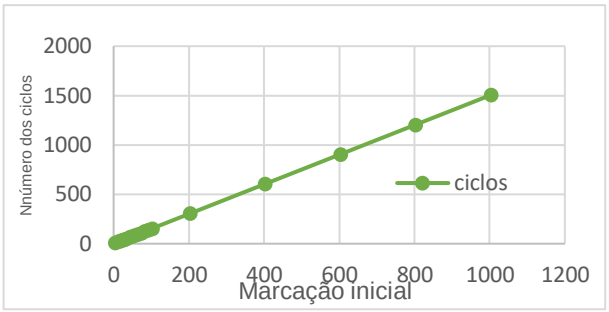


Gráfico 5.4: Tamanho de grafo do espaço de estados do modelo park2in1out correspondente a marcação inicial que representa a capacidade do parque.

Para cada valor correspondente a capacidade do parque, foi feito a validação do modelo utilizando a aplicação implementada e uma cópia da aplicação gerada no servidor IOTP-Tools, onde foram obtidos os tempos computacionais gastos na construção do espaço de estados, representado na Tabela 5.4.

Tabela 5.4: Valores obtidos na construção do espaço de estados, utilizando modelo park2in1out com a marcação inicial que corresponde a capacidade do parque.

Park2in1out			
Marcação inicial (capacidade do parque)	**Tempo em CPU utilizando a aplicação implementada (seg)**	**Tempo em GPU utilizando a aplicação implementada (seg)**	**Tempo em CPU utilizando a cópia do algoritmo do servidor IOPT-Tools (seg)**
1	0,160745	0,0133173	0,000239
10	0,2354472	0,0678431	0,000523
15	0,257336	0,1011731	0,000652
20	0,285086	0,1246771	0,000806
23	0,314151	0,1517835	0,000962
30	0,347003	0,1959679	0,001078
40	0,39955	0,2365219	0,001388
50	0,436143	0,2927663	0,001646
60	0,503238	0,3537876	0,001944
70	0,566506	0,4178553	0,002285
80	0,632056	0,4921346	0,002585
90	0,665651	0,5193667	0,002819
100	0,737233	0,6031517	0,003093
200	1,312432	1,1966482	0,00609
400	2,418332	2,3043005	0,011852
600	3,467399	3,3872495	0,0185
800	4,512695	4,4170728	0,024357
1000	5,496024	5,3983589	0,030466

Na construção do espaço de estados utilizando a aplicação implementada, os resultados obtidos em GPU e em CPU são diretamente proporcionais a marcação inicial no lugar (place) que corresponde a capacidade do parque de estacionamento. A representação dos resultados obtidos nos dois processadores encontra-se representado no Gráfico 5.5. Pode se ver que o tempo gasto na construção do espaço de estados na GPU aumenta à medida que é aumentado a marcação inicial no lugar (*place*), que representa a lugares vagos no parque. Quando é aumentado a marcação inicial a dimensão do grafo do espaço de estados aumenta e a GPU demora mais tempo para terminar a execução da construção do espaço de estados.

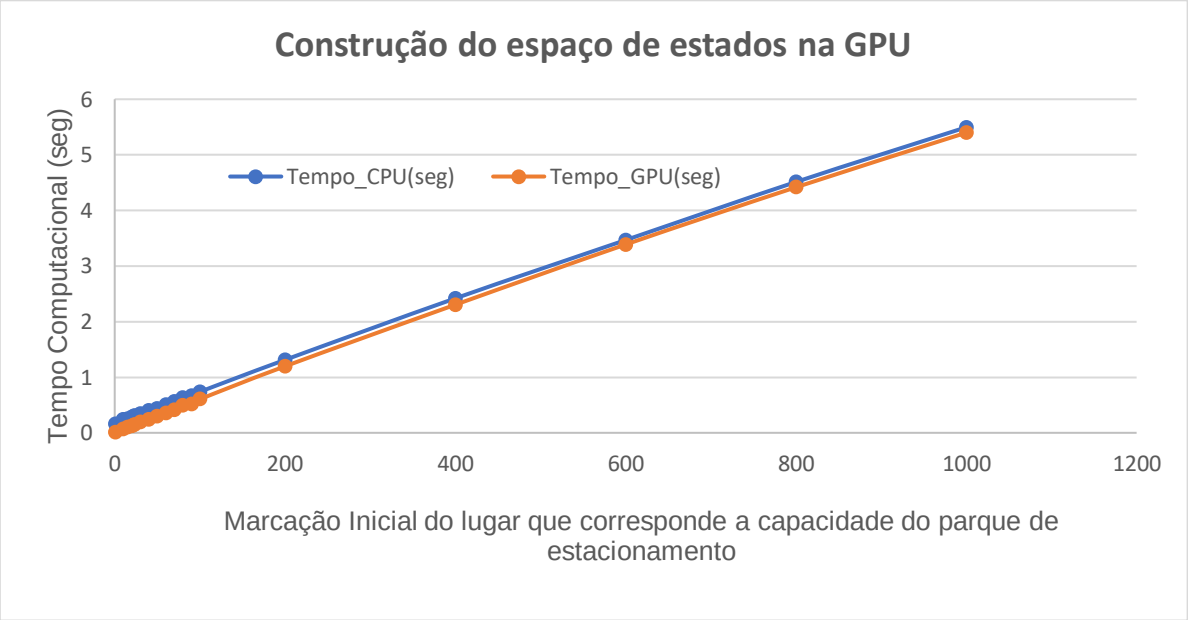


Gráfico 5.5: Tempo gasto na Construção do espaço de estados do modelo park2in1out na GPU referente a marcação inicial do lugar que corresponde a capacidade do parque de estacionamento

Ainda no Gráfico 5.5, é visível que o tempo em GPU é menor que em CPU. Na aplicação implementada para a construção do espaço de estados em GPU, há uma comunicação entre a CPU e a GPU, ou seja, a CPU faz a parte sequencial do programa que é responsável por transferir os dados para a GPU. Enquanto a GPU estiver a tratar dos dados recebidos, a CPU fica à espera de receber os dados que foram processados pela GPU. O tempo que a CPU fica a espera é contabilizado e isto faz com que em GPU seja menor que em CPU.

Ainda no modelo *park2in1out*, foi aplicado as mesmas estratégias de aumentar a capacidade do parque na aplicação implementada no servidor IOPT-Tools. A

aplicação é executada apenas em CPU. Sendo assim, foi feita uma comparação em termos de tempos gasto em CPU na construção do espaço de estados utilizando a aplicação implementada e utilizado a aplicação implementada pelo servidor IOPT-Tools. Os valores dos resultados obtido em CPU encontram se na Tabela 5.4 e representado no Gráfico 5.6.

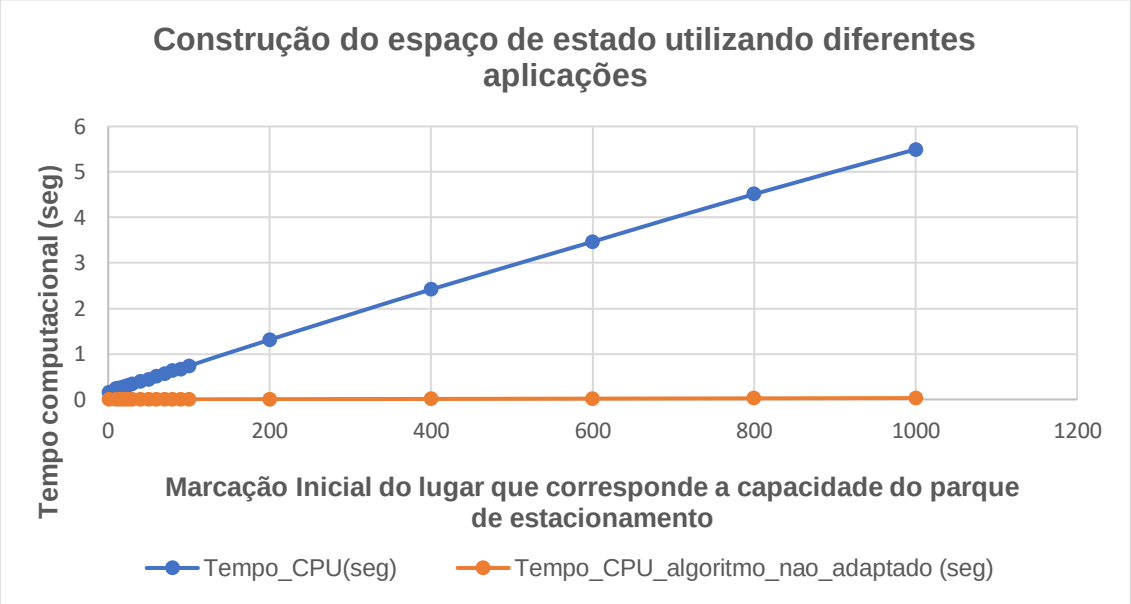


Gráfico 5.6: Tempo gasto em CPU na Construção do espaço de estados do modelo park2in1out utilizando diferentes aplicações, com a marcação inicial do lugar que corresponde a capacidade do parque de estacionamento

No Gráfico 5.6, mostra nos que o tempo gasto em CPU na construção do espaço de estados utilizando a aplicação implementada é muito superior que o tempo gasto em CPU utilizando a aplicação gerada pela IOPT-Tools. Na aplicação implementada, há uma comunicação entre a CPU e a GPU, e isto faz com que consome tanto tempo na construção do espaço de estados, ou seja, a CPU tem de ler os estados não processados e enviá-los para a GPU, isto é, lançamento das *threads* no *kernel*. Depois que o *kernel* terminar, a CPU é responsável por guardar os estados processado e enviar novo estados não processados para a GPU. Enquanto que na aplicação do servidor a informação sobre o estado já se encontra disponível no grafo do espaço de estados e a CPU só tem a necessidade de ler os estados não processados e executar. Há casos que com o aumento da marcação inicial do modelo no lugar denominado por *Init*, aumenta apenas o tamanho do grafo, ou seja, aumenta o número de nós do grafo do espaço de estados. Este exemplo do modelo encontra se representado na

Figura 5.2 (ICIT13_quad_encoder), que é um codificador que faz contagem dos pulsos da placa FPGA.

Pode-se ver que no Gráfico 5.7, há um aumento significativo do tamanho do grafo do espaço de estados. Isto resulta em ter muitos estados não processados e com isso faz com que a GPU demore mais tempo na construção do espaço de estados.

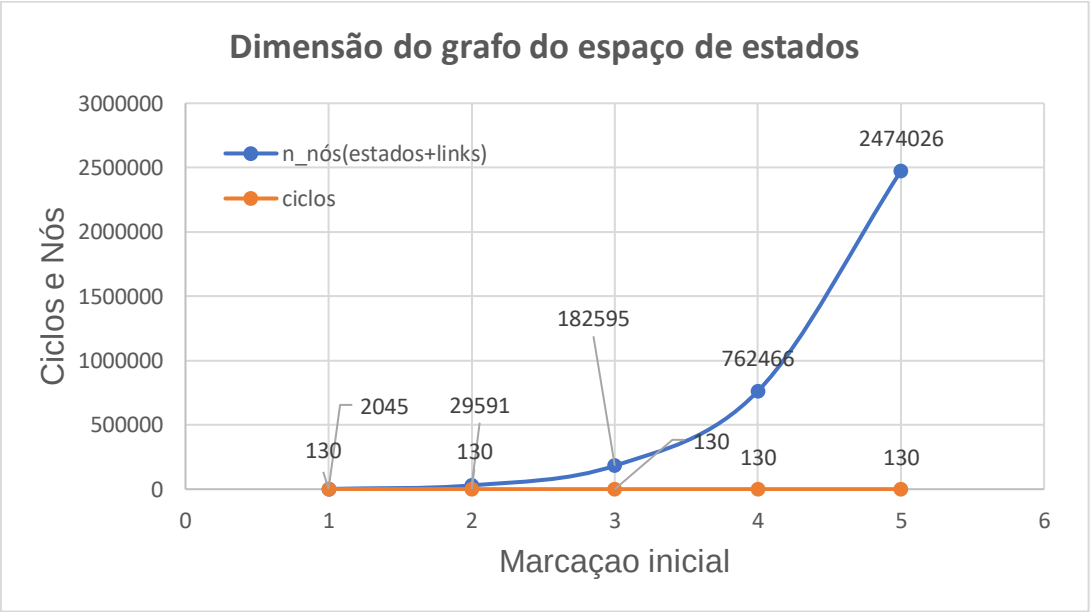


Gráfico 5.7: Dimensão do grafo do espaço de estados do modelo ICIT13_quad_encoder referente a marcação inicia no lugar denominado por Init.

No Gráfico 5.8 pode-se notar que o tempo gasto pelo GPU foi menor que o tempo gasto em CPU, mas não houve muita diferença em termos de tempo gasto, ou seja, tiveram valores muitos próximos. Isto porque num nó tem muitos estados não processados e com isso é utilizado um grande número de *threads* para processar os estados, e quanto mais *thread* a ser utilizado mais tempo levará para terminar a construção do espaço de estados.

Neste modelo deu para notar que em poucos números de marcação inicial, resultou em um grafo do espaço de estados muito grande, ou seja, o modelo tem a tendência em atingir facilmente o número máximo de estados.

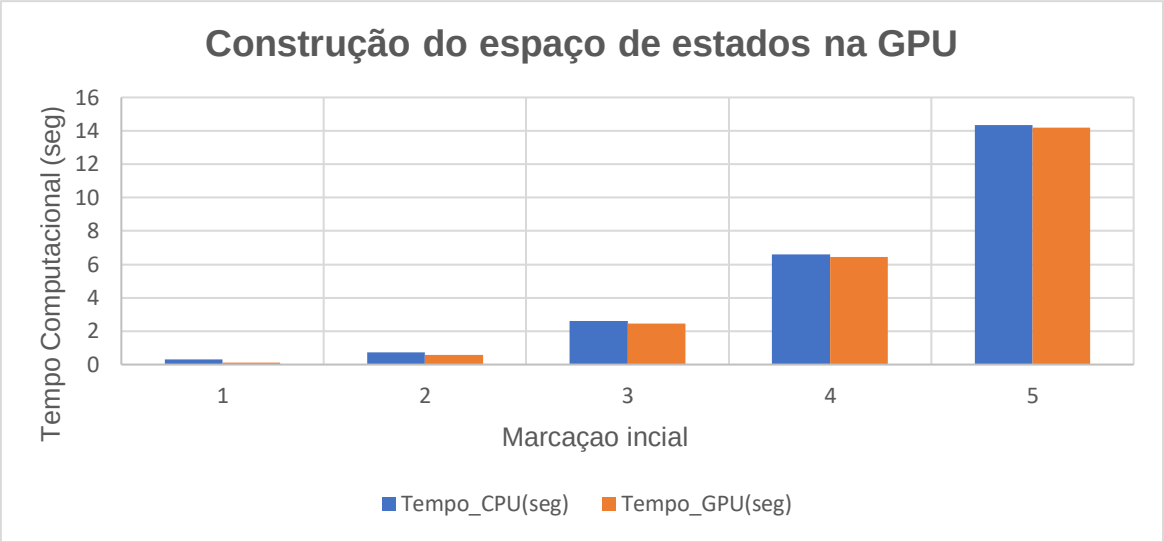


Gráfico 5.8: Tempo gasto na Construção do espaço de estados do modelo ICIT13_quad_encoder na GPU referente a marcação inicial.

A existência de grande número de estados não processados e à utilização de um grande número de threads em paralelos, a GPU precisou de levar mais tempo para calcular todos os estados, bem como o tempo gasto em CPU e com isso resultou em ter resultados elevados quanto a desempenho gasto pelo GPU, isto é, com o aumento da marcação inicial, aumentaram-se os estados não processados e o *kernel* levou mais tempo para terminar a execução. No Gráfico 5.9 encontra se representado o desempenho computacional, do modelo *ICIT13_quad_encoder* em relação a marcação inicial.

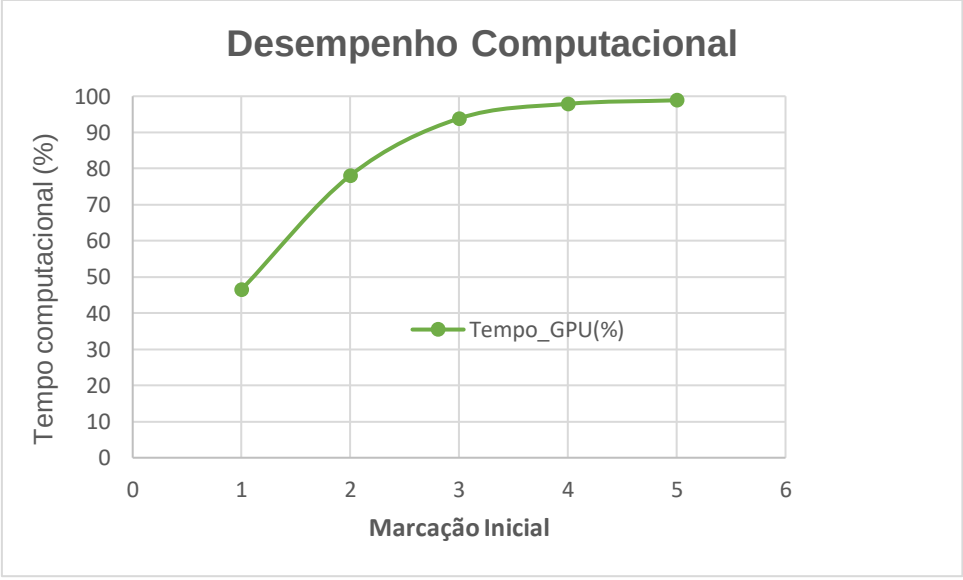


Gráfico 5.9: Desempenho gasto pela GPU na construção do espaço de estados.

5.2.2 Comparação dos resultados obtidos em CPU na construção do espaço de estados dos modelos em diferentes aplicações

Os modelos representados na Tabela 5.2 que foram utilizados para teste e validação da aplicação implementada, também foram usados para se realizar um teste à aplicação fornecida pelo gerador do espaço de estados do servidor IOPT-Tools. Sendo assim foram comparados os resultados obtidos em CPU com a aplicação do servidor (Sem_adapt_codigo) na construção do espaço de estados e com aplicação implementada (com_adaptaçao) para a construção do espaço de estados na GPU. Na Tabela 5.5, encontram se os resultados obtidos em CPU durante a execução de construção do espaço de estados (Gráfico 5.10).

Tabela 5.5: Determinação de tempo gasto na CPU durante a construção do espaço de estados dos modelos.

Modelo	Tempo_CPU_ Sem_adapt_codigo (segundos)	Tempo_CPU_ com_adaptaçao (segundos)
FMMS13_car3	0,000187	0,1639954
Park2in1out	0,000239	0,160745
Park2levels	0,000569	0,282174
Concrete_mixer2_6x	0,00112	0,356158
Concrete_mixer_6xA	0,000299	0,174224
ICIT13_quad_encoder	0,000617	0,2448384
ICIT13_pwm_gen	0,003327	0,851345
Concrete_mixer	0,015248	4,617775

Pode-se ver que os resultados obtidos são muito diferentes uns dos outros, ou seja, o tempo de execução em CPU na aplicação do servidor na construção do espaço de estados demora menos tempo em relação ao tempo de execução em CPU na aplicação implementada na GPU.

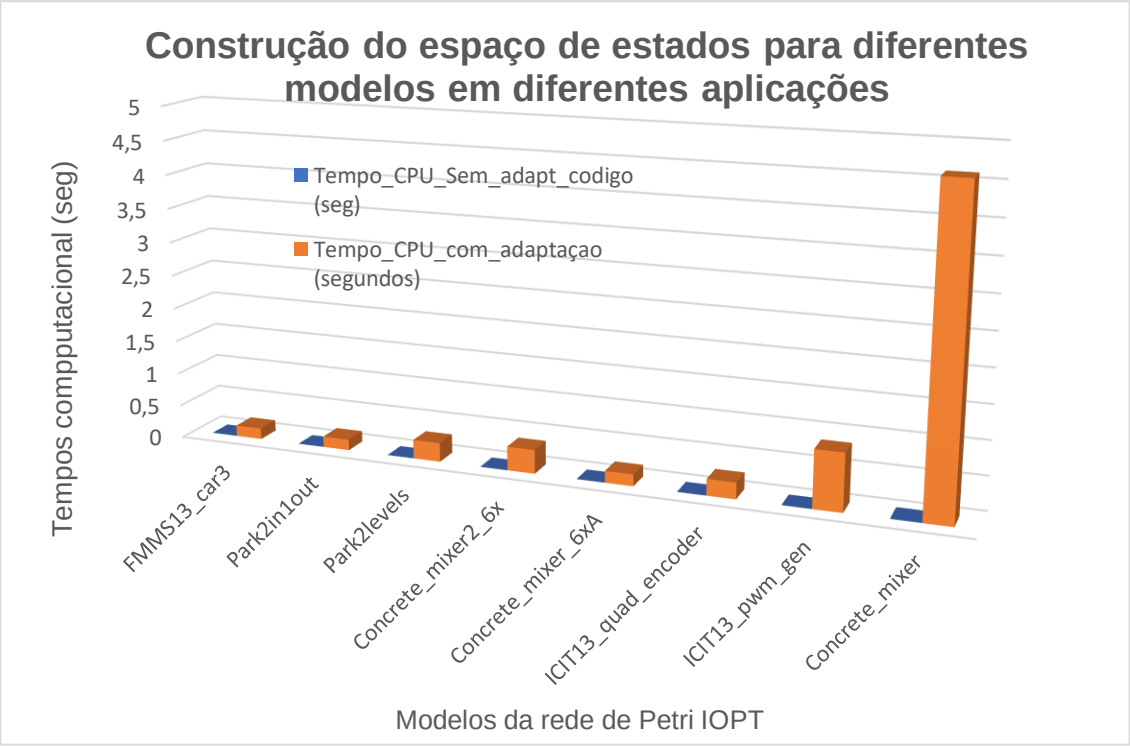


Gráfico 5.10: Tempo de execução da construção de espaço de estados obtido em CPU.

Através do Gráfico 5.10 pode-se ver que o tempo de execução do espaço de estados em CPU é muito superior, porque na aplicação implementada para GPU o tempo é influenciado pela reserva de espaço de memória e pela transferência de dados entre CPU e GPU, tal que a execução depende dos dois processadores. Quanto maior for o número dos ciclos e número dos nós, maior será o tempo de execução em CPU.

6

6 Conclusão

Neste capítulo são apresentados o balanço geral da dissertação desenvolvida e algumas das opções do trabalho futuro.

6.1 Balanço geral

A principal motivação que serviu para dar início ao trabalho desta dissertação foi a ideia de melhorar o tempo na construção do espaço de estados na GPU e aproveitar do algoritmo fornecido pelo gerador de espaço de estados na plataforma IOPT-Tools. A GPU é utilizada como um processador de enorme capacidade de computação adequada para implementação de operações paralelas sobre grandes volumes de dados. Sendo assim, pode se dizer que a GPU é uma ótima proposta para aceleração de dados computacionalmente muito pesado.

Este trabalho serviu para explorar melhor a plataforma NVidia CUDA. A plataforma CUDA facilita a programação em GPU. Com o uso da linguagem de programação C, simplifica o seu uso e aprendizagem de programação. A CUDA permite visibilizar a comunicação entre a CPU e a GPU, designados por *device* e *host.*

A aplicação implementada permite paralelizar e calcular os nós do grafo do espaço de estados de uma rede de Petri IOPT em GPU. Esta aplicação permitiu gerir e partilhar recursos, isto é, no tratamento da memória *device*, na transferência dos dados entre

host e *device*, na técnica de quantificar o número de *blocks* e *threads* por *blocks* a serem usados durante a execução do *kernel*.

Na validação da aplicação os resultados obtidos em GPU e em CPU proporcionaram um tempo razoável. Foi possível identificar o tempo gasto na CPU e na GPU separadamente. O formato da dimensão do grafo do espaço de estados teve a influência sobre o resultado obtido, ou seja, o número do estado não processados em cada ciclo, sendo que é necessário usar um número de *threads* que permita ter o proveito sobre a capacidade de execução paralela da GPU. Os resultados obtidos em CPU foi sempre maior que em GPU. A diferença dos tempos entre a CPU e a GPU foi influenciada pelo tempo de comunicação entre esses dois processadores.

Este trabalho permitiu comparar os resultados obtidos na aplicação implementada com a aplicação fornecida pela IOPT-Tools. Se formos ver por esses resultados não serviu muito utilizar a GPU, isto porque termos de ficar à espera de mais tempos para a construção do espaço de estados. Tendo em conta que a comunicação entre a CPU e a GPU consomem tempo, utilizar estes modelos de baixo nível na GPU, só se justifica utilizar um ambiente deste tipo, quando o tempo de computação da construção do espaço de estados superar o tempo de comunicação e com este modelo o tempo de computação é pequena. Portanto para este modelo de baixo nível, esta estratégia justificar-se-ia se a nossa rede fosse bastante maior para garantir que o volume de processamento (computação) fosse bastante maior comparado com o tempo de comunicação. Para isso, seria necessário a utilização de redes maiores, ou utilizar outras classes de redes de Petri permitindo o processamento de dados (como a classe de RdP Coloridas).

6.2 Trabalhos Futuros

Tendo em conta a conclusão obtida, uma sugestão para o trabalho futuro, seria utilizar modelos da rede de Petri de alto nível para o processamento em GPU. Sabendo que as classes IOPT, só existe condições para teste e depois para gerar uma ou duas marcas, dependendo do peso dos arcos. Mas se tiver uma rede de alto nível, isto é uma rede colorida, onde pode receber uma marca com certos valores e a transição vai provocar transformações nesses valores para os arcos de saída e essas transformações podem ser funções quaisquer que podem ser pesados do ponto de

vista da computação. A utilização da GPU pode justificar as transformações nas marcas. A rede colorida com transformações nas marcas, a arquitetura desenvolvida poderia ser benéfica porque a computação aumenta e a comunicação vai se manter. A outra sugestão, seria usar mais dimensão de *grid* para processar os espaços de estados de forma a que diminuísse mais o tempo na GPU.

7

7 Referências Bibliográficas

[1] I. Ruiz, J. I. Garcia, e C. Collazos, «Modeling in Petri Nets of a Dispersed System in a Physical Rehab 50 REVISTA INGENIERÍA BIOMÉDICA ModelageM eM Redes PetRi de uM sisteMa disPeRso nuMa Reabilitação física».

[2] W. Reisig, *Understanding petri nets : modeling techniques, analysis methods, case studies*. Springer, 2013.

[3] P. C. Yianni, L. C. Neves, D. Rama, e J. D. Andrews, «Accelerating Petri-Net simulations using NVIDIA Graphics Processing Units», *Eur. J. Oper. Res.*, vol. 265, n. 1, pp. 361–371, Fev. 2018.

[4] C. S. Kouzinopoulos, J.-A. M. Assael, T. K. Pyrgiotis, e K. G. Margaritis, «A Hybrid Parallel Implementation of the Aho-Corasick and Wu-Manber Algorithms Using NVIDIA CUDA and MPI Evaluated on a Biological Sequence Database», Jul. 2014.

[5] J. Ghorpade, J. Parande, M. Kulkarni, e A. Bawaskar, «GPGPU Processing in CUDA Architecture», Fev. 2012.

[6] D. A. Zaitsev, «Paradigm of computations on the Petri nets», *Autom. Remote Control*, vol. 75, n. 8, pp. 1369–1383, Ago. 2014.

[7] C. Seatzu, M. Silva, e J. H. van. Schuppen, *Control of discrete-event systems : automata and petri net perspectives*. Springer, 2013.

[8] B. Berthomieu, D. Le Botlan, e S. D. Zilio, «Petri Net Reductions for Counting Markings», Jul. 2018.

[9] E. Badouel, L. Bernardinello, e P. Darondeau, *Petri Net Synthesis*. Berlin,

Heidelberg: Springer Berlin Heidelberg, 2015.

[10] V. Kordic, *Petri net, theory and applications*. InTech, 2008.

[11] Z. Li e MengChu Zhou, *Deadlock Resolution in Automated Manufacturing Systems*. London: Springer London, 2009.

[12] G. Mejia, J. P. Caballero-Villalobos, e C. Montoya, «Petri Nets and Deadlock-Free Scheduling of Open Shop Manufacturing Systems», *IEEE Trans. Syst. Man, Cybern. Syst.*, vol. 48, n. 6, pp. 1017–1028, Jun. 2018.

[13] K. Barylska, E. Erofeev, M. Koutny, Ł. Mikulski, e M. Piątkowski, «Reversing Transitions in Bounded Petri Nets», *Fundam. Informaticae*, vol. 157, n. 4, pp. 341–357, Jan. 2018.

[14] R. Bouyekhf e A. ElMoudni, «On the Analysis of Some Structural Properties of Petri Nets», *IEEE Trans. Syst. Man, Cybern. - Part A Syst. Humans*, vol. 35, n. 6, pp. 784–794, Nov. 2005.

[15] T. Hujsa e R. Devillers, «On Deadlockability, Liveness and Reversibility in Subclasses of Weighted Petri Nets», *Fundam. Informaticae*, vol. 161, n. 4, pp. 383–421, Jul. 2018.

[16] L. F. dos S. Gomes, «Redes de Petri reactivas e hierárquicas - integração de formalismos no projecto de sistemas reactivos de tempo-real», 1997.

[17] J. M. Mendes, P. Leitão, A. W. Colombo, e F. Restivo, «High-level Petri nets for the process description and control in service-oriented manufacturing systems», *Int. J. Prod. Res.*, vol. 50, n. 6, pp. 1650–1665, Mar. 2012.

[18] L. Gomes, F. Moutinho, F. Pereira, J. Ribeiro, A. Costa, e J.-P. Barros, «Extending input-output place-transition Petri nets for distributed controller systems development», em *2014 International Conference on Mechatronics and Control (ICMC)*, 2014, pp. 1099–1104.

[19] L. Gomes, J. P. Barros, A. Costa, e R. Nunes, «The Input-Output Place-Transition Petri Net Class and Associated Tools», em *2007 5th IEEE International Conference on Industrial Informatics*, 2007, pp. 509–514.

[20] F. Pereira, F. Moutinho, e L. Gomes, «IOPT Tools User Manual», Faculdade de Ciências e Tecnologias -FCT, GRES Research Group, Portugal, 2014.

[21] X. Ye, J. Zhou, e X. Song, «On reachability graphs of Petri nets», *Comput. Electr. Eng.*, vol. 29, n. 2, pp. 263–272, Mar. 2003.

[22] F. Pereira, F. Moutinho, L. Gomes, e R. Campos-Rebelo, «IOPT Petri net state

space generation algorithm with maximal-step execution semantics», em *2011 9th IEEE International Conference on Industrial Informatics*, 2011, pp. 789–795.

[23] F. Pereira e L. Gomes, «The IOPT-Flow framework pairing Petri nets and data-flows for embedded controller development», em *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, 2016, pp. 4832–4837.

[24] NVIDIA, «CUDA C PROGRAMMING GUIDE Design Guide version 10.1», 2019.

[25] M. Mawson e A. Revell, «Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs», Set. 2013.

[26] H. Zhou, K. Lange, e M. A. Suchard, «Graphics Processing Units and High-Dimensional Optimization», *Stat. Sci.*, vol. 25, n. 3, pp. 311–324, Ago. 2010.

[27] Y. Cai e S. See, *GPU Computing and Applications*. Singapore: Springer Singapore, 2015.

[28] D. Kirk e W. Hwu, *Programming massively parallel processors : a hands-on approach*. 2016.

[29] F. Pereira, F. Moutinho, L. Gomes, J. Ribeiro, e R. Campos-Rebelo, «An IOPT-net state-space generator tool», em *2011 9th IEEE International Conference on Industrial Informatics*, 2011, pp. 383–389.

[30] J. E. M. Pimenta, «Configurador de sistemas distribuídos de controlo especificados através de redes de Petri IOPT», 2016.

[31] «The World's Most Powerful Graphics Card | NVIDIA TITAN V». [Em linha]. Disponível em: https://www.nvidia.com/en-us/titan/titan-v/. [Acedido: 12-Ago-2019].

[32] «Nvidia Titan V é a nova placa de vídeo para PCs mais poderosa do mundo – Computador». [Em linha]. Disponível em: https://tecnoblog.net/229574/nvidia-gpu-titan-v/. [Acedido: 12-Ago-2019].

[33] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, e J. S. Vetter, «NVIDIA Tensor Core Programmability, Performance & Precision», Mar. 2018.

[34] «Inside Volta: The World's Most Advanced Data Center GPU | NVIDIA Developer Blog». [Em linha]. Disponível em: https://devblogs.nvidia.com/inside-volta/. [Acedido: 12-Ago-2019].

[35] «IOPT Tools». [Em linha]. Disponível em: http://gres.uninova.pt/IOPT-Tools. [Acedido: 13-Ago-2019].

[36] «XSL Transformations (XSLT) Version 2.0». [Em linha]. Disponível em:

https://www.w3.org/TR/xslt20/#what-is-xslt. [Acedido: 21-Set-2019].

[37] F. Pereira, F. Moutinho, e L. Gomes, «A State-Space Based Model-Checking Framework for Embedded System Controllers Specified Using IOPT Petri Nets», 2012, pp. 123–132.

[38] F. Pereira e L. Gomes, «FPGA based speed control of Brushless DC Motors using IOPT Petri Net models», em *2013 IEEE International Conference on Industrial Technology (ICIT)*, 2013, pp. 1011–1016.

[39] R. M. Á. de Lima, «Wave4IOPT - editor e visualizador web de formas de onda – aplicação a controladores digitais especificados com modelos de Redes de Petri IOPT», 2016.