UMA DERIVAÇÃO DO PARADIGMA DE REESCRITA DE MULTICONJUNTOS GAMMA PARA A ARQUITETURA GPU

Rubens Henrique Pailo de Almeida

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Gabriel Antoine Louis Paillard

Rio de Janeiro
Maio de 2015

UMA DERIVAÇÃO DO PARADIGMA DE REESCRITA DE MULTICONJUNTOS GAMMA PARA A ARQUITETURA GPU

Rubens Henrique Pailo de Almeida

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Gabriel Antoine Louis Paillard, D.Sc.

Prof. Cristiana Barbosa Bentes, D.Sc.

Prof. Ricardo Cordeiro de Farias, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
MAIO DE 2015

Almeida, Rubens Henrique Pailo de

Uma Derivação do Paradigma de Reescrita de Multiconjuntos Gamma para a Arquitetura GPU/Rubens Henrique Pailo de Almeida. – Rio de Janeiro: UFRJ/COPPE, 2015.

XV, 147 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França

Gabriel Antoine Louis Paillard

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2015.

Referências Bibliográficas: p. 124 – 131.

1. Gamma. 2. GPU. 3. Computação Heterogênea. 4. Processamento Paralelo. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Dedico este trabalho aos meus pais Gastão e Luci, a minha esposa Mariana, e aos meus irmãos Carlos e Fernanda, que são as pessoas que dão sentido a minha vida.

"Another lesson we should have learned from the recent past is that the development of "richer" or "more powerful" programming languages was a mistake in the sense that these baroque monstrosities, these conglomerations of idiosyncrasies, are really unmanageable, both mechanically and mentally. I see a great future for very systematic and very modest programming languages."
(E. W. Dijkstra, 1972)

"Outra lição que deveríamos ter aprendido com o passado recente é que o desenvolvimento de linguagens de programação "mais ricas"ou "mais poderosas"foi um erro no sentido de que essas monstruosidades barrocas, esses conglomerados de idiossincrasias, são realmente não gerenciáveis, tanto mecanicamente como mentalmente. Eu vejo um grande futuro para linguagens de programação muito sistemáticas e modestas."
(E. W. Dijkstra, 1972)

Agradecimentos

Agradeço primeiramente a Deus pela minha vida e de todos os meus familiares. Obrigado Senhor por toda a proteção que nos dá, e por iluminar o meu caminho permitindo que eu chegasse até aqui.

Ao meu pai, Gastão, e minha mãe, Luci, pelo amor, carinho, e cuidado que sempre me deram de forma incondicional e sem medir esforços. Agradeço pela educação que recebi deles e pelos exemplos de trabalho, dedicação, e honestidade, além do incentivo contínuo ao estudo. Devo tudo o que sou a eles, e me considero privilegiado de tê-los como meu pai e minha mãe.

A minha esposa Mariana, por ser a pessoa maravilhosa que é, pelo amor, companheirismo, compreensão, paciência, e incentivo que me dá em todos os momentos ("Fundamental é mesmo o amor, é impossível ser feliz sozinho").

Ao meu irmão Carlos Rodrigo, e minha irmã, Fernanda, pelo companheirismo, amizade e incentivo.

Ao grande amigo Rui Rodrigues, por ter divido comigo esta longa trajetória durante o mestrado, pelas incontáveis contribuições e conhecimentos compartilhados em todas as disciplinas que cursamos, além do apoio e ajuda durante o desenvolvimento desta dissertação.

Ao Prof. Felipe França, pela oportunidade de ingresso no mestrado, e pelo apoio, acompanhamento e orientação desta dissertação.

Ao Prof. Gabriel Paillard, pela orientação atenciosa e pelas críticas feitas a este trabalho, que muito ajudaram durante o seu desenvolvimento.

Aos professores Claudio Amorim, Valmir Barbosa, Ricardo Farias, Leandro Marzulo, Rosa Leão, Jayme Szwarcfiter, e Myrian Costa, pelos ensinamentos transmitidos nas disciplinas cursadas durante o mestrado.

Aos funcionários do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ pelo suporte e apoio prestados.

Aos amigos da linha de arquitetura e sistemas operacionais, Israel, Edílson, Rafael, e Luneque, pela amizade e troca de conhecimentos proporcionados.

À Marinha do Brasil, por permitir a realização deste curso de mestrado.

Aos comandantes Guilherme Sineiro e José Andrada, pela autorização concedida e incentivo dado à realização deste curso.

Ao Centro Nacional de Processamento de Alto Desempenho, instalado na Universidade Federal do Ceará (CENAPAD-UFC), pela disponibilização do ambiente para a realização dos experimentos deste trabalho.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UMA DERIVAÇÃO DO PARADIGMA DE REESCRITA DE MULTICONJUNTOS GAMMA PARA A ARQUITETURA GPU

Rubens Henrique Pailo de Almeida

Maio/2015

Orientadores: Felipe Maia Galvão França
Gabriel Antoine Louis Paillard

Programa: Engenharia de Sistemas e Computação

Este trabalho apresenta uma implementação do paradigma Gamma de reescrita de multiconjuntos sobre a arquitetura das Unidades Gráficas de Processamento (GPUs). Inspirada em uma metáfora de reações químicas, Gamma foi criada como uma linguagem abstrata de alto nível para especificação de programas de forma simples, concisa, e naturalmente paralela, permitindo uma clara separação entre o problema a ser resolvido e os detalhes de implementação subjacentes. Acreditamos que o modelo de computação proposto por Gamma vai diretamente ao encontro do modo de processamento das GPUs, pois em ambos os casos a premissa básica é processar vários itens de dados paralelamente. Assim, criamos uma implementação de Gamma chamada *Gamma-GPU*, estendendo uma implementação paralela e distribuída já existente. Tal extensão consistiu em adicionar o suporte às GPUs na execução de programas escritos em Gamma, incrementando desta forma a arquitetura da implementação original. Vemos com isso um duplo benefício, pois os códigos em Gamma podem tirar proveito do alto poder computacional das GPUs, e as GPUs podem ser utilizadas de forma transparente pelos programadores Gamma, que não precisam se preocupar com os detalhes de programação das mesmas, bastando expressar seus algoritmos de forma abstrata e natural usando a sintaxe simples fornecida por Gamma. Experimentos práticos foram realizados em um *cluster* de GPUs, através dos quais pudemos constatar a corretude da nova implementação, além de termos obtido bons *speedups* quando comparado com a implementação base sem suporte às GPUs, o que demonstra a evolução introduzida pelo nosso novo modelo.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A GPU-BASED DERIVATION FOR THE GAMMA MULTISET REWRITING PARADIGM

Rubens Henrique Pailo de Almeida

May/2015

Advisors: Felipe Maia Galvão França
Gabriel Antoine Louis Paillard

Department: Systems Engineering and Computer Science

This work presents a GPU-based implementation of the Gamma multiset rewriting paradigm. Inspired by the chemical reaction metaphor, Gamma was conceived as a high-level abstract language for programs specification in a very simple, concise, and naturally parallel way, making clearer the distinction between the problem itself and the underlying implementation issues. We believe that the computational model adopted by Gamma matches perfectly with the Graphics Processing Units execution mode, once both have as basic premise the processing of many data items in parallel. We extended an earlier distributed and parallel implementation of Gamma, adding support to execution of Gamma programs over the GPUs, and we called this new implementation *Gamma-GPU*. Thereby, we see a two-way benefit, with the Gamma programs taking advantage of the GPUs' high computational power, and the GPUs being exposed to the programmers in a transparent manner, who do not need to worry about low-level programming details of the device, and can just express their algorithms in a natural and abstract way through the intuitive sintax provided by Gamma. Practical experiments were conducted in a GPU cluster, which showed us the correctness of the new implementation, besides good speedups comparing to the base non-GPU implementation, proving the contribuitions introduced by our new model.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Há alguns anos temos visto o uso do paralelismo crescer vertiginosamente na área da computação como um todo, impulsionado principalmente pelas limitações encontradas no modelo tradicional de processamento sequencial. As linguagens de programação convencionalmente empregadas nos paradigmas sequencias não fornecem uma clara distinção entre o problema a ser resolvido e os detalhes de controle e coordenação da execução. Com o paralelismo entrando em cena esta característica torna-se ainda mais evidente, pois são utilizadas as mesmas linguagens originalmente sequenciais, além de novas e complexas estruturas para sincronização e controle dos vários fluxos paralelos de execução. O paradigma Gamma surgiu como uma linguagem para especificação de programas dotada de uma sintaxe muito simples, que permite distinguir claramente entre questões do problema e de coordenação e controle, ajudando a gerenciar melhor os detalhes de cada uma, e a expressar as soluções de maneira clara e concisa. Adotando apenas uma única estrutura de dados, chamada de multiconjunto, Gamma estabelece restrições mínimas no modo em que os programas podem acessar os dados, podendo inclusive, fazê-lo simultaneamente, o que dá ao paradigma um potencial naturalmente paralelo de execução.

O uso do formalismo Gamma como uma linguagem real de programação necessita de uma implementação sobre alguma arquitetura de hardware existente. Pela própria natureza concorrente do modelo, as principais candidatas são plataformas paralelas e distribuídas de computação, visando tirar o máximo proveito do comportamento concomitante de execução exibido por Gamma. Recentemente, plataformas da área de computação de alto desempenho (*High-Performance Computing*) vêm utilizando maciçamente arquiteturas compostas por processadores *multicore*, que trabalham em conjunto com aceleradores do tipo *many-core*, a fim de obter o melhor de cada um deles. Originalmente criadas para uso específico na área de computação gráfica, as Unidades Gráficas de Processamento (GPUs) são atualmente um dos tipos mais comuns de aceleradores de processamento, sendo empregadas para resolução de problemas de propósito geral nas mais diversas áreas de aplicação. O

modelo de computação introduzido pelas GPUs consiste basicamente em processar vários itens de dados paralelamente, o que vai diretamente ao encontro da premissa do paradigma Gamma, que opera através da reescrita paralela do multiconjunto. Desta forma, uma implementação que una estes dois conceitos pode trazer um duplo benefício, com Gamma tirando vantagem das GPUs em termos de desempenho, e com o acesso às GPUs sendo provido de forma transparente ao programador, que preocupa-se apenas em expressar seus algoritmos de maneira simples, clara, e concisa, usando Gamma.

1.1 Gamma

O paradigma Gamma foi criado no ano de 1986 por BANÂTRE e LE MÉTAYER [1, 2] como um formalismo para a definição de programas sem artificialidades sequenciais, possibilitando que a atenção fosse voltada apenas para a resolução dos problemas em si, devido a sua sintaxe simples e intuitiva. Aliado a isso, devido à natureza mínima da linguagem e à possibilidade de expressão de algoritmos de maneira muito sistemática, procedimentos de verificação de programas baseados em provas e princípios matemáticos podem ser realizados mais facilmente em Gamma, incorrendo na minimização de erros de programação.

A inspiração para o desenvolvimento de Gamma remete a uma metáfora de reações químicas, no qual a computação é vista como uma coleção de valores atômicos ou moléculas reagindo livremente em uma determinada solução, que é sucessivamente modificada até que nenhuma reação possa mais ocorrer, momento no qual se atinge um estado estável da referida solução química [3]. Esta solução é representada pela única estrutura de dados adotada por Gamma, que é o multiconjunto (*multiset*), o qual se diferencia de um conjunto simples (*set*) pelo fato de poder conter múltiplas instâncias de um mesmo elemento. As reações em um programa Gamma são feitas por pares "ação $\Leftarrow$ condição de reação", e o processamento evolui substituindo no multiconjunto os elementos que satisfazem a condição de reação pelos elementos resultantes da aplicação da ação, com sucessivas reescritas do multiconjunto. Um programa simples em Gamma é composto por apenas uma definição de reação, como no exemplo do código a seguir, que computa o maior elemento presente em um multiconjunto:

$$maior : x, y \rightarrow x \Leftarrow (x \geqslant y)$$

A condição de reação $(x \geqslant y)$ especifica a propriedade a ser satisfeita para os elementos selecionados x e y, e se for cumprida, ocasiona a substituição dos dois elementos por apenas o maior deles, que é o x. É possível perceber que nada é dito

em relação à ordem das combinações de elementos a serem testadas, de modo que se várias delas satisfizerem a condição simultaneamente, as reações podem ocorrer em paralelo. Outro exemplo interessante para demonstrar a ideia da expressão de um programa Gamma é o problema de ordenação de valores. Os elementos do multiconjunto são modelados como pares (índice, valor) e a reação consiste em ordená-los até que atinja-se um estado estável na estrutura, ou seja, todos os pares estejam na sequência correta de acordo com o seu valor:

$$ordena : (i, x), (j, y) \rightarrow (i, y), (j, x) \Leftarrow (i > j)\ e\ (x < y)$$

Notamos através destes exemplos introdutórios que Gamma realmente permite a descrição dos algoritmos de uma maneira muito abstrata, na qual expressa-se apenas a ideia central do problema, sem preocupar-se com idiossincrasias sintáticas de estruturas de controle e de acesso aos dados, como no exemplo de ordenação, que pode ser lido da seguinte forma: "troque todos os valores fora de ordem até que todos estejam ordenados".

Esta noção inicial do paradigma foi bastante explorada e incrementada ao longo dos anos após sua criação, através de extensões linguísticas e sintáticas ao modelo, além de seu uso em aplicações de maior porte, como em algoritmos de processamento de imagens e de grafos. Adicionalmente, várias implementações do formalismo foram propostas sobre diferentes plataformas de processamento, na sua maioria buscando explorar o paralelismo intrínseco fornecido por Gamma. Todos estes aspectos serão discutidos em mais detalhes no Capítulo 2.

1.2 GPU

A Unidade Gráfica de Processamento (GPU - *Graphics Processing Unit*) é um tipo de processador dedicado a computações paralelas sobre grandes quantias de dados. Inicialmente, quando surgiu nos anos de 1980, seu uso era especializado e voltado unicamente para a área de computação gráfica, realizando a renderização e manipulação de objetos gráficos através do processamento paralelo de vértices e *pixels* [4]. Com o passar dos anos, as GPUs sofreram rápidas evoluções, impulsionadas principalmente pelo aumento na capacidade da indústria de semi-condutores em colocar mais transistores em um único *chip*, e pela alta demanda da indústria de jogos e entretenimento, que cada vez mais necessitava de visualizações e representações gráficas super realistas, através da exibição de objetos gráficos complexos e tridimensionais. Ao contrário das unidades de processamento centrais (CPUs - *Central Processing Units*) convencionais, que sempre foram projetadas visando a execução de programas sequenciais, as GPUs tem como premissa principal a execução paralela

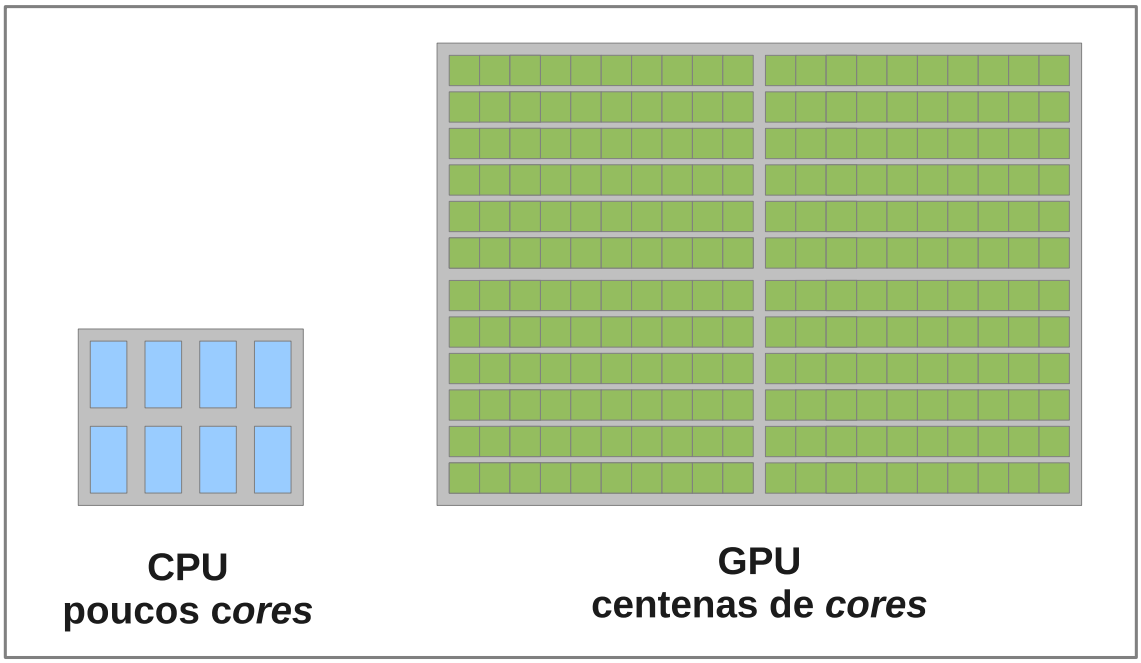


Figura 1.1: Quantia de núcleos de processamento CPU x GPU.

de cálculos aritméticos de forma intensiva, o que leva a uma diferença substancial em suas arquiteturas. Enquanto as CPUs possuem um número reduzido de núcleos (*cores*) para processamento de cálculos matemáticos, as GPUs são dotadas de centenas deles, como podemos notar na Figura 1.1.

Devido a este potencial elevado para a realização de cálculos matemáticos, as GPUs começaram a chamar a atenção de pesquisadores de outras áreas não ligadas à computação gráfica, principalmente dos campos de problemas científicos e de engenharia, os quais demandam geralmente um alto poder computacional para serem resolvidos. Contudo, a programação deste tipo de dispositivo ainda era uma barreira para que as GPUs fossem adotadas como processadores de propósito geral, uma vez que a única forma de acessá-las era através de bibliotecas de programação especializadas em computação gráfica. Visando resolver este impedimento, foi lançada no ano de 2007 pela empresa NVIDIA, uma plataforma completa para programação e utilização das GPUs em problemas de propósito geral, chamada CUDA (*Compute Unified Device Architecture*). Desde então, o emprego das GPUs como processadores de aceleração aritmética vem crescendo consideravelmente nas mais diversas áreas de aplicação, sendo encontradas em boa parte das arquiteturas dos supercomputadores atuais, voltados para solução de problemas que demandam uma alta capacidade de processamento.

1.3 Motivação

Enxergarmos no modelo Gamma uma forma simples e natural de se expressar problemas, sem a necessidade de se preocupar com restrições de sequencialidade presentes em outros paradigmas mais tradicionais, o que fornece um estilo de programação robusto e mais expressivo no tocante ao desenvolvimento de programas paralelos. Devido à natureza mínima da linguagem e sua sintaxe bastante simples, também vemos Gamma como uma linguagem abstrata de alto nível, que pode ser usada como uma linguagem intermediária entre especificações de sistemas e suas implementações propriamente ditas, gerando um mapeamento mais confiável e livre de erros entre estas camadas. Para que Gamma possa ser de fato utilizada como uma linguagem para se expressar algoritmos, é preciso que o formalismo seja implementado sobre plataformas reais de hardware, e é nesse momento que surgem os desafios de lidar com todo o paralelismo exposto pelo modelo, principalmente no tocante ao desempenho da computação realizada. Desta forma, visando obter implementações mais eficientes, é crucial que se utilizem os recursos de processamento mais modernos e mais poderosos existentes atualmente. É neste ponto que entram as GPUs, pois além de serem processadores com elevada capacidade computacional, acreditamos que o modelo de processamento das mesmas vai diretamente ao encontro da metáfora de reações químicas de Gamma, pois em ambos os casos, a premissa principal é realizar a mesma operação sobre uma grande quantia de dados de forma paralela.

A inspiração para este trabalho surgiu de uma implementação de Gamma já existente (será vista no Capítulo 4), a qual possui características de computação distribuída e paralela, para execução sobre plataformas compostas por um ou mais processadores convencionais interligados por uma rede, como por exemplo, um *cluster* de computadores. Assim, podemos dizer que a motivação deste trabalho veio da ideia de prover uma implementação mais atual para o paradigma Gamma, estendendo a implementação base através do mapeamento do modelo para a arquitetura das GPUs, a fim de tirar proveito do poder computacional fornecido por estas unidades gráficas de processamento. Aliado a isto, nos motivou também o fato de podermos prover uma forma transparente para a programação deste tipo de hardware, pois com o uso do nosso novo modelo, o programador pode se beneficiar das GPUs sem necessitar conhecer os detalhes de baixo nível e complexidades das linguagens de programação específicas para as mesmas, expressando seus códigos de forma simples e natural, utilizando a linguagem Gamma.

1.4 Objetivos

Tendo em vista o exposto como motivação do trabalho, podemos identificar os seguintes objetivos para a dissertação:

- Prover uma derivação do paradigma de reescrita de multiconjuntos Gamma para a arquitetura das GPUs, tendo como base uma implementação paralela e distribuída já existente, e com isso:
  - Obter um potencial ganho de desempenho para os programas escritos em Gamma;
  - Fornecer uma abstração para a programação de GPUs.
- Demonstrar a corretude e realizar a análise de desempenho da nova implementação confrontando-a com a implementação base já existente;
- Obter uma implementação eficiente para execução de soluções de problemas reais modelados de forma simples e concisa usando Gamma;
- Fornecer um estudo e uma implementação mais atual relacionados ao paradigma Gamma, de modo a motivar outros pesquisadores a contribuir para a expansão dos estudos ligados ao tema.

1.5 Organização do Documento

O restante do texto desta dissertação está organizado da seguinte forma: nos Capítulos 2 e 3 será realizada a revisão da literatura para os dois conceitos básicos envolvidos neste trabalho. No Capítulo 2 abordaremos o paradigma Gamma, discorrendo sobre seu uso como uma linguagem de programação, sobre algumas de suas extensões linguísticas, bem como a respeito de algumas implementações do formalismo. No Capítulo 3 será feita uma descrição detalhada sobre as GPUs, no qual veremos a arquitetura e o modelo de programação deste tipo de dispositivo, além de sua evolução histórica. No início do capítulo teremos uma seção dedicada para temas relacionados à computação paralela, como sua taxonomia, computação heterogênea, e coprocessadores de aceleração. Os dois capítulos seguintes descrevem as soluções envolvidas no desenvolvimento do trabalho. O Capítulo 4 nos dá uma explicação sobre a implementação paralela e distribuída de Gamma já existente, e que serviu de base para a nova implementação realizada nesta dissertação. Esta nova implementação é o assunto do Capítulo 5, considerado o cerne do trabalho, e que fala detalhadamente sobre a extensão realizada sobre a solução base, a fim de adicionar à arquitetura do modelo poderosos dispositivos de computação paralela,

que são as GPUs. Ainda no Capítulo 5 são citados os trabalhos correlatos a esta dissertação. O Capítulo 6 trata sobre os experimentos práticos realizados com a nova implementação e faz a análise dos resultados obtidos. Finalmente, o Capítulo 7 traz as conclusões e apresenta os trabalhos futuros vislumbrados.

Capítulo 2

Gamma

Neste capítulo, faremos uma abordagem mais completa sobre o formalismo Gamma, descrevendo o paradigma com mais detalhes. Discorreremos sobre seu uso como uma linguagem de programação, e falaremos sobre algumas extensões linguísticas de Gamma, bem como de algumas implementações do formalismo.

2.1 O Paradigma Gamma

O modelo sequencial de computação baseado na arquitetura de *von Neumann* [5] sempre teve um papel preponderante no projeto de muitas linguagens de programação criadas no passado, principalmente por dois motivos [3]:

- Modelos de execução sequenciais forneciam uma forma adequada de abstração para algoritmos, indo ao encontro da percepção intuitiva de um programa definido como uma "receita"para se chegar ao resultado pretendido;
- Implementações eram realizadas sobre arquiteturas com um único processador, refletindo essa visão sequencial abstrata.

O paradigma imperativo de programação é o mais empregado por tais linguagens, que adotam um operador de sequencialidade (`";"`) na descrição de programas. Este operador foi criado pelo fato das linguagens de programação imperativas serem supostamente uma abstração para a arquitetura tradicional de *von Neumann*, fazendo com que os programas contenham instruções sequenciais, que todavia, nem sempre são impostas pela lógica do mesmo, e sim pelo modelo computacional utilizado [6]. Para que se construam programas mais confiáveis e livres de erros, é importante que estes possam ser desenvolvidos seguindo um procedimento sistemático no qual a atenção seja inicialmente voltada ao problema a ser resolvido, e apenas depois volte-se para considerações sobre detalhes de construção da linguagem ou da arquitetura subjacente. Isto é difícil de se atingir com o paradigma imperativo porque

estes dois tipos de sequencialidades, a imposta pela lógica do programa e a imposta pelo modelo computacional, devem ser tratadas conjuntamente em um único passo.

A solução para que se atinja esta derivação sistemática de programas, é que o programador seja capaz de expressar em um primeiro momento, apenas uma versão abstrata do programa em uma linguagem de alto-nível livre de artificialidades sequenciais, a qual em um segundo momento, poderá então ser implementada em vários tipos de arquiteturas, e somente nesse estágio, as imposições de sequencialidades extras serão levadas em conta. As linguagens de programação funcionais representam uma tentativa inicial de um formalismo de alto-nível não-imperativo, contudo, elas fazem uso maciço de recursividade, tanto nas estruturas de dados quanto nos programas, o que acaba se tornando uma forma disfarçada de sequencialidade [2]. Esta dificuldade de separar a expressão do problema a ser resolvido de sua implementação propriamente dita, faz também com que a tarefa de programar paralelamente seja bastante complicada, pois várias linhas de controle sequenciais precisam ser gerenciadas ao mesmo tempo.

É neste contexto que surgiu o Gamma (*General Abstract Model for Multiset mAnipulation*), originalmente proposto no ano de 1986 por BANÂTRE e LE MÉTAYER [1, 2], como um formalismo de alto-nível para especificação de programas de forma elegante e naturalmente paralela, através do uso do operador Gamma (Γ). A proposta era criar uma linguagem muito sistemática e simples, inspirada na disciplina de programação exibida por DIJKSTRA [7]. O modelo computacional descrito pelo operador Γ foi baseado em uma metáfora de reações químicas, na qual os dados são vistos como uma solução química composta por um conjunto de moléculas, e a computação como uma sucessão de reações químicas que ocorrem com as moléculas, caso algumas regras para as reações sejam cumpridas. Tais regras indicam os tipos de moléculas que podem reagir conjuntamente, bem como o resultado produzido pela reação. A ideia é que as moléculas possam interagir livremente, com as reações podendo ocorrer em qualquer ordem e possivelmente ao mesmo tempo, caso as condições de reação apropriadas estejam presentes. A computação termina quando nenhuma reação puder mais ocorrer sobre nenhum subconjunto de moléculas, momento no qual atinge-se uma solução química estável, o que representa o fim do programa.

A estrutura de dados básica no Gamma é o multiconjunto (*multiset*), que consiste na mesma estrutura conhecida como conjunto (*set*), exceto pelo fato de poder conter múltiplas ocorrências do mesmo elemento. Esta estrutura é também usualmente conhecida como *Bag*. O multiconjunto não possui nenhuma restrição no acesso aos seus elementos, nem impõe nenhum tipo de hierarquia entre os mesmos, o que possibilita uma representação ideal para os dados em Gamma, no qual as reações devem ocorrer livremente sem nenhum tipo de restrição de ordem no acesso aos

valores, introduzindo o chamado modelo caótico de execução. Os tipos de dados do multiconjunto podem ser desde tipos simples, como valores inteiros ou decimais, até tipos compostos chamados de tuplas, que consistem em uma coleção de tamanho variável de valores simples. Assim, a computação realizada pelo operador Γ ocorre sobre o multiconjunto (M) através da aplicação de reações que são compostas por pares de funções que representam as regras das mesmas, e são os argumentos do operador Γ:

$$\boxed{\Gamma((C_1, A_1), \ldots, (C_m, A_m))(M)}$$

As funções C_i são as chamadas "condições de reação", e consistem em funções do tipo *booleanas* que indicam os casos nos quais alguns elementos do multiconjunto podem reagir. As funções A_i são chamadas de "ações"e descrevem os resultados das reações que reagiram, realizando a escrita destes no multiconjunto. Desta forma, a computação evolui realizando uma série de transformações no multiconjunto pela aplicação dos pares ($A_i \Leftarrow C_i$), surgindo assim um modelo baseado na **reescrita de multiconjuntos**, que é uma denominação alternativa para referir-se ao paradigma Gamma. Podemos agora introduzir uma definição mais completa e formal para o operador Γ:

$\mathbf{\Gamma}((\mathbf{C_1}, \mathbf{A_1}), \ldots, (\mathbf{C_m}, \mathbf{A_m}))(\mathbf{M}) =$

if	$\forall\ i \in [1, m],\ \forall\ x_1, \ldots, x_n \in M,\ \neg C_i(x_1, \ldots, x_n)$
then	M
else	**let** $x_1, \ldots, x_n \in M,$ **let** $i \in [1, m]$ **such that** $C_i(x_1, \ldots, x_n)$ **in** $\Gamma((C_1, A_1), \ldots, (C_m, A_m))((M - \{x_1, \ldots, x_n\}) + A_i(x_1, \ldots, x_n))$

O efeito de uma reação (C_i, A_i) sobre o multiconjunto é substituir em M o subconjunto de elementos $\{x_1, \ldots, x_n\}$ tal que $C_i(x_1, \ldots, x_n)$ é verdadeira, pelos elementos gerados por $A_i(x_1, \ldots, x_n)$. Se nenhum elemento de M satisfaz nenhuma condição de reação ($\forall\ i \in [1, m],\ \forall\ x_1, \ldots, x_n \in M,\ \neg C_i(x_1, \ldots, x_n)$), então o resultado é o próprio M. Caso contrário, o resultado é obtido aplicando a reação sobre o multiconjunto $((M - \{x_1, \ldots, x_n\}) + A_i(x_1, \ldots, x_n))$, e prosseguindo com o mesmo processo, tendo agora o multiconjunto modificado como entrada para o operador Γ. Esta definição implica que se uma ou várias condições de reação são verdadeiras para vários subconjuntos de elementos ao mesmo tempo, as reações podem ser aplicadas independentemente e simultaneamente, levando a um modelo não determinístico de escolha entre combinações de elementos que podem reagir. Essa é uma das razões básicas dos programas Gamma exibirem um potencial elevado

para o paralelismo [6].

2.2 A Linguagem Gamma

Agora que já apresentamos uma definição formal de Gamma, vamos dedicar esta seção para vermos como este paradigma baseado na metáfora das reações químicas pode ser utilizado como uma linguagem de programação, alterando a forma em que expressamos e pensamos sobre algoritmos.

2.2.1 Estilo de Programação

Para entendermos como é o estilo de programação em Gamma, vamos usar alguns exemplos simples de algoritmos que resolvem problemas bastante comuns. Inicialmente, consideremos um programa que tem como tarefa encontrar o maior elemento em um conjunto de valores:

$$\begin{aligned} maior(M) = \Gamma((C, A))\ (M)\ where \\ C(x, y) = x \geqslant y \\ A(x, y) = x \end{aligned}$$

Nesse caso, os elementos do multiconjunto são selecionados em pares (x, y) e se a condição de reação $(C(x, y) = x \geqslant y)$ for verdadeira, a ação é executada, e consiste em reescrever no multiconjunto apenas o elemento x. Consequentemente o elemento y é removido da estrutura. Se a condição de reação não for satisfeita, nada é feito. O programa termina quando restar apenas um elemento no multiconjunto, que será o maior deles. Perceba que a definição do programa não diz nada a respeito da ordem da seleção de elementos para a realização das comparações, assim, se vários pares disjuntos de elementos satisfazem a condição, as reações podem ser realizadas em paralelo. Nas linguagens de programação mais tradicionais, a primeira decisão a ser tomada para se implementar este algoritmo seria a respeito de qual estrutura de dados usar para armazenar os elementos do conjunto. Nas linguagens imperativas a escolha mais usual seria utilizar um *array*, já nas linguagens declarativas poderia ser uma lista. O programa seria definido como uma iteração pelo *array*, ou por uma caminhada recursiva pela lista. Em qualquer um dos casos, o fato é que a estrutura utilizada imporia restrições na ordem que os elementos são acessados, contrastando com a ideia de Gamma, onde o multiconjunto não possui qualquer tipo de ordem imposta, possibilitando que os elementos atômicos interajam livremente [8].

Vejamos agora como é expresso em Gamma um programa para somar todos os

elementos do multiconjunto:

$$soma(M) = \Gamma((C, A))\ (M)\ where$$
$$C(x, y) = true$$
$$A(x, y) = x + y$$

Os elementos são selecionados em pares (x, y) e o resultado a ser reescrito no multiconjunto é a soma entre os dois ($A(x, y) = x + y$). A condição de reação é sempre verdadeira ($C(x, y) = true$), o que faz com que todas as combinações reajam, até que reste apenas um elemento no multiconjunto, que será o resultado da soma total dos valores inicialmente existentes.

Um programa para encontrar todos os números primos existentes até um certo valor limite N tem o seguinte formato:

$$primos(N) = \Gamma((C, A))\ (2, \ldots, N)\ where$$
$$C(x, y) = multiplo\ (x, y)$$
$$A(x, y) = y$$

O programa funciona eliminando do multiconjunto todos os elementos que são múltiplos de algum outro, para que no final restem apenas os que são números primos. A condição de reação é que x seja múltiplo de y, e a ação é eliminar o x e reescrever o y. Como pudemos perceber nos exemplos dados, os programas escritos em Gamma são muito concisos, e podem ser expressos de forma muito simples e natural, de modo que o programador preocupa-se apenas em modelar a ideia da resolução do problema em si, não sendo necessário envidar esforços com detalhes de mais baixo nível da computação.

Os programas exibidos foram expressos com a notação original proposta pelo paradigma Gamma, a qual é bastante formal. Uma sintaxe alternativa menos formal para a expressão da lógica de programação introduzida por Gamma seria:

replace $(x_1, x_2, ..., x_n)$
such that "texto da condição de reação"
by "texto da ação"

De fato, essa sintaxe por ser mais simples e menos formal, acaba por ser mais adequada quando pensamos em termos de implementação da linguagem. Conforme veremos no Capítulo 4, a sintaxe utilizada pela implementação de Gamma que serviu como base para este trabalho seguiu este caminho.

2.2.2 Padrões de Construção de Reações

A medida que os criadores de Gamma foram ganhando experiência no desenvolvimento de programas com o uso da linguagem, os mesmos constataram que alguns padrões de programação eram recorrentes na expressão das reações, e que estes eram suficientes para construir a maioria das aplicações mais comuns. Assim, foram expostos em [8] cinco esquemas básicos de programação em Gamma chamados de *tropes*, acrônimo para ***T****ransmuter,* ***R****educer,* ***OP****timiser,* ***E****xpander, e* ***S****elector*. Vejamos agora a ideia de cada um deles:

1. *Transmuter*: este esquema realiza a transformação do multiconjunto aplicando a mesma operação sobre todos os elementos individualmente, até que nenhum deles satisfaça a condição de reação. O padrão *transmuter* não altera o tamanho do multiconjunto, apenas transforma os valores dos elementos. Por exemplo, podemos imaginar um multiconjunto composto por tuplas de tamanho dois $[x, y]$, na qual a aplicação do *transmuter* consistiria em trocar o valor de y por $x + 1$ em cada tupla, obtendo $[x, x + 1]$, e consequentemente, transformando todo o multiconjunto até que a condição de reação, por exemplo $C([x, y]) = x > y$, não fosse satisfeita para nenhuma tupla. A definição genérica formal deste esquema é:

$$T(C, f) = x \longrightarrow \ f(x) \Leftarrow C(x)$$

2. *Reducer*: este padrão altera o multiconjunto através da redução se seu tamanho, aplicando reações que atuam sobre pares de elementos e geram como resultado um único valor. O programa de soma dos elementos apresentado na seção anterior é um exemplo da aplicação do padrão *reducer*, pois a cada par de elementos do tipo (x, y) selecionados do multiconjunto, apenas um elemento é reescrito, cujo valor é a soma dos valores dos elementos $(x + y)$. A definição genérica formal deste esquema é:

$$R(C, f) = x, y \longrightarrow \ f(x, y) \Leftarrow C(x, y)$$

3. *Expander*: este padrão altera o multiconjunto através da expansão de seu tamanho, aplicando reações que fazem a decomposição de elementos em uma coleção de valores. Um exemplo da aplicação do padrão *expander* seria uma reação cuja condição é $C(x) = x > 1$, e que a ação correspondente consiste em reescrever para cada x dois valores distintos: $x - 1$ e 1. Um programa como esse serviria para decompor valores positivos n maiores que 1 em n ocorrências de 1. A definição genérica formal deste esquema é:

$$E(C, f_1, f_2) = x \longrightarrow \ f_1(x), f_2(x) \Leftarrow C(x)$$

4. *Selector*: este padrão atua como um filtro, removendo do multiconjunto elementos específicos que satisfazem à condição de reação. Assim como o padrão *reducer*, o *selector* diminui o tamanho do multiconjunto. Um exemplo de aplicação deste padrão é a reação que encontra os números primos exposta na seção anterior, pois ela remove do multiconjunto todos os elementos que cumprem a condição de serem múltiplos de algum outro elemento, e dessa forma, atua como um filtro de valores. A definição genérica formal deste esquema é:

$$S_{i,j}(C) = x_1, ..., x_i \longrightarrow x_j, ..., x_i \Leftarrow C(x_1, ..., x_i) \quad (where\ 1 < j \leqslant (i+1))$$

5. *Optimiser*: este padrão atua otimizando o multiconjunto de acordo com alguma condição em particular, e assim como o padrão *transmuter*, ele não altera o tamanho da estrutura. Este padrão pode ser aplicado em problemas de ordenação de valores e de cálculo de comprimento em cadeias de caracteres [8]. A definição genérica formal deste esquema é:

$$O(<, f_1, f_2, S) = x, y \longrightarrow f_1(x,y), f_2(x,y) \Leftarrow (f_1(x,y), f_2(x,y)) < (x,y) \quad and\ S(x,y)\ and\ S(f_1(x,y), f_2(x,y))$$

2.2.3 Aplicações

Até agora vimos o estilo de programação Gamma através de alguns exemplos de aplicações bastante simples. Contudo, com o passar dos anos, o paradigma foi empregado em diversas classes de programas de maior porte, pertencentes a domínios de problemas de grande interesse. Veremos a seguir alguns destes exemplos.

Em [9], Gamma foi utilizado para modelar uma aplicação de processamento de imagens, a qual tinha como propósito fazer o reconhecimento de uma topografia tridimensional da rede vascular cerebral contida em duas radiografias. Uma versão da mesma aplicação havia sido implementada em outra linguagem antes, mas o autor afirma que ela estava ficando enorme e difícil de gerenciar. Com o uso de Gamma, foi possível se obter um melhor entendimento dos pontos chave da aplicação, além de reduzir o número de erros. Ainda no campo de processamento de imagens, foi exibido em [6] uma aplicação escrita em Gamma para realizar a detecção de bordas de objetos em imagens representadas em escala de cinza. Para cada *pixel*, foi calculado o gradiente em relação a seus vizinhos, e após isso, para descobrir se um *pixel* fazia parte da borda do objeto, aplicou-se uma verificação sobre a intensidade do gradiente relativamente a um determinado limiar. Os elementos do multiconjunto eram tuplas de quatro elementos (P, I, min, max), onde P representa o par de coordenadas do

pixel, I é a intensidade do nível de cinza do *pixel*, e min e max são respectivamente os valores mínimo e máximo entre os gradientes dos vizinhos do *pixel*. Pelos dois casos expostos, Gamma parece ser adequada para representar esta classe de algoritmos, talvez pela razão básica de que muitos dos tratamentos no campo de processamento de imagens são naturalmente expressos como uma coleção de aplicações locais de regras específicas.

Em [6] são exibidos uma série de programas em Gamma para vários domínios de aplicação, como por exemplo, problemas de processamento de *strings*, problemas de grafos, problemas geométricos, problemas de ordenação, e problemas de sincronização de processos. Na categoria de grafos, um dos programas mostrados realiza o cálculo do caminho mais curto (*shortest path*) entre os nós de um grafo dirigido ponderado. As arestas possuem pesos e o comprimento de um caminho consiste na soma destes. Os elementos do multiconjunto são tuplas do tipo (n, m, c), que representam uma aresta entre os nós n e m com seu custo c associado. No final do processamento, o campo c de cada tupla contém o caminho mais curto entre n e m no grafo. O programa foi expresso da seguinte forma:

$$
\begin{aligned}
shortest\_path(G) = \Gamma((C, A))\ (G)\ where \\
C((v_1, v_2, c_{1,2}), (v_2, v_3, c_{2,3}), (v_1, v_3, c_{1,3})) = c_{1,3} > (c_{1,2} + c_{2,3}) \\
A((v_1, v_2, c_{1,2}), (v_2, v_3, c_{2,3}), (v_1, v_3, c_{1,3})) = \\
\{(v_1, v_2, c_{1,2}), (v_2, v_3, c_{2,3}), (v_1, v_3, c_{1,2} + c_{2,3})\}
\end{aligned}
$$

Na categoria de problemas geométricos, é exibida uma aplicação em Gamma que encontra o fecho convexo de um conjunto de pontos em um plano, o qual é definido como o menor polígono convexo contendo todos estes pontos. O multiconjunto inicial contém todos os pontos representados por tuplas (i, j) de suas coordenadas no plano. Para um ponto qualquer ser considerado como vértice do fecho convexo ele não pode estar contido dentro de qualquer triângulo formado por outros três pontos quaisquer do plano. A computação evolui eliminando os pontos que não se enquadram como vértices do fecho, ou seja, que em alguma reação caem dentro (*inside*) de um triângulo. O código pode ser visto a seguir:

$$
\begin{aligned}
convex(Points) = \Gamma((C, A))\ (Points)\ where \\
C((i_1, j_1), (i_2, j_2), (i_3, j_3), (i_4, j_4)) = \\
inside\ ((i_4, j_4)((i_1, j_1), (i_2, j_2), (i_3, j_3))) \\
A((i_1, j_1), (i_2, j_2), (i_3, j_3), (i_4, j_4)) = \{(i_1, j_1), (i_2, j_2), (i_3, j_3)\}
\end{aligned}
$$

O problema dos filósofos famintos foi mostrado na categoria de sincronização de processos, no qual o desafio básico é prover uma correta alocação de recursos, ga-

rantindo que não haja problemas como *deadlocks* ou *starvation* [10]. Neste exemplo, o uso de Gamma ocorreu de uma maneira um pouco diferente, pois o interesse não estava propriamente no resultado final do processamento, e sim nos possíveis valores do multiconjunto durante a computação. Ou seja, o multiconjunto foi visto neste caso como o estado do sistema em um determinado instante. Maiores detalhes desta abordagem podem ser vistos em [6].

Para finalizar os exemplos de aplicações de maior porte escritos em Gamma, não podemos deixar de citar um trabalho contemporâneo a esta dissertação, que pode ser visto em [11]. O referido trabalho dedicou-se a modelar um problema real de acompanhamento de alvos através do uso de técnicas de fusão de dados usando Gamma. Maiores detalhes sobre esta aplicação serão exibidos no Capítulo 6, no qual usaremos o programa de fusão de dados como uma das aplicações para testar a implementação de Gamma desenvolvida nesta dissertação.

2.3 Extensões

Veremos nesta seção algumas extensões linguísticas que foram propostas para Gamma ao longo do tempo, as quais visam de alguma forma incrementar a capacidade de representação sintática e semântica do formalismo.

2.3.1 Operadores de Composição

Os programas Gamma que vimos até então são compostos por apenas uma reação (um par $A \Leftarrow C$). Visando prover uma maior modularidade e uma melhor estruturação para a expressão de alguns problemas, é desejável que haja alguma forma para conectar mais de uma reação, estabelecendo alguma ordem entre as mesmas. Esta composição de reações permite que se obtenham programas mais complexos a partir de outros mais simples, além de permitir a inserção de uma certa restrição no acesso ao multiconjunto, o que em alguns casos pode ser bastante útil.

Diante do exposto, foi proposto em [12] um par de operadores para composição de programas Gamma, e suas propriedades foram estudadas. Considerando duas reações R_1 e R_2, os dois operadores criados foram o de composição sequencial ($R_1 \circ R_2$) e o de composição paralela ($R_1 + R_2$). A semântica imposta pelo operador sequencial é de que a primeira reação R_1 deve executar por completo, para que só depois a segunda reação R_2 possa iniciar. Executar por completo significa realizar todas as possíveis reações no multiconjunto até que nenhuma condição seja avaliada como verdadeira. Neste caso, é como se o multiconjunto estável obtido ao término da execução de R_1 fosse passado como parâmetro de entrada para R_2. Por outro lado, o operador paralelo estabelece que as duas reações podem executar em qualquer

ordem de forma concomitante sobre o multiconjunto, e o processamento termina somente quando nenhuma condição de reação é satisfeita nas duas reações.

Em [13], temos outro trabalho no qual os operadores de composição foram estudados e refinados, possibilitando novas formas de composições semânticas para Gamma.

2.3.2 *High-Order* Gamma

Outra forma de prover uma nova funcionalidade semântica para Gamma é através da inserção da capacidade de escrever programas de alta ordem (*high-order programs*). Esta inspiração vem das linguagens funcionais, as quais implementam a manipulação de programas como se fossem dados comuns. É nesse sentido que em [14] foi proposta uma extensão de alta ordem para Gamma (*high-order Gamma*), na qual a principal contribuição foi a de unificar as duas principais estruturas de Gamma em uma única noção de configuração. Ou seja, no Gamma original os principais componentes são o multiconjunto e os programas em si (expressos pelos pares condição de reação e ação), os quais são encarados separadamente como estruturas independentes. Já na abordagem *high-order*, estes dois conceitos são misturados e encarados como uma "configuração ativa", que podem elas próprias, aparecerem dentro de um multiconjunto.

Em outras palavras, temos a possibilidade de utilizar os programas como elementos do multiconjunto, e isso altera a condição para o término de um programa em Gamma, pois agora, além de não existir nenhum subconjunto de elementos que satisfaçam a condição de reação, é preciso também que não haja nenhuma "configuração ativa"no multiconjunto. Outros trabalhos que também propuseram extensões de alta ordem para Gamma podem ser vistos em [15, 16].

2.3.3 Gamma Estruturada

Como já pudemos perceber, a proposta original de Gamma fundamenta-se basicamente na ideia de processar reações sobre os elementos de uma única estrutura de dados, o multiconjunto, de forma livre e independente, sem impor nenhum tipo de restrição ou ordem para o acesso ao mesmo. Todavia, quando surge a necessidade de modelar uma estrutura de dados específica para a resolução de algum problema, como por exemplo, o uso de uma lista encadeada ou uma árvore, a tarefa do programador Gamma torna-se mais complexa, e o código resultante também acaba ficando complexo e menos legível.

Visando contornar esta situação, foi proposta em [17] uma extensão ao Gamma chamada de Gamma Estruturada (*Structured Gamma*), a qual buscou solucionar o problema da falta de sintaxe para descrever tipos estruturados, sem todavia colocar

em risco as qualidades da linguagem, ou seja, sem impor restrições de ordem para o acesso às novas estruturas, visto que isto iria diretamente contra a premissa básica do formalismo. A maior inovação do referido trabalho foi introduzir o chamado multiconjunto estruturado (*structured multiset*), que se diferencia do multiconjunto tradicional pelo fato dos elementos não serem mais representados diretamente como valores, e sim como um par de endereços e valores associados. Assim, surge uma nova definição de tipo de dados, que pode ser definido pelas regras de uma gramática. Como exemplo, considere a seguinte gramática que define um novo tipo para representação de uma lista encadeada, onde *next* representa uma relação binária e *end* uma relação unária:

$$\begin{aligned} List &= L\ x \\ L\ x &= next\ x\ y,\ L\ y \\ L\ x &= end\ x \end{aligned}$$

Assim, qualquer multiconjunto que possa ser produzido por esta gramática será do tipo *List*. Se quiséssemos representar por exemplo, uma lista de valores $[4, 5, 6]$ nesta nova abordagem, teríamos um conjunto de endereços $\{a_1, a_2, a_3\}$ associados aos valores da seguinte forma: $\overline{a_1} = 4$, $\overline{a_2} = 5$, $\overline{a_3} = 6$, onde a notação $\overline{a_i}$ representa o valor contido no endereço de índice i. E a lista seria expressa assim:

$$next\ a_1\ a_2,\ next\ a_2\ a_3,\ end\ \ a_3$$

Desta forma, em Gamma Estruturada, as reações consistem em testar e modificar as relações entre endereços, e testar e modificar os valores associados a estes endereços, ao passo que as ações são agora descritas como atribuições de valores a determinados endereços. Adicionalmente aos novos tipos suportados, o referido trabalho também abordou a criação de um algoritmo para verificação dos mesmos em tempo de compilação, garantindo que os tipos não se degenerem durante o processamento, ou seja, que o programa mantenha a estrutura subjacente de um certo tipo durante sua execução.

2.4 Implementações

A própria natureza do formalismo Gamma remete a um ambiente concorrente de execução, uma vez que a ideia é que as reações possam ocorrer livremente em qualquer ordem sobre o multiconjunto, podendo inclusive, ocorrer paralelamente. Com isso, a tarefa de programar paralelamente, que usualmente é bastante complexa em

outras linguagens, torna-se fácil e direta através de Gamma, pois sua sintaxe leva a este caminho naturalmente. Contudo, o que devemos deixar bastante claro, é que a proposta original de Gamma consiste em um formalismo teórico, o qual precisa ser mapeado para uma arquitetura real através de uma implementação da linguagem. É neste momento que aparece a difícil tarefa de gerenciar todo o potencial paralelismo oferecido pela linguagem de maneira eficiente, e decisões complexas tem que ser tomadas, buscando a melhor forma de tornar o modelo abstrato em algo concreto. Nesse contexto, as tentativas de implementação do formalismo Gamma podem ser dividas nas seguintes classes [3]:

- **Implementações de Memória Distribuída**: realizadas em redes de computadores (*clusters*), se subdividem em dois tipos:
  - Com Controle Centralizado: os elementos do multiconjunto são distribuídos pelas memórias locais dos processadores, e há um controlador central que monitora toda a troca de informação de forma síncrona;
  - Com Controle Distribuído: a troca de informações entre os processadores é realizada de forma assíncrona e apenas entre unidades vizinhas, sem a presença de um controlador central. O algoritmo de terminação pode se tornar oneroso neste caso, pois o mesmo precisa ser distribuído entre todos os processadores.
- **Implementações de Memória Compartilhada**: o multiconjunto é armazenado como uma estrutura única em apenas uma região de memória, que é compartilhada para leitura e escrita entre os processadores. Multiprocessadores de memória compartilhada e processadores *multicore* são bons candidatos para realizar esta classe de implementação.
- **Implementações em Hardware**: operações básicas sobre o multiconjunto são modeladas diretamente em hardwares configuráveis, como por exemplo, em FPGAs (*Field Programmable Gate Arrays*).

Como exemplo de uma implementação de memória distribuída, temos o trabalho descrito em [18], o qual na realidade realizou duas implementações, uma síncrona com controle centralizado e outra assíncrona com controle distribuído em uma *Connection-Machine* (classe de supercomputador SIMD). Em [19], temos outro trabalho no qual foi realizada uma implementação com controle distribuído em uma *Intel iPSC2 Machine* (arquitetura hipercubo com dezesseis processadores). Uma implementação de memória compartilhada foi descrita em [20], onde foi utilizada uma *Sequent Symmetry S16 MIMD-Machine* (multiprocessador simétrico) com seis processadores. Finalmente, em [21] temos uma implementação em hardware das

operações chamadas de *tropes* (definidas na Subseção 2.2.2), no qual um circuito pode ser produzido a partir de uma descrição de programa.

Em um trabalho mais recente, mostrado em [22], foi introduzida a linguagem HOCL (*Higher-Order Chemical Language*), que consiste em uma implementação de uma extensão de alta ordem para Gamma, conhecida como *Gamma Calculus* (ou γ-*calculus*). Além disso, HOCL ainda adiciona algumas funcionalidades a mais para a sintaxe de *Gamma Calculus*, como suporte a expressões, novos tipos, soluções vazias e esquema de nomes. Foi desenvolvido um compilador na linguagem Java para HOCL, sendo portanto, um ambiente completo para programação em *high-order* Gamma.

Como estamos falando em implementações de Gamma, não podemos deixar de ressaltar que a essência do nosso trabalho consiste em estender uma implementação paralela e distribuída do formalismo (detalhada no Capítulo 4), a fim de fornecer à arquitetura do modelo, suporte à execução de programas Gamma sobre as Unidades Gráficas de Processamento (GPUs), solução que será descrita no Capítulo 5. Além disso, na Subseção 5.6.1, que trata dos trabalhos correlatos a esta dissertação, relacionados a implementações do paradigma Gamma, abordaremos com mais detalhes algumas outras tentativas de se implementar o modelo de execução proposto por Gamma.

Capítulo 3

Unidade de Processamento Gráfico (GPU)

Neste capítulo falaremos sobre as Unidades de Processamento Gráfico (GPUs - *Graphics Processing Units*), abordando temas como sua evolução histórica, arquitetura, e modelo de programação. Porém, antes de nos aprofundarmos nestes tópicos sobre as GPUs, teremos inicialmente uma seção introdutória sobre computação paralela.

3.1 Computação Paralela

Durante muitos anos, seguindo o preconizado pela Lei de *Moore* (*Moore's Law* [23]), o poder computacional dos processadores dobrava em períodos de aproximadamente dezoito a vinte e quatro meses. Contudo, mais recentemente, a computação realizada por um único processador começou a encontrar barreiras na busca pela melhoria de desempenho, entre as quais destacam-se três tipos de limitações (*walls*), que em conjunto foram chamadas de *"Brick Wall"* [24]:

- *Power Wall*: limitação pelo alto consumo energético e por fatores físicos dos *chips* na dissipação de calor;
- *Memory Wall*: limitação pela velocidade de acesso à memória em operações de leitura e escrita; e
- *ILP Wall*: limitação no ganho de desempenho obtido via técnicas de exploração do paralelismo no nível de instruções (*Instruction-Level Parallelism*), como predição de desvios, execução fora de ordem, e execução especulativa[1].

[1] execução especulativa consiste em executar antecipadamente operações que podem ser necessárias mais adiante no programa. Caso se descubra mais tarde que as operações não deviam ter sido executadas, seus resultados são ignorados.

Desta forma, para que fosse possível alcançar novos níveis para a melhoria de desempenho nos sistemas computacionais, surgiu a necessidade da criação de técnicas que permitissem de alguma forma o uso de várias unidades de processamento trabalhando em conjunto e paralelamente, o que levou a um aumento de interesse nas pesquisas da área da computação paralela. A partir do ano de 2005, os principais fabricantes de microprocessadores começaram a oferecê-los dotados de dois *cores* ao invés de apenas um, e nos anos seguintes, o desenvolvimento seguiu com a liberação de CPUs compostas por três, quatro, seis, e até oito núcleos. Este período marcou uma grande guinada no mercado de computação, e ficou conhecido como a "revolução *multicore*"[4].

Ao contrário da tendência *multicore*, que buscava principalmente maximizar a velocidade de execução de programas sequenciais, surgiu no mesmo período uma trajetória alternativa nos projetos de microprocessadores, chamada de *many-thread* ou *many-core*, a qual tinha como foco principal aumentar a eficiência de execução em aplicações paralelas. Tais dispositivos, como por exemplo as GPUs, mostraram-se de fato mais poderosos que as CPUs convencionais para realizar cálculos computacionalmente intensivos normalmente encontrados em aplicações com alto potencial de paralelismo [25]. A partir de então, surgiu um amplo movimento no sentido de unir em um único ambiente estes dois tipos de processadores, utilizando o que cada um tem de melhor em busca do máximo desempenho possível na execução das aplicações, levando a um cenário de computação heterogênea, no qual as CPUs convencionais executam a parte sequencial, e os processadores do tipo *many-core* atuam como coprocessadores de aceleração para partes específicas do processamento.

Nesta seção, faremos uma breve introdução sobre este extenso tema da computação paralela, onde falaremos sobre a taxonomia de arquiteturas paralelas, e sobre a noção de computação heterogênea com o uso de coprocessadores de aceleração.

3.1.1 Taxonomia

Do ponto de vista do software, existem basicamente dois tipos de paralelismo expostos pelas aplicações [26]:

- *Data-Level Parallelism (DLP)*: é o paralelismo no nível de dados, e surge quando há muitos itens de dados que podem ser operados ao mesmo tempo;
- *Task-Level Parallelism (TLP)*: é o paralelismo no nível de tarefas, que ocorre quando há várias tarefas que podem atuar simultaneamente e independentemente sobre cargas de trabalho distintas.

O hardware por sua vez, pode explorar estes dois tipos de paralelismo das aplicações de quatro maneiras principais [26]:

- *Instruction-Level Parallelism (ILP)*: explora o *data-level parallelism* através de técnicas de *pipelines* e execução especulativa;
- *Vector-Processors* e *Graphics Processing Units (GPUs)*: exploram o *data-level parallelism* através da aplicação de uma única instrução sobre uma coleção de dados em paralelo;
- *Thread-Level Parallelism*: explora tanto o *data-level parallelism* quanto o *task-level parallelism* através de um modelo de hardware fortemente acoplado que permite a interação entre *threads* paralelas;
- *Request-Level Parallelism*: explora o *task-level parallelism* em um modelo fortemente desacoplado de hardware que atua sobre tarefas altamente independentes.

Estes modos do hardware suportar os dois tipos de paralelismo das aplicações remete a um estudo bem mais antigo, que se propôs a classificar as arquiteturas dos computadores também em quatro categorias, e ficou amplamente conhecido como a **"Taxonomia de *Flynn*"**, a qual classifica os sistemas de acordo com o número de fluxos de instruções e de fluxos de dados que podem ser gerenciados simultaneamente [27]:

- *Single Instruction, Single Data (SISD)*: é o modelo clássico de *Von Neumann*, no qual uma única instrução opera sobre um único dado, e o paralelismo só é explorado no nível de instruções (ILP) via técnicas como processadores superescalares, *pipeline*, e *superpipeline*;
- *Single Instruction, Multiple Data (SIMD)*: uma única instrução opera sobre múltiplos dados paralelamente, explorando o *data-level parallelism*. Um sistema SIMD pode ser encarado como tendo uma única unidade de controle e várias unidades lógicas aritméticas (ALUs). Como exemplos desta arquitetura temos os processadores vetoriais, as extensões multimídia ao conjunto de instruções[2], e as Unidades Gráficas de Processamento (GPUs);
- *Multiple Instruction, Single Data (MISD)*: múltiplas instruções atuam sobre um único dado. Esta classe apenas completa o esquema proposto, pois não há nenhum processador conhecido deste tipo até o momento;

[2]são instruções adicionais ao conjunto de instruções padrão com o objetivo de melhor atender aos requisitos de processamento vetorial de aplicações gráficas.

- *Multiple Instruction, Multiple Data (MIMD)*: múltiplas instruções atuam sobre múltiplos dados paralelamente. Cada processador busca suas próprias instruções e opera sobre seus próprios dados, explorando assim o *task-level parallelism*. São exemplos desta classe as arquiteturas de memória compartilhada fortemente acopladas que exploram *thread-level parallelism*, e as arquiteturas de memória distribuída fracamente acoladas, como *clusters* e *warehouse-scale computers*, que exploram o *request-level parallelism*.

Muitos processadores paralelos reais são híbridos em relação a esta taxonomia, apresentando características de mais de uma das classes ao mesmo tempo. Um exemplo de particular interesse para o nosso trabalho são as GPUs, as quais nas classificações que vimos até então foram enquadradas como sendo do tipo SIMD, por explorarem o paralelismo no nível de dados (DLP). Contudo, analisando mais a fundo este tipo de hardware, percebemos que na realidade as GPUs consistem em uma mistura dos modelos SIMD e MIMD de computação, pois possuem vários multiprocessadores de *streaming* (*Streaming Multiprocessors*), que individualmente são do tipo SIMD, mas cada um pode estar executando diferentes instruções, levando a um modelo MIMD que explora o paralelismo no nível de tarefas (TLP). Por essa razão, a arquitetura das GPUs é muitas vezes enquadrada em uma nova classificação, conhecida como SIMT (*Single Instruction, Multiple Thread*) [26, 28]. Outra classificação encontrada em [25] é chamada de SPMD (*Single Program, Multiple Data*), e diz respeito ao modelo de programação paralela das GPUs.

3.1.2 Computação Heterogênea

Nosso mundo é heterogêneo por natureza, no qual há uma grande diversidade de entidades que proveem uma riqueza de detalhes difíceis de descrever, e por isso, existe uma vasta gama de elementos que são otimizados para ambientes e tarefas específicas. No contexto da computação, os sistemas heterogêneos também oferecem uma capacidade aumentada aos programadores para selecionar a melhor arquitetura de acordo com uma certa tarefa, ou ainda, escolher a tarefa correta para utilizar de maneira ótima uma determinada arquitetura. Os sistemas heterogêneos são aqueles que, por definição, misturam elementos de diferentes classes arquiteturais com o intuito de trabalhar sinergicamente em prol de um objetivo comum. Nos ambientes computacionais modernos, tais elementos são por exemplo, microprocessadores *multicores*, processadores de sinais digitais, hardwares reconfiguráveis (FPGAs), e processadores de aceleração *many-core* mais recentes como as GPUs. Neste contexto, uma dificuldade que pode aparecer é a falta de ambientes de programação padronizados que sejam capazes de gerenciar o conjunto diverso de recursos de processamento disponíveis [29].

Em [30] são propostos três níveis de heterogeneidade para os sistemas computacionais modernos:

1. Baixo Nível: sistemas compostos por processadores *multicore* (possivelmente organizados como multiprocessadores simétricos - SMPs) ligados em rede, os quais tem acesso a poderosos processadores especializados (ou coprocessadores), como por exemplo, as GPUs;

2. Nível Médio: redes de computadores locais, como por exemplo, em um laboratório de pesquisa. Neste caso, a rede interconecta um grande número de computadores potencialmente heterogêneos entre si em termos de arquitetura de hardware e software, incluindo diferentes tipos de CPUs operando a diferentes frequências, com tamanhos de memórias e estratégias de *cache* distintas, além de sistemas operacionais diferentes;

3. Alto Nível: redes de computadores do tipo WAN (*Wide Area Networks*), com inúmeros computadores geograficamente separados formando um imenso sistema distribuído de alta capacidade computacional, levando a uma arquitetura complexa e largamente heterogênea.

Considerando a classificação proposta, nosso trabalho enquadra-se no primeiro nível de sistema heterogêneo, pois a ideia é fazer com que a execução dos programas escritos em Gamma ocorra sobre um sistema distribuído, onde cada nó possui uma ou mais CPUs (possivelmente *multicore*) compartilhando o processamento com as GPUs disponíveis em seu contexto, levando a um ambiente de computação heterogênea, conforme veremos em detalhes no Capítulo 5.

No primeiro nível de computação heterogênea, a premissa básica é a coexistência de uma CPU convencional *multicore* com um processador mais específico, chamado de coprocessador de aceleração. Estes coprocessadores geralmente possuem uma arquitetura baseada em um número elevado de núcleos (arquiteturas *many-core*), e são utilizados para acelerar o desempenho de determinadas partes das aplicações, geralmente aquelas que possuem um alto grau de paralelismo a ser explorado, expondo grandes quantias de dados (preferencialmente em formato vetorial) e possibilitando um ambiente altamente *multi-threaded*. Surge desta forma, uma arquitetura heterogênea *multicore/many-core*.

Nesse contexto, uma abordagem mais recente é a proposta de unir toda esta heterogeneidade em um único *chip*, nos chamados projetos *SoC (Systems-on-Chip)*, os quais juntam vários componentes especializados pertencentes a arquiteturas distintas de uma maneira compacta e de baixo custo. Como exemplo, temos o processador *Cell Broadband*, desenvolvido em conjunto pelas empresas *Sony*, *Toshiba* e IBM, e que combina em um único *chip*, uma CPU tradicional de apenas um núcleo, com

vários pequenos e simples *cores* de alto desempenho, visando à melhoria das características de consumo energético do sistema. Temos também os *SoCs* que combinam CPU/GPU dispostos de maneira eficiente em um único *chip* voltados para sistemas de alto desempenho, como por exemplo, a arquitetura *Sandy Bridge* da *Intel*, que combina quatro núcleos de CPU *Sandy Bridge* com uma GPU embarcada, e as arquiteturas da AMD (*Advanced Micro Devices*) que dão origem a dispositivos chamados de APUs (*Accelerated Processing Units*), os quais oferecem mecanismos integrados de controle para acesso à memória e ao barramento compartilhado entre os núcleos da CPU e da GPU [29].

3.1.3 Coprocessadores de Aceleração

Como vimos na subseção anterior, os coprocessadores de aceleração são uma nova classe de processadores específicos que usualmente possuem uma arquitetura do tipo *many-core*, e trabalham em conjunto com um processador principal, tipicamente uma CPU *multicore* convencional, formando uma arquitetura heterogênea de baixo nível do tipo *multicore/many-core*. Este arranjo leva a uma computação cooperativa entre as duas classes de dispositivos, na qual as CPUs geralmente ficam responsáveis pelos trechos sequenciais e de maior complexidade em termos de controle de fluxo, enquanto os coprocessadores assumem as partes da aplicação dotadas de maior paralelismo e voltadas principalmente a cálculos aritméticos intensivos. Isto está diretamente ligado à arquitetura destes coprocessadores, onde a predominância é por um grande número de unidades de processamento aritmético de dados (*many-core*) em detrimento de estruturas sofisticadas de controle e memória *cache*, como as encontradas nas CPUs. Além disso, a largura de banda para acesso à memória principal destes dispositivos é geralmente bastante grande, visto que a ideia é que se busquem muitos elementos de dados simultaneamente para que sejam processados em paralelo [25].

Atualmente, existem dois tipos principais de coprocessadores de aceleração que vem sendo mais amplamente utilizados na área de computação de alto desempenho (HPC - *High-Performance Computing*), que são as Unidades Gráficas de Processamento (GPUs) e os processadores fabricados pela *Intel* chamados de *Xeon Phi*. As GPUs são uma classe de processadores originalmente desenvolvidas para processamento de computações gráficas, mas que com o passar do tempo, evoluíram e acabaram sendo empregadas também como coprocessadores de propósito geral, voltados para execução de aplicações altamente paralelas das mais diversas áreas. As principais GPUs atualmente disponíveis são fabricadas pelas empresas NVIDIA e AMD/ATI, e compartilham basicamente as mesmas premissas arquiteturais, possuindo na casa de milhares de núcleos simples e eficientes, com memórias *cache* me-

nores e unidades de controle relativamente mais simples que as CPUs convencionais. No restante deste capítulo as GPUs serão abordadas com mais profundidade, com um enfoque principal baseado nos dispositivos da NVIDIA e no modelo associado de programação das mesmas, chamado CUDA, uma vez que estas foram as tecnologias utilizadas para a implementação das soluções desenvolvidas neste trabalho.

O outro tipo de coprocessador que vem sendo bastante empregado é o chamado *Xeon Phi*, lançado no final de 2012, e desenvolvido sobre a arquitetura MIC (*Many Integrated Core*) da *Intel*, que pode ser encarada como um conjunto de multiprocessadores simétricos (SMPs) dispostos em um único *chip*. O coprocessador *Xeon Phi* sempre trabalha ligado a um processador principal da classe *Intel Xeon* via barramento padrão *PCI Express*, e possui as seguintes características [31]:

- Desenvolvido com a tecnologia de 22 nm dotado de aproximadamente cinco bilhões de transistores;
- Suporta as mesmas ferramentas e linguagens padrão de desenvolvimento dos processadores da família *Intel*;
- Arquitetura *Many-Core*:
  - Mais de 50 *cores*;
  - Cada *core* suporta instruções padrão x86 de 64 bits;
  - Quatro *threads* por hardware em cada *core*, resultando em mais de 200 *threads* simultâneas no coprocessador;
  - Rede de interconexão entre os *cores* de alta velocidade, bidirecional em topologia de anel;
  - Frequências maiores que 1 GHz em cada *core*;
  - Coerência de *cache* em todo o coprocessador.
- Instruções especiais em adição às do padrão x86 64-bits:
  - Instruções SIMD de 512 bits;
  - Suporte de alto desempenho a operações matemáticas como raízes, potências e recíprocos;
  - Capacidades do tipo *Scatter/Gather*[3] para atingir larguras efetivas de banda de memória mais altas.
- Suporte a até 8GB de memória GDDR5;

[3] *Scatter/Gather* são operações paralelas sobre dados onde uma grande quantia de itens são lidos (*gather*) ou escritos (*scatter*) em uma determinada localidade de memória.

O lançamento do coprocessador *Xeon Phi* foi percebido pela comunidade da computação de alto desempenho como uma resposta da *Intel* às já consolidadas GPUs fabricadas pela AMD/ATI e principalmente pela NVIDIA. Apesar destes dois tipos de processadores compartilharem várias características em comum, como o número elevado de *cores* simplificados visando suportar aplicações de cálculos intensivos altamente paralelos, existem algumas diferenças básicas, tanto em termos de suporte a modelos de programação, como em relação aos ganhos efetivos de desempenho alcançados nas aplicações, as quais levam a uma discussão entre os especialistas da área (e principalmente entre os fabricantes) sobre qual dispositivo seria o mais eficiente e o mais facilmente utilizável nas plataformas e programas de HPC. A Tabela 3.1 compara algumas características entre o coprocessador da *Intel Xeon Phi* 5110P [32] e a GPU da NVIDIA *Tesla* K20X [33]. Estes dois dispositivos são representantes significativos dentre os utilizados em plataformas de alta *performance* atualmente.

Tabela 3.1: Características dos coprocessadores de aceleração *Intel Xeon Phi* 5110P e GPU NVIDIA *Tesla* K20X.

Especificação	***Intel Xeon Phi* 5110P**	**GPU NVIDIA *Tesla* K20X**
Número de *Cores*	60	2688
Frequência do *Core*	1.053 GHz	732 MHz
Potência	225 W	235 W
Memória Máxima	8 GB GDDR5	6 GB GDDR5
Largura de Banda da Memória	320 GB/s	250 GB/s
Interface do Barramento	PCI Express	PCI Express
Conjunto de Instruções	x86 64-bits	NVIDIA PTX
Linguagem de Programação	Padrão (C, C++, etc)	CUDA

De um lado, os defensores do coprocessador *Xeon Phi* da *Intel* afirmam que este possui como vantagem sua maior programabilidade e flexibilidade em relação às GPUs, uma vez que suportam o mesmo conjunto de instruções x86 64-bits dos processadores convencionais da família *Intel*, e que linguagens de programação padrão como C e C++ podem ser usadas diretamente para programar o dispositivo. Dizem também que programas que se beneficiam de GPUs irão do mesmo modo se beneficiar com o uso do *Xeon Phi*, mas que o contrário não é verdadeiro, pois algumas aplicações que são suportadas pela alta flexibilidade do coprocessador da *Intel* não seriam passíveis de execução sobre as GPUs [31]. Do outro lado, os entusiastas das

GPUs defendem que não é possível obter ganhos de desempenho aceitáveis com o coprocessador *Xeon Phi* simplesmente recompilando e executando códigos nativos sobre o mesmo. Pelo contrário, afirmam que a programação para otimizar aplicações requer um esforço similar tanto nas GPUs quanto no *Xeon Phi*, e que os resultados obtidos em termos de ganho de desempenho são significativamente melhores nas GPUs [34]. Em [35], [36] e [37], temos exemplos de trabalhos que procederam comparações entre algoritmos implementados em GPUs da NVIDIA e coprocessadores *Xeon Phi* da *Intel*.

A discussão sobre qual dos coprocessadores de aceleração seria o mais eficiente parece não ter um consenso. De fato, podemos perceber que ambos vem sendo amplamente utilizados nos principais projetos de supercomputadores nos dias atuais, conforme exposto na última lista do *website* TOP500 [38], liberada em novembro de 2014, e que reúne os quinhentos supercomputadores mais rápidos do mundo ordenados pelo número de operações de ponto flutuante executados por segundo para o *Linpack Benchmark*. Na referida lista, um total de 75 sistemas computacionais estão usando tecnologias com coprocessadores de aceleração, um acréscimo de 13 em relação aos 62 da lista de novembro de 2013. Destes 75 supercomputadores, temos 49 que utilizam GPUs (46 da NVIDIA e 3 da AMD/ATI) como coprocessadores de aceleração, 21 que utilizam *Xeon Phi*, 4 híbridos com GPUs NVIDIA e *Xeon Phi* trabalhando em conjunto, e apenas um possui um outro tipo de coprocessador, que é o PEZY-SC desenvolvido no Japão. A Figura 3.1 demonstra a fatia que cada classe de coprocessador possui no total dos supercomputadores com esta tecnologia.

Pelo gráfico exposto, fica mais fácil visualizar que as GPUs, compostas majoritariamente pelas da NVIDIA, são maioria entre os coprocessadores de aceleração utilizados nos supercomputadores mais poderosos do mundo, o que nos leva a acreditar que atualmente ainda são a melhor opção para esta finalidade, estando a frente dos coprocessadores *Xeon Phi* da *Intel*.

3.2 Evolução das GPUs

O surgimento da Unidade Gráfica de Processamento, do inglês *Graphics Processing Unit* (GPU), remete ao final da década de 1980 e início da década de 1990, quando começaram a se popularizar os sistemas operacionais com interfaces gráficas, nomeadamente o sistema *Windows* da *Microsoft*, pois os usuários passaram a comprar aceleradores gráficos 2D para seus computadores pessoais, os quais ofereciam operações de *bitmap* via hardware e melhoravam a usabilidade gráfica dos sistemas. Já na metade dos anos 90, a demanda dos consumidores cresceu rapidamente em relação a aplicações gráficas em três dimensões (3D), liderada principalmente pelo mercado de *games*, com o lançamento de jogos imersivos em "primeira-pessoa" como

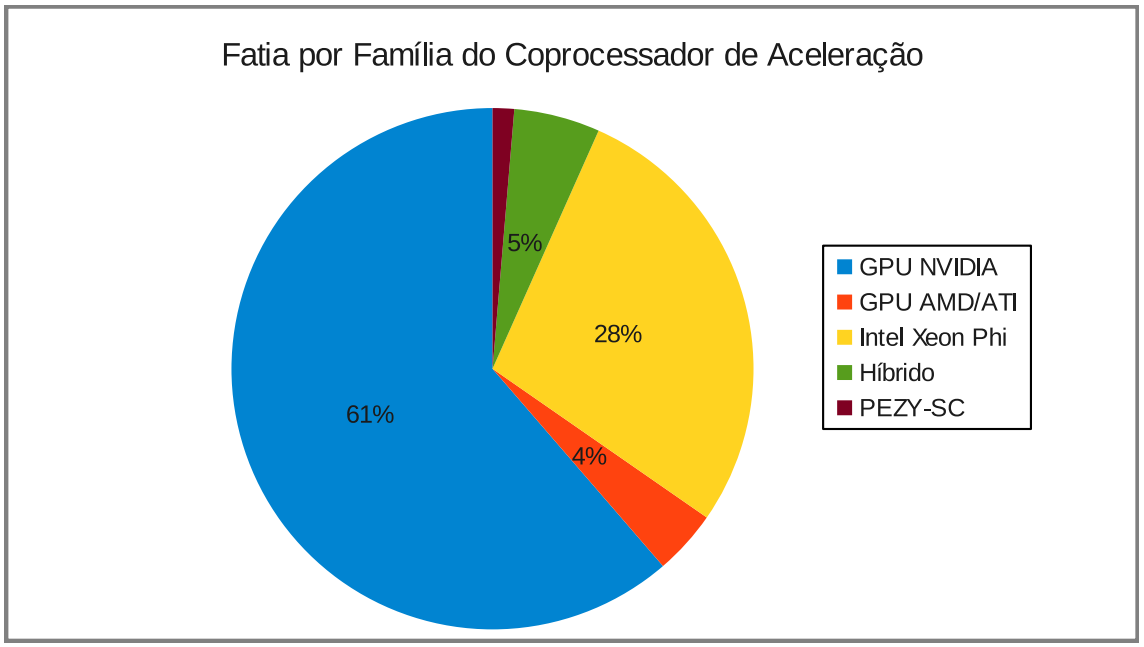


Figura 3.1: Fatia de participação dos coprocessadores de aceleração nos 75 supercomputadores que utilizam tal tecnologia, constantes na lista do TOP500 de novembro de 2014 [38].

Doom, *Duke Nukem* 3D, *Quake*, e *Wolfenstein* 3D. Aliado a isso, empresas como NVIDIA e ATI *Technologies* lançavam no mesmo período aceleradores gráficos a preços acessíveis, o que atraiu grande atenção e fez com que a tecnologia de gráficos 3D obtivesse destaque nos anos seguintes [4].

Nessa época, as GPUs consistiam em um hardware de função fixa que não podia ser programado de forma genérica, e operava em uma sequência de passos realizando operações gráficas sobre os dados, em uma estrutura conhecida como *pipeline* gráfico. O acesso a estas GPUs era feito utilizando-se APIs (*Application Programming Interface*) como a *DirectX* da *Microsoft* e a *OpenGL*, que possibilitavam às aplicações apenas enviar comandos predefinidos às GPUs para que desenhassem objetos na tela do monitor. O trabalho básico do *pipeline* gráfico era desenhar (ou renderizar) triângulos, de modo que a superfície de um objeto fosse representada por uma coleção de triângulos, e para tanto, continha tipicamente os seguintes estágios: controle de vértices (*vertex control*); iluminação, transformação e sombreamento de vértices (*vertex shading, transform and lighting*); configuração do triângulo (*triangle setup*); rasterização (*raster*); sombreamento (*shader*); e refinamento de rasterização (*raster operation ROP*). Maiores detalhes da dinâmica e função dos estágios podem ser encontrados em [25].

Com o passar do tempo, a evolução das GPUs ocorreu via incremento das funcionalidades dos vários estágios do *pipeline* gráfico, através da introdução de novos recursos de hardware e novas configurações. Contudo, a demanda dos desenvol-

vedores foi ficando cada vez mais sofisticada e as novas caraterísticas requisitadas não podiam mais ser atendidas por um hardware com funções fixas, o que levou a evolução a tomar o caminho de tornar programáveis alguns estágios do *pipeline*. Foi então que em 2001, a NVIDIA lançou a família de GPUs chamada *GeForce 3*, sendo a pioneira a disponibilizar um dispositivo em que alguns dos estágios do *pipeline* gráfico, como o de vértices e de sombreamento, podiam ser programados pelos desenvolvedores, tendência seguida nos anos seguintes com a criação de novas funções programáveis, tanto pela própria NVIDIA quanto por outras fabricantes, como ATI e até mesmo pela *Microsoft* com o lançamento de seu console de *games XBox* 360 que permitia os estágios de sombreamento de vértices e *pixels* serem executados em um único processador gráfico [25].

O surgimento da possibilidade de programação do *pipeline* das GPUs atraiu muitos pesquisadores que compartilhavam a ideia de utilizar este tipo de processador para outras finalidades que não somente a renderização gráfica, guiados principalmente pelo fato da alta taxa de cálculos aritméticos que estes dispositivos eram capazes de realizar (pois trabalhavam com milhões de vértices e *pixels*). Entretanto, a única forma de interação com as GPUs ainda era somente as APIs *DirectX* e *OpenGL*, o que complicava a expressão de problemas genéricos, visto que os mesmos deviam ser modelados pelos programadores de forma a parecerem problemas de renderização gráfica a fim de serem executados pelas GPUs. Para contornar este obstáculo, foi lançada no ano de 2006 pela NVIDIA a GPU *GeForce* 8800 GTX, que foi a primeira construída com suporte à nova arquitetura criada pela corporação chamada de CUDA (será detalhada na Seção 3.3), a qual tinha como objetivo expor as GPUs para programação de propósito geral sem impor as restrições das antigas unidades gráficas. Além da criação da arquitetura em si, no início de 2007 a NVIDIA lançou também um compilador para uma nova linguagem de programação baseada em algumas extensões sobre a linguagem C, chamada de CUDA C, a fim de facilitar a exploração da capacidade de processamento das novas GPUs. A partir de então, os programadores não precisaram mais ter nenhum conhecimento sobre as APIs gráficas de programação, nem tampouco necessitavam expressar seus programas "disfarçados" de problemas de computação gráfica [4].

Devido ao início do uso das GPUs em aplicações com outras finalidades que não a computação gráfica, surgiu um novo conceito chamado de GPU *Computing* ou GPGPU (*General Purpose Graphics Processing Unit*), que se concentrou na utilização das GPUs para computações de propósito geral. Até mesmo a placa de hardware em si, algumas vezes passou a ser chamada de GPGPU ao invés de apenas GPU, quando utilizada neste novo contexto de aplicações[4]. Deste momento

[4]Continuaremos utilizando o termo GPU no restante do trabalho, ainda que nosso uso tenha sido para computação de propósito geral.

em diante, o uso das GPUs cresceu vertiginosamente nos mais diversos campos de aplicação, sendo na maioria dos casos utilizadas a tecnologia CUDA e as placas de hardware da NVIDIA, que desde então vem evoluindo e adicionando novas funcionalidades aos seus produtos, liberando novas arquiteturas em períodos de dois em dois anos aproximadamente (ver Subseção 3.3.4). Exemplos de aplicações que se beneficiaram, e ainda se beneficiam do uso das GPUs devido seu alto poder computacional são: modelagem molecular, sistemas financeiros, simulação de reservatórios de óleo e gás, simulação de *n-corpos* (*n-body simulation*), sistemas médicos por imagem, dinâmica dos fluidos, e sistemas de modelagem ambiental [4, 25].

O caminho natural é que as GPUs sejam cada vez mais empregadas para a resolução de muitas classes de problemas, principalmente aqueles que demandam um alto poder computacional. Conforme vimos na Subseção 3.1.3, atualmente as GPUs vem de fato sendo bastante utilizadas em supercomputadores de alto desempenho na função de coprocessadores de aceleração, em ambientes que as utilizam em grande número de forma distribuída e paralela, como por exemplo *clusters* de computadores, nos quais cada nó possui uma ou mais GPUs agregadas, levando a uma estrutura de múltiplas GPUs chamado de *Multi-GPU*. Desta forma, a tendência para o futuro é que as GPUs continuem passando por vigorosas evoluções arquiteturais, com técnicas cada vez mais agressivas sendo introduzidas nas novas gerações, visando aumentar a eficiência de utilização das unidades de cálculo aritmético, permitindo assim que os desenvolvedores prossigam com novas descobertas e implementações de novas otimizações para a resolução de seus problemas [25].

3.3 Arquitetura GPU e CUDA

Veremos nesta seção como as GPUs são organizadas arquiteturalmente e como podem ser utilizadas para a realização dos processamentos massivamente paralelos a que são submetidas. As principais GPUs atuais são fabricadas pela NVIDIA e pela AMD/ATI, e suas premissas arquiteturais e modelo de processamento possuem muitas similaridades. Assim sendo, a fim de tornar nossa abordagem mais racional e de fácil entendimento, utilizaremos como base apenas as tecnologias da NVIDIA durante o texto[5], com o enfoque voltado para a arquitetura (e modelo de programação) criada pela corporação chamada de CUDA (*Compute Unified Device Architecture*), a qual permitiu que as GPUs fossem de fato utilizadas como processadores paralelos de propósito geral. A plataforma CUDA é na realidade tanto uma arquitetura como também um ambiente completo de software, o qual permite aos desenvolvedores programarem as GPUs usando linguagens de alto nível, através de extensões sobre

[5]Para detalhes específicos sobre a arquitetura mais recente da AMD/ATI para GPUs chamada GCN (*Graphics Cores Next*) ver [39].

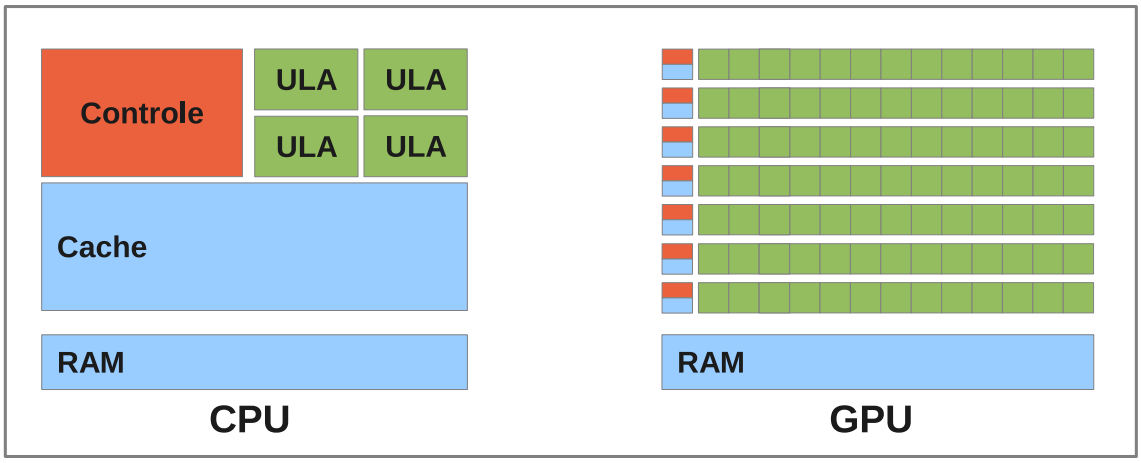


Figura 3.2: Arquitetura básica CPU x GPU (baseado em [40]).

linguagens tradicionais como C, C++, *Fortran*, entre outras [40]. Alguns pequenos exemplos de código, que serão usados durante esta seção, estarão expressos em CUDA C[6], que é a extensão da linguagem C com suporte a CUDA.

Para entendermos a arquitetura básica das GPUs, vamos inicialmente compará-la com a arquitetura das CPUs tradicionais. Na Figura 3.2 podemos notar as diferenças fundamentais na filosofia de projeto dos dois tipos de processadores. A CPU foi projetada originalmente para prover um bom desempenho a aplicações sequenciais, e por isso faz uso de uma unidade de controle sofisticada, que permite a execução paralela no nível de instruções de uma certa *thread*, com o uso de técnicas avançadas como *pipelining*, predição de desvios, e execução fora de ordem e especulativa, mantendo contudo, a aparência de uma execução sequencial. Além disso possui grandes memórias *cache* que visam reduzir a latência no acesso a dados e instruções. Assim, uma área menor do *chip* fica dedicada para as unidades lógicas e aritméticas (ULAs)[7], que são quem de fato realizam os cálculos computacionais [25].

Por outro lado, a GPU foi projetada para realizar computações de cálculos intensivos paralelos (exatamente a proposta original da renderização gráfica), de modo que a maior área do *chip* é dedicada a unidades de processamento aritmético, ao invés de unidade de controle e memória *cache*. A ideia é que um mesmo programa seja executado paralelamente sobre muitos elementos de dados usando um número grande de *threads*, o que requer menos lógica de controle (pois o programa é o mesmo) e também menos memória *cache*, pois a latência de acesso à memória principal pode ser escondida pelos cálculos em si, uma vez que enquanto uma *thread* espera por dados da memória, existem várias outras aptas a serem escalonadas para

[6]Nos referiremos a CUDA C somente como CUDA, pois é a forma mais usual encontrada na literatura.

[7]A figura exibe mais de uma ULA na CPU devido ao fato das mesmas atualmente serem *multicore*.

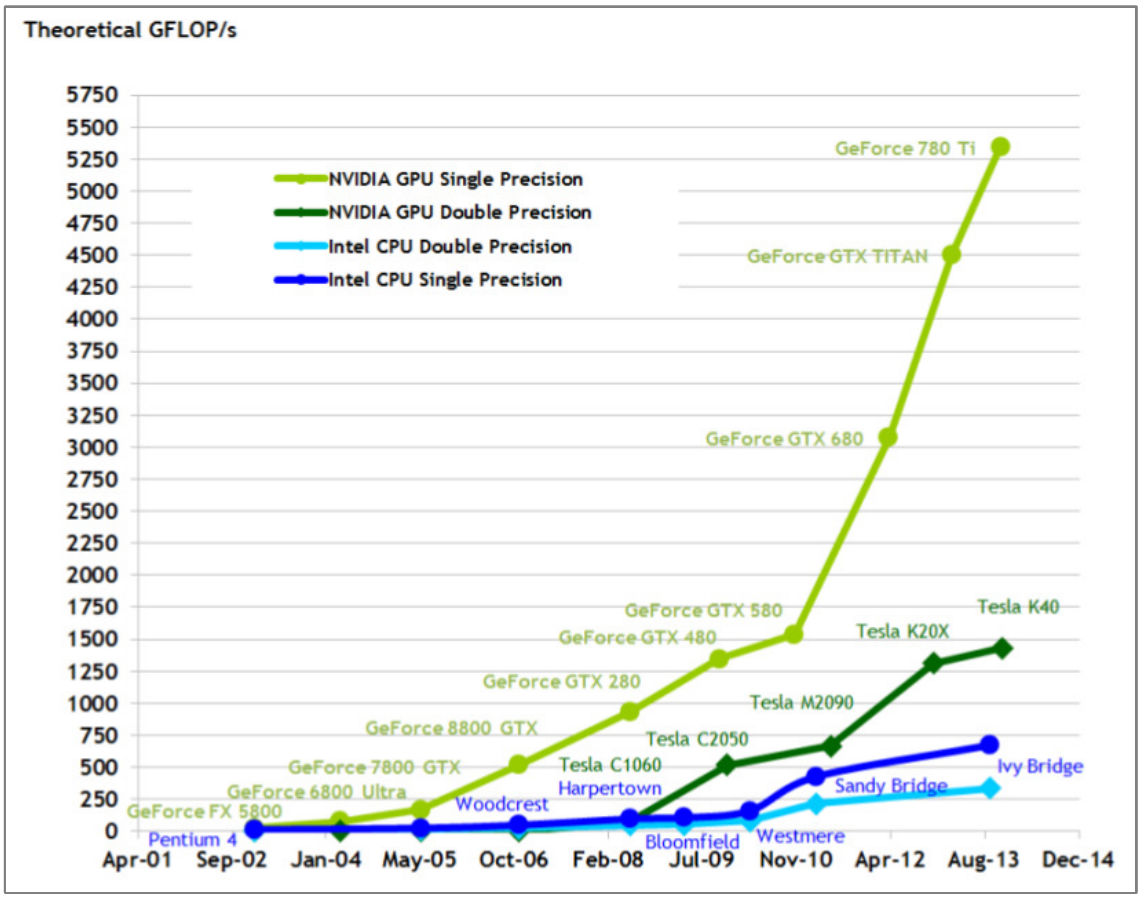


Figura 3.3: Evolução do desempenho CPU x GPU medido em número de operações de ponto flutuante por segundo (GFLOP/s)(retirado de [40]).

computação útil[8]. Outra característica importante e distinta entre os dois processadores é a largura de banda de acesso à memória principal, uma vez que as memórias das GPUs foram projetadas para fornecer uma largura de banda bem maior que as memórias das CPUs tradicionais, podendo chegar a valores superiores a 300 GB/s enquanto a da CPU costuma ser menor que 100 GB/s [28].

A Figura 3.3 demonstra a evolução e a superioridade das GPUs sobre as CPUs no quesito cálculos aritméticos, comprovando a proposta e filosofia da arquitetura de computação intensiva das unidades gráficas de processamento.

3.3.1 Modelo de Processamento

Para entendermos como se dá o processamento nas GPUs, devemos nos lembrar que o uso das mesmas ocorre em ambientes de computação heterogêneos, nos quais temos o processamento controlado por uma CPU convencional que delega trechos das aplicações para serem executadas paralelamente pelas GPUs. Dito isso, vejamos primeiramente qual é a estrutura de um programa escrito em CUDA, que reflete a

[8]computação de cálculos aritméticos relacionados ao problema que está sendo processado.

coexistência de uma CPU (*host*), e de uma ou mais GPUs (*devices*) no computador. Um arquivo fonte em CUDA é formado por uma mistura de dois tipos de código: o *host-code*, que executa na CPU, e o *device-code* que executa na GPU. Para que seja possível identificar quais são os trechos que devem executar no *device*, existem palavras reservadas especiais para demarcar o código, que consistem justamente na extensão que CUDA provê sobre a linguagem C. Com estas marcações, pode-se declarar as estruturas de dados bem como as funções que executarão na GPU, as quais geralmente são aquelas que exibem um grande paralelismo de dados.

Na Figura 3.4 podemos ver o processo de compilação de um programa escrito em CUDA. Como o código fonte não consiste mais em apenas linguagem C, e sim possui extensões especiais, é preciso que se utilize um novo compilador capaz de interpretar estas alterações sintáticas. Este é o NVCC (*NVIDIA's CUDA Compiler*), criado pela NVIDIA e liberado juntamente com a nova linguagem a fim de prover um ambiente completo de desenvolvimento. O NVCC processa o programa escrito em CUDA usando as palavras reservadas especiais para separar o *host-code* e o *device-code*. O *host-code* é delegado pelo NVCC ao compilador padrão do *host* para linguagem C, que o compila para gerar a parte do executável relativa à CPU. O *device-code* é transformado primeiramente para a linguagem de montagem (*assembly*) expressada pelo conjunto de instruções padrão das GPUs da NVIDIA chamado PTX (*Parallel Thread eXecution*)[9], e depois é compilado por um componente do *runtime* do NVCC (compilador *just-in-time*), gerando a parte do executável relativa à GPU. Ao final do processo, temos como saída um arquivo executável hibrido CPU/GPU que deve ser executado em uma plataforma heterogênea de computação.

A parte da computação designada para a execução nas GPUs é aquela com maior potencial de paralelismo, de modo que a aplicação possa se beneficiar do alto número de *threads* que podem ser criadas para processamento simultâneo. Para serem capazes de executar esta grande quantia de *threads* de maneira concorrente e eficiente, as GPUs são construídas sobre um conjunto de processadores *multithreaded* chamados de Multiprocessadores de *Streaming* (SM - *Streaming Multiprocessors*), que empregam uma arquitetura conhecida como SIMT (*Single Instruction, Multiple Thread*). No escopo de uma única *thread*, visando maior desempenho, é empregada a técnica de *pipeline* (similar a das CPUs) para prover paralelismo no nível de instruções. Contudo, técnicas mais agressivas normalmente encontradas nas CPUs convencionais, como execução fora de ordem, predição de desvios, e execução especulativa de instruções, não são suportadas, pois isto tornaria a unidade de controle demasiadamente complexa, ocupando uma maior área no *chip* do processador e automaticamente diminuindo o espaço para unidades de cálculos aritméticos, que são a prioridade nas GPUs [40].

[9]Para maiores informações sobre o conjunto de instruções PTX da NVIDIA ver [41].

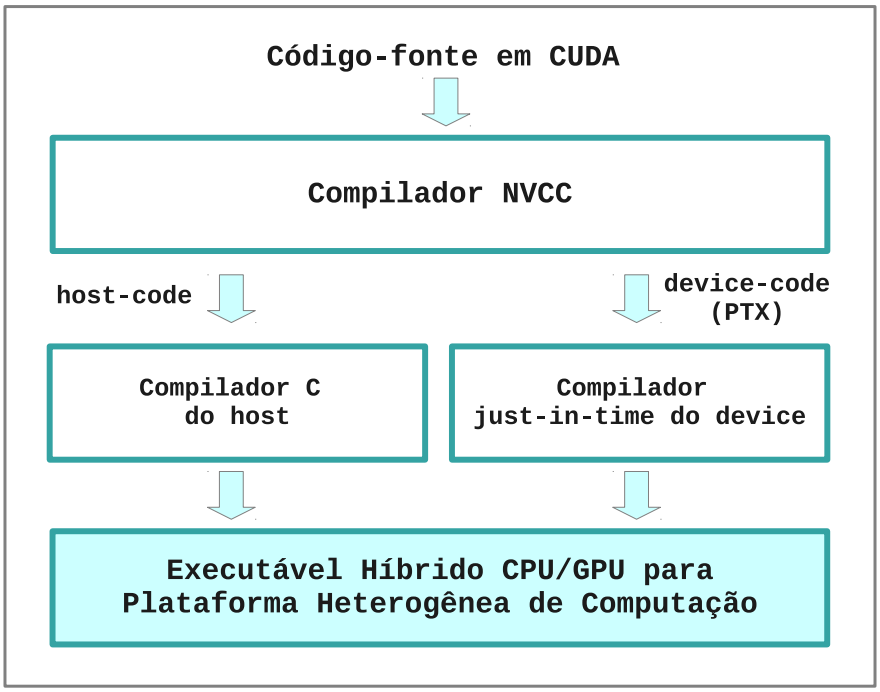


Figura 3.4: Processo de compilação de um programa CUDA (baseado em [25]).

A arquitetura SIMT dos multiprocessadores está relacionada com o modelo SIMD (*Single Instruction, Multiple Data*) de execução paralela, no qual uma única instrução opera sobre múltiplos dados. O que ocorre na SIMT, é que uma mesma instrução é aplicada em várias *threads* paralelas, que são criadas, gerenciadas, e escalonadas pelo multiprocessador em grupos de 32 *threads*, chamados de *warps*[10]. No contexto de um *warp*, cada *thread* inicia no mesmo endereço de programa, e o passo de execução é controlado por uma única unidade de controle para todo o grupo, de modo que apenas uma instrução é executada a cada instante. Todavia, é permitido que cada *thread* tome individualmente desvios no programa, executando de forma independente, gerando o efeito conhecido como divergência de caminhos. A maior eficiência é atingida quando todas as *threads* de um *warp* tomam o mesmo caminho, pois todas estarão executando a mesma instrução. Quando há divergência, devido a desvios condicionais, o *warp* executa serialmente cada caminho tomado, desabilitando as *threads* que não estão no caminho ativo. Uma vez que todos os caminhos se encerrem, as *threads* convergem novamente para o mesmo fluxo de execução. Vale ressaltar que divergências só ocorrem dentro de um mesmo *warp*, *threads* de *warps* distintos executam de forma totalmente independente umas das outras.

Para se ter uma ideia dos componentes que tipicamente compõem um *Streaming Multiprocessor* (SM), consideremos o adotado pela arquitetura de GPUs da NVIDIA chamada *Kepler GK110* [42], na qual os multiprocessadores passaram a ser chamados de SMX (*Streaming Multiprocessor eXtreme*) devido a evoluções implementadas.

[10]O termo *half-warp* para um conjunto de 16 *threads* também é comumente empregado.

Na *Kepler GK110*, cada GPU pode conter até quinze SMX e seis controladores de memória de 64 bits. Cada SMX é formado por: 192 *CUDA-Cores* (núcleos de precisão simples); 64 unidades de cálculo de precisão dupla (*Double-Precision Units*); 32 unidades de funções especiais (*Special Function Units*); 32 unidades de leitura/escrita (*Load/Store Units*); 4 escalonadores de *warps*; 8 unidades de despacho de instruções (*Instruction Dispatch Units*); memórias *cache* de dados, de instruções, de constantes, e de textura.

Do ponto de vista do hardware do multiprocessador, já vimos que as *threads* criadas para execução na GPU são agrupadas em *warps*. Voltemos agora a um nível um pouco mais alto, para ver como são criadas as várias *threads* e como são organizadas hierarquicamente quando o processamento é passado para a GPU em um programa CUDA. As funções que devem ser computadas pela GPU são chamadas de *kernels*, e especificam o código a ser executado por todas as *threads* em paralelo, que são criadas no momento em que o *kernel* é invocado. A definição de um *kernel* é feita com o uso da palavra reservada `__global__`, que é uma das extensões de CUDA sobre a linguagem C. Uma função qualificada como `__global__` indica que a mesma deve ser executada no *device* e pode ser chamada a partir do *host*. Existem outras duas palavras reservadas para qualificar funções em CUDA, que são `__host__` e `__device__`, sumarizadas na Tabela 3.2.

Tabela 3.2: Palavras reservadas em CUDA para qualificação de funções.

Assinatura da Função	**Executa no:**	**Chamada a partir do:**
`__device__ float DeviceFunc ()`	`device`	`device`
`__global__ void KernelFunc ()`	`device`	`host ou device`
`__host__ float HostFunc ()`	`host`	`host`

Para invocar um *kernel* a partir do código no *host* é necessário que se utilize uma nova sintaxe `<<<...>>>` de configuração de execução, que especifica o número de *threads* que serão criadas. Em seguida, aparecem os parâmetros convencionais entre parênteses, como na linguagem C:

```
//Invocação do Kernel com N threads
kernelFunc<<< 1, N >>>(a,b,...);
```

Cada *thread* que executa em um *kernel* recebe um identificador único, chamado de índice da *thread*, o qual pode ser acessado dentro do *kernel* através de uma variável automaticamente preenchida e disponibilizada pelo ambiente CUDA, chamada `threadIdx`. Desta forma, cada *thread* pode usar o seu índice para acessar posições específicas da estrutura de dados durante o processamento. Esta abordagem é bastante usual nos programas em CUDA, visto que os dados a serem

processados estão geralmente dispostos em formato de *array*, e o paralelismo é obtido com cada *thread* operando sobre porções distintas do mesmo. Como exemplo, vejamos o trecho de código a seguir que faz a adição de dois *arrays* `A` e `B` de tamanho `N`, e armazena o resultado no *array* `C`:

```
//definição do Kernel
__global__ void SomaArray(float*A, float*B, float*C){
     int i = threadIdx.x;
     C[i] = A[i] + B[i];
}

int main(){
     ...
     //Invocação do Kernel com N threads
     SomaArray<<< 1, N >>>(A, B, C);
     ...
}
```

Neste caso, cada *thread* realiza apenas uma operação de adição cujos operandos são posições dos *arrays* `A` e `B` de acordo com seu índice. No exemplo em questão, o número de *threads* a serem criadas foi especificado na chamada do *kernel* usando-se a variável `N`, que é um valor simples do tipo inteiro. Assim, os índices das *threads* dentro do *kernel* são criados sequencialmente, de `0` a `N-1`, levando a uma estrutura unidimensional de índices acessíveis pelo campo `'x'` da variável `threadIdx`. Em muitos casos, é conveniente que a organização das *threads* e de seus índices seja feita de maneira hierárquica e multidimensional, para prover uma forma natural de realizar computações sobre estruturas de dados também multidimensionais, como matrizes e volumes [40]. Por isso, CUDA fornece uma maneira de organizar as *threads* em uma hierarquia de dois níveis, composta por ***grid*** e **blocos**. Um *grid* consiste em um ou mais blocos, e cada bloco consiste em uma ou mais *threads*. Assim como as *threads* dentro de um bloco, os blocos dentro de um *grid* também recebem índices, que podem ser acessados através da variável `blockIdx`. O número de blocos de um *grid* e o número de *threads* de um bloco são as dimensões dos mesmos, e podem ser acessados dentro dos *kernels* pelas variáveis `gridDim` e `blockDim` respectivamente.

Tanto o *grid* como os blocos podem ter suas dimensões divididas de maneira tridimensional, através do uso de um novo tipo de variável fornecido pelo ambiente

CUDA, chamado de `dim3`, que consiste em uma *struct* da linguagem C com três campos do tipo *unsigned integer*: `x`, `y`, e `z`, onde cada campo representa uma dimensão. O programador pode escolher quantas dimensões quer usar, bastando que os campos das dimensões não utilizadas recebam o valor 1. Como já vimos, nos casos onde deseja-se usar apenas uma dimensão, é oferecido ao programador a possibilidade de usar diretamente um valor do tipo inteiro, ao invés de uma variável `dim3`. Nesse caso, o valor inteiro, que pode ser proveniente de uma expressão aritmética, é atribuído ao campo `"x"`, e os campos `"y"` e `"z"` são automaticamente preenchidos com o valor 1. No nosso exemplo anterior, havíamos passado na chamada ao *kernel* dois valores inteiros, `"1"` e `"N"`, para especificar o número de *threads* que queríamos criar. Nesse caso, o que ocorreu na realidade é que foi criado um *grid* composto de apenas um bloco, o qual por sua vez, contém as `N` *threads* requeridas. Agora, a organização exata do *grid* pode ser melhor especificada na chamada ao *kernel*, através do uso do operador de configuração de execução `<<<...>>>` recebendo dois parâmetros do tipo `dim3`. O primeiro especifica a dimensão do *grid* em número de blocos, e o segundo especifica a dimensão do bloco em número de *threads*.

Para exemplificar, vejamos como especificamos uma chamada de *kernel* com um *grid* bidimensional de três colunas por duas linhas de blocos (3x2), totalizando seis blocos, e com cada bloco composto também de forma bidimensional, com quatro colunas por três linhas de *threads* (4x3), totalizando doze *threads* por bloco, e consequentemente, 72 *threads* no *grid*. O resultado desta configuração pode ser visualizado na Figura 3.5, e o código para criá-la seria:

```
dim3 BlocksPerGrid(3,2);
dim3 ThreadsPerBlock(4,3);
kernelFunc<<< BlocksPerGrid, ThreadsPerBlock >>>(...);
```

Supondo que fossemos processar uma matriz bidimensional de dimensões 6x12 utilizando essa configuração, com cada *thread* acessando apenas uma posição da mesma via especificação de linha e coluna, teríamos que calcular os pares de índices (i, j) únicos para cada *thread* dentro do *kernel*, o que poderia ser feito da seguinte forma:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
//Faz algum processamento baseado na posição calculada
matriz[i][j] = "operação qualquer";
```

Considere por exemplo, baseado na Figura 3.5, a *Thread* (2,1) pertencente ao Bloco (1,1). Pelas dimensões da matriz suposta, e pela organização dos blocos de *threads*, para que possamos cobrir a matriz de forma completa, a *Thread* (2,1) do

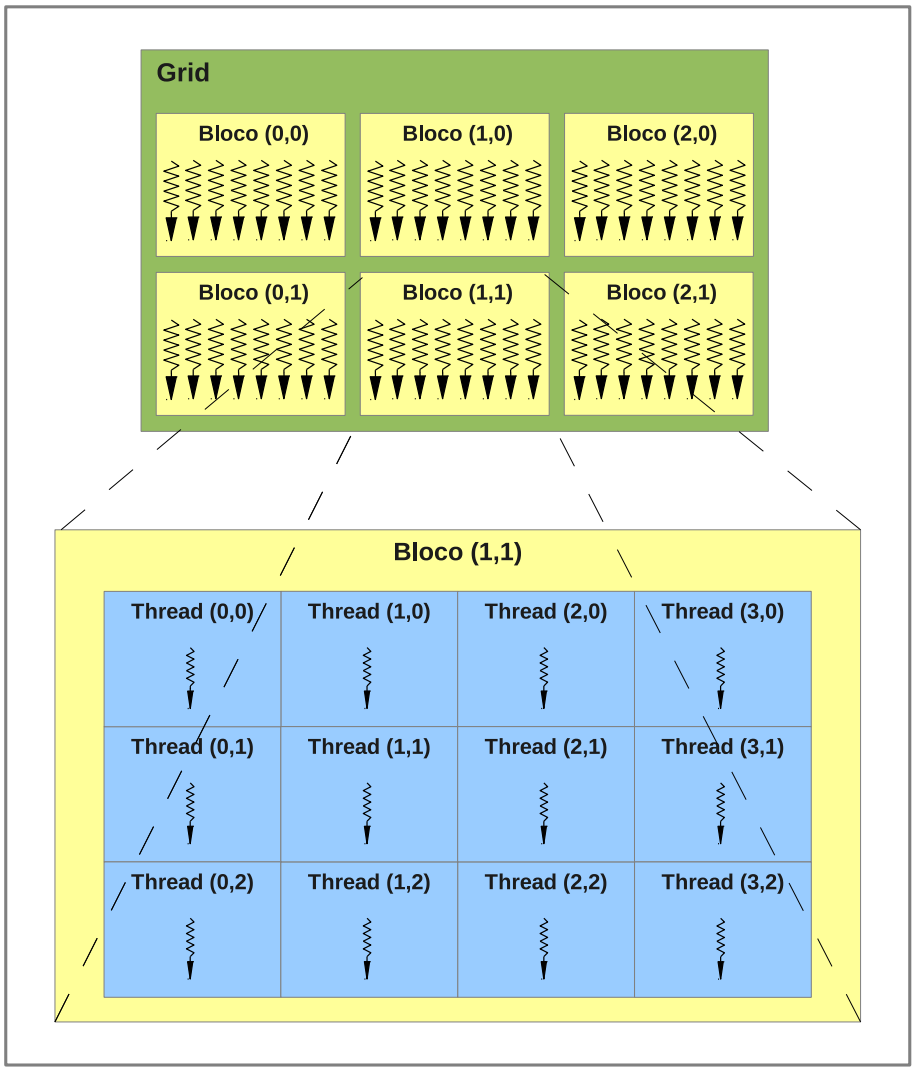


Figura 3.5: Organização hierárquica das *threads* em *Grid* e Blocos (baseado em [40]).

Bloco (1,1) deve naturalmente processar o elemento de índices (i=6, j=4) da matriz, lembrando que os índices começam do zero. Então, usando o código mostrado para o cálculo dos índice temos:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
 ↓
int i = 1 * 4 + 2;
int j = 1 * 3 + 1;
 ↓
int i = 6;
int j = 4;
```

Da mesma forma, outra aplicação que possua uma configuração hierárquica qualquer, pode realizar os cálculos dos índices para as *threads* no *kernel*, baseado sempre

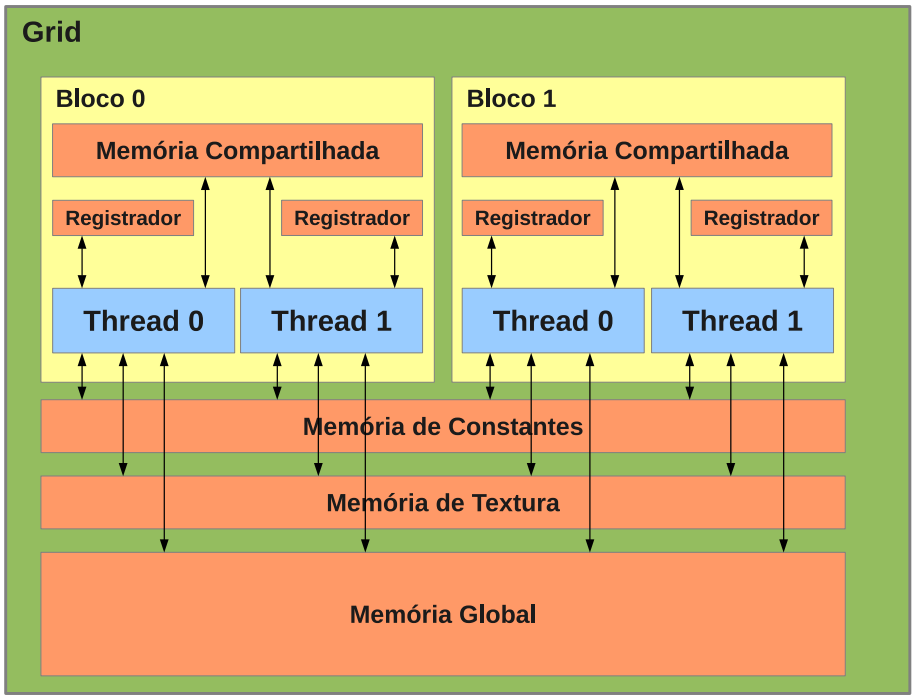


Figura 3.6: Organização da estrutura de memória no ambiente CUDA. (baseado em [25])

nas variáveis de dimensão e de índice dos blocos dentro do *grid*, e das *threads* dentro dos blocos, tornando o modelo de estruturação provido pelo ambiente CUDA bastante flexível para atuar sobre diversos formatos de dados [25].

3.3.2 Estrutura de Memória

No contexto de computação de CUDA, o *host* e o *device* possuem espaços separados de memória, de modo que cada um possui sua própria memória principal. Estas memórias são geralmente do tipo DRAM (*Dynamic Random Access Memory*), e nas GPUs são chamadas de memória global ou memória do *device* [25].

As *threads* criadas quando um *kernel* é invocado possuem a sua disposição uma estrutura de memória composta por vários espaços distintos, que variam em tamanho e latência de acesso. Esta organização de memória no ambiente CUDA pode ser visualizada na Figura 3.6. Cada *thread* possui individualmente sua própria memória privada, que são os registradores (*register memory*), e que não podem ser acessados por nenhuma outra *thread*. Cada bloco de *threads* tem acesso a uma memória compartilhada (*shared memory*), a qual é visível apenas para as *threads* pertencentes ao bloco. Finalmente, todas as *threads* independentemente de qual bloco sejam, possuem acesso a três espaços de memória acessíveis por qualquer uma delas, que são: a memória global (*global memory*), a memória de constantes (*constant memory*), e a memória de textura (*texture memory*).

As memórias do tipo registrador e compartilhada podem ser acessadas com ve-

locidades muito altas, mas possuem um tamanho reduzido. Já a memória global possui um tamanho bem maior, mas uma latência mais alta de acesso. As memórias de constantes e de textura são do tipo somente leitura, o que provê uma latência menor se comparadas com a memória global, contudo, são de menor capacidade de armazenamento. As memórias global, de constantes, e de textura, são persistentes entre chamadas de diferentes *kernels* em uma mesma aplicação, e são otimizadas para situações distintas. A memória de textura, por exemplo, oferece modos extra de endereçamento e filtragem de dados para alguns formatos específicos de dados [40]. A Tabela 3.3 mostra como as variáveis podem ser qualificadas com palavras reservadas de CUDA para especificar qual espaço de memória utilizar. Para o caso da memória de texturas, por ser um tipo mais complexo, alguns campos extras são requeridos.

Tabela 3.3: Palavras reservadas em CUDA para qualificação de variáveis.

Declaração da Variável	**Memória**	**Escopo**
variáveis escalares	registrador	thread
__device__ __shared__ int sharedVar	compartilhada	bloco
__device__ __constant__ int constVar	constantes	grid
__device__ int globalVar	global	grid
texture <DataType, Type, ReadMode> texVar	textura	grid

Para realizar a alocação de memória no *device* e a cópia de dados entre *host* e *device*, existem chamadas fornecidas pela API do CUDA *Runtime*, como podemos notar no seguinte trecho de código, que estende o nosso exemplo anterior de soma de *arrays* de tamanho N, adicionando os trechos de manipulação de memória que ocorrem pré e pós invocação do *kernel*:

```
int main(){
     float *A, *B, *C;
     float *dA, *dB, *dC;
     int size = N * sizeof(float);
     //Aloca arrays A, B e C no host e lê valores de A e B
     ... chamadas linguagem C padrão ...
     //Aloca memórias no device
     cudaMalloc((void **) &dA, size);
     cudaMalloc((void **) &dB, size);
     cudaMalloc((void **) &dC, size);
     //Copia os arrays A e B do host para dA e dB no device
```

```
    cudaMemcpy(dA, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dB, B, size, cudaMemcpyHostToDevice);
    //Invocação do Kernel com N threads
    SomaArray<<< 1, N >>>(dA, dB, dC);
    //Copia o array resultado dC do device para o C no host
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    //Libera memórias no device
    cudaFree(dA); cudaFree(dB); cudaFree(dC);
}
```

A sequência de operações exposta é a forma padrão normalmente utilizada para qualquer programa em CUDA, formada por: alocação de memória; cópia de memória *HostToDevice*; invocação do *kernel*; cópia de memória *DeviceToHost*; e liberação de memória.

3.3.3 Programação GPU

Já vimos que a corporação NVIDIA foi a pioneira na introdução de um ambiente completo para programação de propósito geral sobre as GPUs, através da criação do CUDA no ano de 2007. Além de ser uma arquitetura que serve como base para o modelo de computação das GPUs, CUDA é também uma linguagem de programação que permite utilizar tais dispositivos em um ambiente heterogêneo composto por CPUs e GPUs. A linguagem CUDA é na realidade uma extensão sobre linguagens de programação tradicionais, já bastante consolidadas e de ampla aceitação no mercado de desenvolvimento de aplicações, como C, C++, Java, *Fortran*, entre outras. Originalmente, CUDA estendeu a linguagem C, sendo denominada como CUDA-C, e ainda é a escolha mais usual atualmente nas aplicações desenvolvidas em CUDA. Nas seções anteriores, já abordamos alguns aspectos principais do uso da linguagem CUDA, suficientes para criar um núcleo básico para diversas aplicações. Um aprofundamento sobre a sintaxe e sobre as funções providas pela API do CUDA podem ser encontradas em [40], onde são cobertos tópicos como *streams*, eventos, sincronização, funções *multi-device*, etc.

Depois de CUDA, e a medida que o uso das GPUs foi se popularizando para aplicações de propósito geral, surgiram outras linguagens para programação deste tipo de dispositivo, as quais geralmente compartilham muitas semelhanças com CUDA, até mesmo pelo fato de muitas delas terem se inspirado no modelo CUDA de programação. Entre tais linguagens, algumas das principais são [25]:

- *OpenCL*: a linguagem *OpenCL* (*Open Computing Language*) teve sua criação iniciada pela *Apple* e é atualmente gerenciada por um consórcio de tecno-

logia não-lucrativo chamado *Khronos Group*. *OpenCL* permite desenvolver aplicações sobre uma vasta gama de dispositivos de diferentes fabricantes. Ela suporta vários níveis de paralelismo em sistemas homogêneos ou heterogêneos, compostos por um ou vários dispositivos como CPUs x86, ARM, e *PowerPC*, além de GPUs da AMD/ATI e da NVIDIA. Assim como CUDA, a linguagem *OpenCL* é uma extensão sobre a linguagem C, com suporte opcional a C++, além de possuir adaptadores para várias outras linguagens, como Java, *Python* e .NET. O modelo de programação é muito similar a CUDA, pois a linguagem inspirou-se em CUDA e "emprestou" muitos dos conceitos originalmente propostos pela plataforma da NVIDIA [29].

- *OpenACC*: foi incialmente desenvolvida pelo *Portland Group* (PGI), *Cray Inc.*, e NVIDIA. A API do *OpenACC* (*Open Accelerators*) provê um conjunto de diretivas de compilação, rotinas de bibliotecas, e variáveis de ambiente, que podem ser usadas para escrever programas paralelos em C, C++, e *Fortran*, os quais executam em dispositivos aceleradores, como as GPUs. Assim como as linguagens anteriores, ela é uma extensão às linguagens tradicionais. A maior diferença em comparação com CUDA é a possibilidade do uso de diretivas de compilação, as quais permitem paralelizar um código sequencial de maneira bastante simples, bastando anotar os trechos que devem ser paralelizados com as diretivas desejadas. Contudo, uma desvantagem é o fato de que o desempenho de um programa em *OpenACC* é muito dependente das capacidades do compilador usado, que devem ser capazes de se beneficiar totalmente das diretivas no código. Isso não ocorre com CUDA, pois o paralelismo é escrito de forma mais explícita, tendo uma dependência menor do compilador [25].

- C++ AMP: o modelo de programação C++ AMP (*Accelerated Massive Parallelism*) foi inicialmente desenvolvido pela *Microsoft*, e atualmente é uma especificação aberta que recebe contribuições de múltiplas fontes, notavelmente da AMD e da NVIDIA. A linguagem é utilizada para escrever algoritmos paralelos que visam explorar ambientes heterogêneos de computação, com a presença de GPUs. Trata-se de uma extensão de C++, e compartilha muitos dos conceitos introduzidos pelo CUDA, com a diferença de poder ser portada para várias plataformas de hardware [25].

3.3.4 Arquiteturas NVIDIA

Para finalizarmos este capítulo sobre as GPUs, vejamos quais foram as arquiteturas já lançadas, e quais estão por vir, pela pioneira na área de processamento de propósito geral com o uso de GPUs, a corporação NVIDIA. A Tabela 3.4 sumariza

Tabela 3.4: *Roadmap* das arquiteturas de GPUs da NVIDIA [43].

Arquitetura	**Ano de Lançamento**	**Principal Novidade**
Tesla	2007	CUDA
Fermi	2010	FP64
Kepler	2012	Dynamic Parallelism Hyper-Q GPU Direct
Maxwell	2014	DX12
Pascal	2016	3D-Stacked RAM NVLink Mixed Precision
Volta	2018	?

o *roadmap* (mapa de evolução) das arquiteturas, mostrando os anos de lançamento (ou previsão de lançamento), e as principais novidades introduzidas por cada uma delas. As informações exibidas na tabela são baseadas na palestra de abertura do cofundador e CEO (*Chief Executive Officer*) da NVIDIA Jen-Hsun Huang durante a conferência "GPU *Technology Conference*" organizada pela corporação no ano de 2015 [43].

A primeira arquitetura de GPUs da NVIDIA com suporte total a CUDA foi chamada de *Tesla*, e teve seu lançamento em 2007. Depois, em 2010, foi lançada a arquitetura *Fermi*, cuja principal novidade introduzida foi a grande evolução no desempenho de operações de ponto flutuante de dupla precisão (FP64). Em 2012, a arquitetura *Kepler* [42] trouxe uma série de inovações em relação às suas antecessoras, entre as quais destacam-se: o Paralelismo Dinâmico (permite à GPU lançar *kernels* por si mesma); *Hyper-Q* (permite vários *cores* de CPU invocar *kernels* na GPU simultaneamente); e GPU *Direct* (transferência direta entre memórias de GPUs). Mais recentemente, no ano de 2014, foi lançada a arquitetura *Maxwell*, que é a mais moderna atualmente. Sua principal novidade foi o suporte à API *DirectX* 12 da *Microsoft* (DX12), além de uma melhoria no *tradeoff "performance X consumo energético"*, aumentando o desempenho por *watt* consumido [44]. Finalmente, foi anunciado para 2016 o lançamento da nova arquitetura da NVIDIA chamada de *Pascal*, que trará como principais inovações as seguintes funcionalidades: 3D-*Stacked* RAM (aumento de até 3 vezes na largura de banda de acesso a memória); *NVLink* (novo barramento até 5 vezes mais rápido que o PCI *Express*); e *Mixed Precision* (taxas de execução distintas para operações com tipos de dados de diferentes precisões). Apesar de não haver detalhes, apareceu durante a conferência o nome da arquitetura que deve substituir a *Pascal* em 2018, chamada de *Volta*.

Capítulo 4

Gamma Paralela e Distribuída (Solução Base)

Agora que já realizamos uma revisão geral sobre os dois principais conceitos que compõem a ideia central deste trabalho, ou seja, o modelo computacional Gamma e as Unidades de Processamento Gráfico (GPUs), dedicaremos os dois próximos capítulos para descrever as soluções envolvidas no desenvolvimento da dissertação. No presente capítulo, faremos uma explicação sobre a implementação de Gamma já existente, que foi utilizada como solução base para o trabalho, a qual chamaremos a partir de agora de *Gamma-Base*. No Capítulo 5, mostraremos a solução que foi criada e implementada para estender a solução base, provendo suporte ao processamento na arquitetura das GPUs, chamada de *Gamma-GPU*.

4.1 Visão Geral

A ideia de desenvolver uma nova implementação do paradigma Gamma, que fosse capaz de executar de maneira paralela sobre o hardware das GPUs, teve como inspiração uma implementação paralela e distribuída de Gamma, realizada por Juarez Muylaert e Simon Gay, e estendida por Gabriel Paillard, no ano de 1999 [45, 46]. A extensão realizada pelo último, focou principalmente na criação e verificação de novos tipos de dados, não suportados originalmente pelo Gamma, tornando-se na realidade, uma implementação de Gamma Estruturada (definida na Seção 2.3.3). Contudo, essa nova abordagem estruturada não influenciou no modelo de execução paralelo e distribuído que já existia na implementação do Gamma original, pois a mudança afetou somente os tipos de dados suportados e adicionou a capacidade de verificação e não-degeneração das novas estruturas do programa em tempo de compilação. Assim sendo, a implementação desenvolvida nesta dissertação, descrita no Capítulo 5, foi feita sobre a implementação Gamma original (*Gamma-Base*), e

não sobre a implementação de Gamma Estruturada, uma vez que os esforços foram movidos da ideia de suporte e verificação a tipos de dados, para o suporte e ampliação do paralelismo utilizado intrinsecamente pelo modelo, através do uso das GPUs. A integração do suporte a tipos de dados estruturados presentes em Gamma Estruturada com a nova implementação utilizando GPUs ficará como um trabalho futuro.

Mais do que servir apenas como motivação e inspiração para o trabalho, a implementação paralela e distribuída a qual estamos nos referenciando, foi de fato utilizada como o alicerce para a nova implementação, desde a adoção dos mesmos moldes arquiteturais, até a utilização do código-fonte em si. Em outras palavras, a implementação *Gamma-Base* foi utilizada como base de desenvolvimento, e teve sua capacidade estendida no que diz respeito ao uso e exploração da concorrência e do paralelismo dos programas Gamma, alcançado com a utilização das Unidades Gráficas de Processamento. Esse fato nos faz realçar a importância desta implementação base, pois todas as suas características e particularidades, sejam elas pontos positivos ou negativos, têm impacto direto sobre a implementação *Gamma-GPU*, como poderemos verificar mais detalhadamente no Capítulo 6, que trata dos experimentos e análise dos resultados. Nesse contexto, as demais seções deste capítulo serão dedicadas para que possamos obter um melhor entendimento sobre a implementação *Gamma-Base*, nas quais falaremos sobre sua arquitetura, o compilador criado, e alguns detalhes do ambiente de execução.

4.2 Arquitetura Distribuída

A propriedade do formalismo Gamma de poder ser interpretado como uma linguagem de programação com natural potencial de concorrência, levou a várias tentativas de implementações do modelo, em diferentes tipos de plataformas paralelas. A implementação *Gamma-Base* foi uma dessas investidas, na qual buscou-se explorar o paralelismo dos programas Gamma sobre uma arquitetura distribuída de processamento. A ideia principal foi a de distribuir o processamento do multiconjunto para vários núcleos processantes, mas tendo as informações monitoradas e a dinâmica da execução regida por um elemento controlador central. Assim, podemos classificar a implementação *Gamma-Base* como sendo de memória-distribuída com controle centralizado [3], conforme já vimos na Seção 2.4.

A comunicação entre os elementos de processamento foi realizada através do uso de troca de mensagens, com a adoção da interface de troca de mensagens MPI (*Message Passing Interface*). Dado um programa Gamma qualquer, são criados processos MPI para executar o trabalho de cada célula de processamento, que podem ser de quatro tipos:

- ***Bag-Cell***: é a célula que gerencia o envio e recebimento do multiconjunto (*Bag*) para as demais células que executarão as reações, servindo portanto como a regente do escalonamento, pois uma determinada célula que desejar receber o *Bag* para processar, deve primeiro requisitá-lo à *Bag-Cell*, que o enviará caso não esteja sendo processado por nenhuma outra célula, e depois de processá-lo, a célula deve devolvê-lo à *Bag-Cell*, para que outras células tenham a oportunidade de trabalhar, evitando a preterição indefinida de alguma delas, o que causaria o efeito conhecido como inanição (*starvation* [10]). Com isso, fica garantida também a exclusão mútua para acesso às variáveis do multiconjunto, evitando a modificação simultânea por duas reações diferentes, o que poderia causar inconsistências durante a execução;

- **Controladora Principal (CP)**: é a célula que controla a execução das demais células, enviando mensagens que sinalizam que há trabalho a ser feito. Além disso, realiza também o controle de término do programa, que ocorre quando nenhuma condição de reação é satisfeita nas reações que compõe o programa;

- **Trabalhadora (T)**: é o tipo de célula responsável por executar uma reação sobre o multiconjunto, ou seja, testar as condições de reação e, caso satisfeitas, executar as ações correspondentes;

- **Controladora (C)**: é o tipo de célula que atua controlando a execução de outras duas células, que podem ser do tipo trabalhadora ou controladora. Esta classe de células existe para programas Gamma compostos por mais de uma reação, que são conectadas pelos operadores de composição paralelo ou sequencial.

Seja qual for o programa Gamma submetido para execução na implementação *Gamma-Base*, haverá sempre uma, e somente uma, célula do tipo *Bag-Cell*. O mesmo ocorre para a célula Controladora Principal. Já as células dos tipos Controladora e Trabalhadora, são criadas em quantidade variável, de acordo com o número de reações existentes no programa executado. A arquitetura distribuída que descrevemos pode ser visualizada na Figura 4.1.

4.3 Compilador Gamma

Para que uma implementação do formalismo Gamma, encarada como linguagem de programação, seja concretizada, é necessário que se construa uma ferramenta que realize a transformação de um arquivo de entrada escrito em linguagem Gamma, em um arquivo que seja passível de execução sobre alguma plataforma de hardware

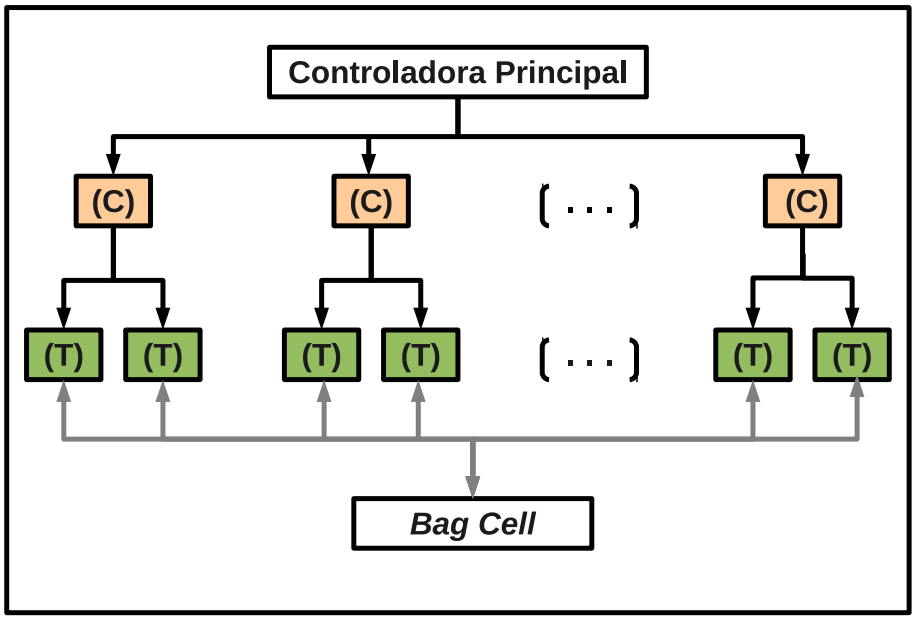


Figura 4.1: Arquitetura Distribuída da implementação *Gamma-Base*.

existente. Na implementação *Gamma-Base*, optou-se por se desenvolver um compilador de programas Gamma, o qual recebe como entrada um arquivo de código-fonte escrito em Gamma (*.gm*), realiza todas as fases de um compilador típico sobre este código, e gera como saída um arquivo escrito em linguagem C contendo chamadas em MPI. Esse arquivo é então submetido a um compilador C padrão, para que se gere um binário executável compatível com um ambiente distribuído composto de múltiplos processadores[1]. Nas subseções a seguir, daremos mais detalhes sobre o compilador criado, abordando as fases de compilação e a sintaxe suportada.

4.3.1 Análise Léxica e Sintática

Em programas que possuem representações escritas, seguindo alguma forma de estruturação, duas tarefas que são sempre executadas consistem em dividir a entrada em elementos que possuam algum significado, e depois descobrir o relacionamento entre eles. Em programas Gamma, tais elementos seriam por exemplo, os valores constantes, os nomes de variáveis, as palavras reservadas, os operadores, os símbolos delimitadores de tuplas e do multiconjunto, entre outros. A análise léxica é a fase responsável por realizar esta divisão da entrada em unidades discretas (conhecidas como lexemas ou *tokens*), e isto é feito pelo analisador léxico, também chamado de *scanner*. Já a tarefa de descobrir os relacionamentos entre os lexemas, como por exemplo, expressões, declarações, atribuições, entre outros, fica a cargo da fase de análise sintática, que é realizada através do *parser*. A lista de regras que define os relacionamentos entendidos pelo programa é chamada de gramática. O serviço de

[1] o compilador C é utilizado em conjunto com a biblioteca MPI.

parsing consiste em detectar automaticamente todas as sequências de *tokens* que cumprem com sucesso uma das regras da gramática, bem como identificar os erros de sintaxe, que ocorrem quando uma sequência de *tokens* não cumpre nenhuma das regras da gramática [47].

O compilador Gamma da implementação *Gamma-Base* foi desenvolvido na linguagem de programação C, e para cumprir as fases de análise léxica e sintática da compilação, foram utilizadas as ferramentas padrão UNIX *lex* & *yacc* da AT&T [47]. Ao término dessas fases, tem-se como resultado uma série de ponteiros que referenciam inúmeras estruturas de dados, as quais armazenam todas as informações do programa de entrada, já separadas por tipos, como expressões, nomes dos programas, elementos do multiconjunto e definições. Essa última se divide ainda em outros ponteiros, que apontam para estruturas contendo as partes que compõem uma reação, que são as condições de reação e as ações que modificarão o multiconjunto. Em [45] são dados maiores detalhes sobre estas duas fases da compilação, além de outras particularidades do compilador.

Concluídas as etapas citadas, o compilador encontra-se apto para progredir para as etapas seguintes, entre as quais destaca-se a de geração de código, como veremos adiante.

4.3.2 Palavras Reservadas e Operadores

Conforme vimos no Capítulo 2, o paradigma Gamma foi proposto como um formalismo para a definição de programas de maneira simples e natural, sem a adição de restrições ou artificialidades comumente encontradas nos tradicionais paradigmas sequenciais. Devido a esta natureza mínima da linguagem, o número de recursos para a expressão de programas também é mínimo, e consequentemente, o número de palavras reservadas e operadores requeridos para uma implementação de Gamma não é grande. Contudo, para possibilitar a expressão de programas um pouco mais complexos, surgiram propostas de extensões linguísticas ao Gamma, como por exemplo, os operadores de composição de programas (ver Subseção 2.3.1), os quais possibilitaram criar algum tipo de restrição no acesso ao multiconjunto, além de uma melhor modularização do problema, através do uso de várias reações conectadas pelos referidos operadores.

A implementação *Gamma-Base* realizou exatamente o que expusemos, possuindo um número reduzido de palavras reservadas, e dando suporte aos operadores de composição de programas sequencial e paralelo, além dos operadores aritméticos, lógicos e relacionais mais tradicionais. Em relação à principal estrutura que armazena os dados do programa, *Gamma-Base* suporta o uso de um único multiconjunto, o qual pode ser composto por elementos simples (valores numéricos), elementos compostos

(tuplas), ou por uma mistura dos dois tipos. Na Tabela 4.1 temos um resumo dos símbolos suportados pela implementação *Gamma-Base*.

Tabela 4.1: Símbolos suportados pela implementação *Gamma-Base*.

Tipo	**Símbolo**	**Descrição**
Palavra Reservada	if	Teste da condição de reação
	replace	Padrão de seleção de elementos do *Bag*
	by	Ação (Reescrita) no *Bag*
	where	Definição de reações
	true	Valor booleano verdadeiro
	false	Valor booleano falso
	empty	Valor vazio
Operador Lógico	and	Conectivo Lógico E
	or	Conectivo Lógico OU
Operador Relacional	>	Maior do que
	<	Menor do que
	>=	Maior ou igual a
	<=	Menor ou igual a
	==	Igual a
	!=	Diferente de
Operador Aritmético	+	Adição
	-	Subtração
	*	Multiplicação
	/	Divisão
	%	Módulo
Operador de Composição	;	Sequencial
	|	Paralelo
Símbolo Básico	,	Separador de Elementos
	=	Atribuição
	(	Início de Expressão
	)	Fim de Expressão
	[	Início de Tupla
	]	Fim de Tupla
	{	Início do Multiconjunto
	}	Fim do Multiconjunto

Para um melhor entendimento sobre a sintaxe utilizada, veremos agora alguns

exemplos de programas Gamma escritos para a implementação em questão, e poderemos notar o emprego das palavras reservadas e operadores suportados.

Exemplo 1: Cálculo do Fatorial

/* fatorial.gm */

```
fatorial { 1,2,3,...,N }
where
fatorial = replace x,y
           by x*y
           if true
```

Esse exemplo é um programa simples, composto de uma única reação chamada `"fatorial"`, que atua sobre os elementos do multiconjunto `"{1,2,3,...,N}"`. A definição da reação aparece após a palavra reservada `"where"`, e é composta pelo par *"ação(A) ⇐ condição de reação(C)"* ($A \Leftarrow C$). A ação é o que está definido pelas palavras reservadas `"replace"` e `"by"`, ou seja, `"replace x,y by x*y"`, e só ocorre caso a condição de reação `"if true"` for satisfeita. Nesse programa, a condição de reação é sempre verdadeira, e a ação ocorre até que reste apenas um elemento no multiconjunto, que será a resposta do cálculo do fatorial de N *(N! = N * N-1 * N-2 * ... * 1)*. Podemos notar que o programa utiliza o padrão de construção de reações chamado de *Redução*, pois a reescrita do multiconjunto ocorre retirando-se dois elementos e adicionando apenas um, que é o produto entre eles.

Exemplo 2: Algoritmo de Ordenação

/* sort.gm */

```
init ; sort { N elementos }
where
init = replace x
       by [0,x]
       if true
sort = replace [i,x], [j,y]
```

```
        by [i,x], [j+1, y]
        if (x <= y and i == j)
```

Neste caso, temos um programa Gamma composto por duas reações conectadas pelo operador de composição sequencial, o qual realiza a ordenação (*sort*) dos elementos do multiconjunto. A primeira reação, `"init"`, aplica o padrão de construção de reações chamado de *Transformação* sobre o multiconjunto, transformando todos os elementos simples, em tuplas de tamanho dois, nas quais o primeiro elemento representa o índice de ordenação (inicialmente todos os índices recebem o valor zero), e o segundo elemento é o próprio valor que estava no multiconjunto inicial. Quando a reação `"init"` termina, a segunda reação, `"sort"`, pode iniciar seu processamento. Essa reação também realiza a *Transformação* sobre o multiconjunto, através da alteração dos índices das tuplas, de acordo com o valor de seus elementos. No final, o multiconjunto resultante consiste em um conjunto de tuplas de dois elementos cada, que estão ordenadas pelos valores de seu segundo elemento, seguindo a ordem sequencial de seus índices: $\{\ [0, x_1], [1, x_2], \ldots, [(N-1), x_n]\ \}\ \mid\ (x_1 \leq x_2 \leq \ldots \leq x_n)$.

Exemplo 3: Modelo Genérico de Programa

/* generico.gm */

R_1 ; R_2 ; ... ; R_N { Elementos do Multiconjunto }
where
R_1 = replace $\{x_1,\ x_2,\ ...,\ x_n\}$
by $f_1(x_1,\ x_2,\ ...,\ x_n)$
if "condição de reação"
R_2 = replace $\{x_1,\ x_2,\ ...,\ x_n\}$
by $f_2(x_1,\ x_2,\ ...,\ x_n)$
if "condição de reação"
.
.
.
R_N = replace $\{x_1,\ x_2,\ ...,\ x_n\}$
by $f_N(x_1,\ x_2,\ ...,\ x_n)$
if "condição de reação"

Este último exemplo demonstra de forma genérica a sintaxe suportada pela implementação *Gamma-Base*. Como podemos notar, é possível construir um programa com um número ilimitado de reações, conectadas pelos operadores de composição sequencial (';'), como no código mostrado, ou paralelo ('|'). Cada reação tem sua própria definição, composta pela sequência padrão de comandos `"replace...by...if..."`. Um detalhe importante que devemos ressaltar é a possibilidade de utilização de reações n-árias no escopo de cada definição, e também, a capacidade de mesclar elementos simples e tuplas nos códigos das cláusulas de programação `"replace"` e `"by"`.

4.3.3 Geração de Código

Após ter recebido como entrada um arquivo escrito em linguagem Gamma (*source.gm*), e uma vez finalizadas as fases iniciais da compilação, que são a análise léxica seguida da análise sintática, o compilador Gamma já possui a árvore gramatical criada, e pode partir para a fase posterior, que é a geração de código. Nela, o compilador realiza uma varredura completa sobre esta árvore, e gera como saída, código escrito em linguagem C (*run.c*), o qual contém todas as rotinas para execução do programa originalmente escrito em Gamma. Tais rotinas englobam a criação e inserção dos elementos no multiconjunto, e as reações, que são compostas pelas verificações das condições de reação e pela reescrita no multiconjunto (ações). Para cada reação, é criada uma rotina contendo chamadas em MPI, denominada célula trabalhadora (T). Em programas que possuem mais de uma reação, conectadas pelos operadores de composição, há a criação de uma rotina para cada operador presente. Esta rotina é denominada de célula controladora (C). Em tempo de execução, cada célula criada será executada por um processo distinto no ambiente distribuído.

Visto que a saída do compilador Gamma é um arquivo escrito em linguagem C, que ainda é de alto nível, podemos fazer uma analogia com os compiladores tradicionais, e considerar que o compilador Gamma realiza a geração de um código intermediário, o qual ainda necessita posterior processamento, para se tornar um arquivo executável compatível com uma arquitetura de processador específico. Logo, temos um esquema de compilação em duas etapas (Figura 4.2). O compilador Gamma executa a primeira, que já foi descrita. A segunda etapa, consiste em compilar o código gerado, conjuntamente com alguns outros arquivos, também escritos em C, que compõem o *framework*[2] da implementação, gerando finalmente, o arquivo binário executável, apto para ser executado em um ambiente distribuído dotado de múltiplos processadores. Na *Gamma-Base*, a segunda etapa foi realizada através do compilador padrão para a linguagem C nos sistemas operacionais Linux, o GCC

[2]conjunto de códigos-fonte que formam o arcabouço de desenvolvimento.

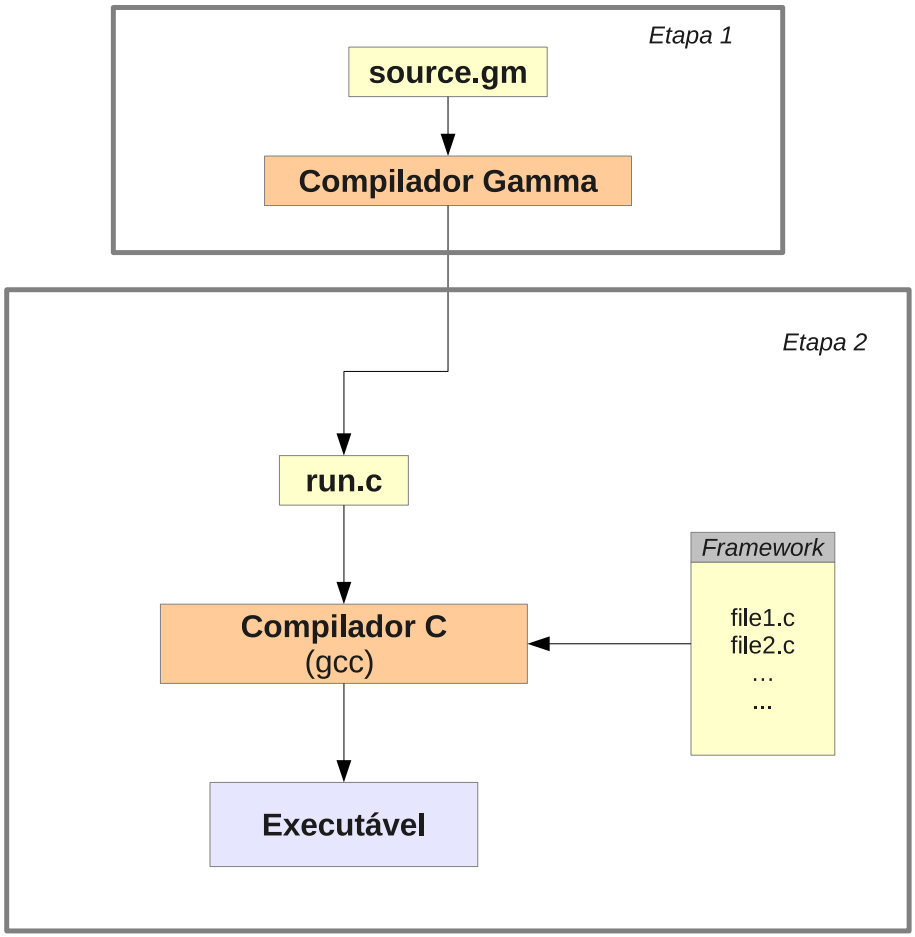


Figura 4.2: Etapas de compilação da implementação *Gamma-Base*.

(*GNU Compiler Collection*) em conjunto com a biblioteca MPI.

4.4 Ambiente de Execução

Depois do programa escrito em Gamma ter passado pelas etapas de compilação descritas, obtém-se o arquivo executável compatível com um ambiente distribuído de processamento. A execução pode ocorrer sobre um processador com arquitetura *multicore*, ou idealmente, sobre um *cluster* de computadores, onde cada nó possui seu próprio processador, possivelmente *multicore*. A chamada ao executável é realizada através do ambiente fornecido pela interface de trocas de mensagens utilizada, que foi a *Message Passing Interface* (MPI), como já dito anteriormente. Desta forma, cada célula criada pelo programa, é atribuída a um processo distinto, e a comunicação entre elas, ocorre através de primitivas de envio e recebimento de mensagens.

Na Figura 4.3, temos um exemplo da configuração de células que seriam criadas no ambiente distribuído, para execução de um programa Gamma composto por três reações `"R1|R2;R3"`, conectadas pelos operadores de composição paralelo e sequencial, respectivamente. Neste caso, as reações `"R1"` e `"R2"` podem execu-

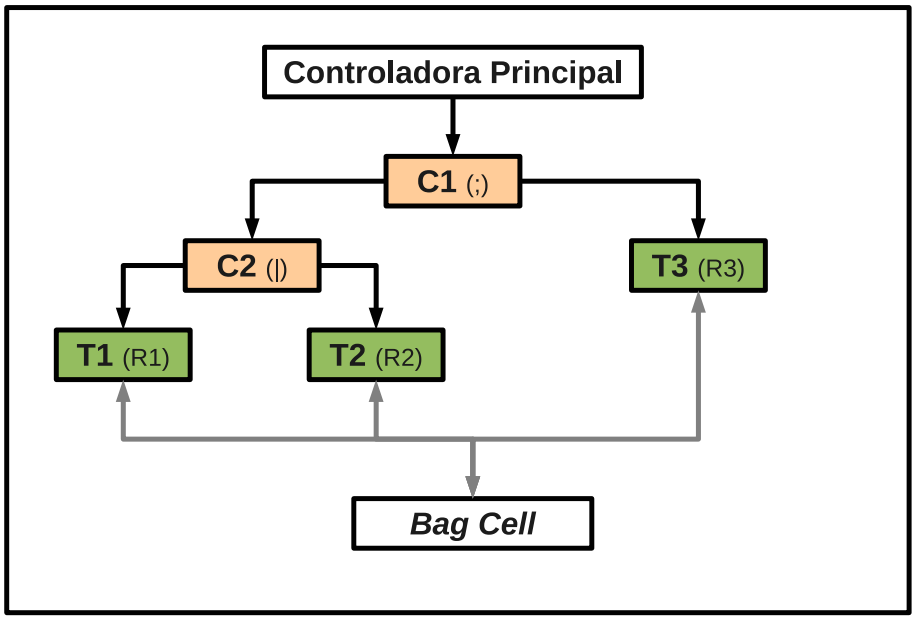


Figura 4.3: Células criadas para um programa composto por três reações `"R1|R2;R3"` na implementação *Gamma-Base*.

tar concorrentemente de forma alternada, ao passo que a reação `"R3"`, só poderá executar após as outras duas terem finalizado seus processamentos.

Podemos notar que foram criadas duas células do tipo Controladora (C1 e C2) e três células do tipo Trabalhadora (T1, T2 e T3). Cada uma das controladoras fica responsável por executar a estratégia de controle relativa a um operador de composição, e cada uma das trabalhadoras, executa o processamento de uma das reações do programa. Depois que todas as trabalhadoras tiverem finalizado suas execuções, a célula Controladora Principal encerra o programa, matando todos os processos associados às demais células.

4.5 Proposta de Escalonador

Podemos dizer que a implementação *Gamma-Base* foi um trabalho bem sucedido, e cumpriu sua principal aspiração, que foi realizar uma implementação paralela e distribuída do formalismo Gamma, com suporte a programas com mais de uma reação do tipo n-ária. Entretanto, assim como em outras implementações de Gamma realizadas, foram constatados problemas de eficiência na execução dos programas, uma vez que os modelos de escalonamento de tarefas eram muito simples, não suportando execução concomitante entre várias reações, ou ainda pelo fato de não haver, uma exploração adequada do potencial paralelismo no acesso ao multiconjunto, mesmo no escopo de uma única reação.

Tendo exposto o fato acima, foi proposto em [45, 48], um novo modelo de escalonamento para Gamma, no qual levou-se em conta a existência de reações n-árias,

podendo estar presentes em qualquer número, e todas atuando sobre o mesmo multiconjunto. Nesta proposta foram discutidos três relevantes problemas relacionados a este aspecto, que são: a seleção dos elementos do multiconjunto para teste das condições de reações; como distribuir esses testes entre os elementos de processamento; e como distribuir os processos no sistema físico que executará os programas Gamma. A ideia era que o escalonador atuasse em tempo de compilação e execução, de modo que elementos do multiconjunto pudessem ser adicionados ou removidos, incorrendo na possível criação de novos processos em tempo de execução.

O algoritmo proposto para o novo modelo de escalonador teve como base o trabalho desenvolvido em [49], que aborda o mapeamento ótimo de sistemas com restrições sobre a vizinhança. Este mapeamento por sua vez, utiliza como representação dos recursos compartilhados, grafos orientados finitos, além de técnicas conhecidas como SER(*Scheduling by Edge Reversal*) e MCC(*Minimum Clique Covering*) [50]. A dinâmica básica do algoritmo consiste em partir de um grafo orientado acíclico, e encontrar um nó *sink*[3], o qual poderá executar. Depois, a cada passo, a reversão de arestas é aplicada, causando o surgimento de novos *sinks*. O número de orientações acíclicas possíveis do grafo, chamado período, é finito, e por essa razão, em algum momento, tais orientações começam a se repetir, fazendo com que os respectivos *sinks* executem novamente. Isso faz com que não ocorram problemas como *deadlocks* ou *starvation* [10], visto que fica garantida a exclusão mútua no acesso aos recursos compartilhados, e também fica a garantia de que um nó qualquer será *sink*, e portanto executará, ao menos uma vez dentro do período.

A implementação deste novo escalonador poderia realmente trazer ganhos de eficiência para a execução de programas Gamma, mas por outro lado, devido a sua política mais elaborada, alguns custos adicionais poderiam aparecer, como por exemplo, custos de comunicação e sincronização entre os processos que representam os nós do grafo, introduzidos pela própria lógica necessária para manter a correta execução do algoritmo de reversão de arestas (SER+MCC). Para descobrir se os custos seriam superados pelo ganho de desempenho, seria preciso experimentar o escalonador, todavia, esta etapa não foi realizada, tendo o trabalho ficado somente como uma proposta para implementação futura.

É nesse contexto que surgiu a ideia da solução desenvolvida nesta dissertação (*Gamma-GPU*), descrita no capítulo seguinte, na qual buscamos aumentar a eficiência da execução de programas Gamma, através de uma melhor exploração do paralelismo intrínseco ao formalismo, fazendo uso das Unidades Gráficas de Processamento (GPUs).

[3] *sink* é um vértice do grafo, onde todas as arestas que estão conectadas ao mesmo, são direcionadas a seu favor.

Capítulo 5

Gamma-GPU (Solução Desenvolvida)

Após termos exposto no capítulo anterior, a implementação de Gamma chamada de *Gamma-Base*, dedicaremos o presente capítulo para a solução desenvolvida nesta dissertação, batizada de *Gamma-GPU*, que estendeu a solução base, adicionando à arquitetura do modelo uma nova e poderosa unidade de processamento paralelo, a GPU.

5.1 Proposta

Podemos sintetizar a proposta deste trabalho na seguinte sentença:

"Mapear e implementar o paradigma de reescrita de multiconjuntos Gamma em uma arquitetura heterogênea de processamento paralelo, baseando-se em uma implementação distribuída já existente, e adicionando suporte ao uso das GPUs".

Em outras palavras, o desafio foi estudar uma implementação já existente de Gamma, e estender a mesma, de modo que passasse a ser capaz de utilizar a capacidade de processamento das GPUs, visando um potencial ganho de desempenho para os programas escritos em Gamma. Adicionalmente, o trabalho almejou fornecer uma abstração para a programação de GPUs, uma vez que o programador Gamma não precisa ter nenhum conhecimento sobre os detalhes de programação das mesmas, pois esta tarefa fica a cargo do compilador Gamma-GPU, sendo transparente ao usuário.

A ideia de unificar estes dois conceitos em um mesmo escopo, ou seja, aliar o modelo computacional Gamma ao processamento das GPUs, surgiu de uma constatação que remete à metáfora de reações químicas de Gamma e ao modelo de paralelismo

adotado nas GPUs. Como vimos no Capítulo 3, a computação das GPUs pode ser classificada como sendo do tipo SIMD (*Single-Instruction, Multiple-Data*) ou SIMT (*Single-Instruction, Multiple-Threads*), e consiste basicamente em aplicar um mesmo fluxo de instruções sobre fluxos de dados distintos, geralmente dispostos na forma de um *array*. Tal abordagem nos faz pensar em uma analogia direta com o paradigma do formalismo Gamma, no qual uma mesma reação (instrução) pode ocorrer simultaneamente sobre moléculas distintas (dados), de modo que a verificação da condição de reação pode ser realizada paralelamente sobre os elementos do multiconjunto (*array*), e aquelas que satisfizerem tal condição, poderão então executar a ação, que consiste na reescrita do multiconjunto.

As demais seções deste capitulo serão utilizadas para descrevermos a solução *Gamma-GPU*, abordando temas como sua arquitetura, as modificações realizadas no compilador Gamma, os detalhes da implementação, e a análise de complexidade. Finalizaremos com a citação de alguns trabalhos correlatos ao desenvolvido nesta dissertação.

5.2 Arquitetura Heterogênea

A arquitetura distribuída da implementação *Gamma-Base* (mostrada na Seção 4.2), é composta por várias células de processamento criadas em tempo de compilação, que são executadas por processos do padrão MPI. O ambiente de hardware para a execução desses processos é formado por múltiplas CPUs, podendo ser um único processador *multicore*, ou idealmente, um *cluster* de computadores, onde cada nó possui sua própria CPU, possivelmente *multicore*. Pois bem, para que pudéssemos realizar o mapeamento do formalismo Gamma para execução sobre o nosso novo modelo *Gamma-GPU*, era necessário escolher em que ponto da arquitetura as GPUs seriam inseridas, e até mesmo, se a característica distribuída da arquitetura já existente seria mantida.

Nossa escolha foi inserir as GPUs no contexto de computação das células trabalhadoras, e manter a mesma arquitetura e o mesmo arcabouço de execução de programas da implementação *Gamma-Base*. O fato de nossa solução ter mantido o suporte a MPI, faz com que possamos executar os programas Gamma tanto em um único computador equipado com uma GPU, ou em um ambiente *multi-GPU* (com múltiplas GPUs), como por exemplo, um *cluster* de GPUs, onde cada nó de processamento é um computador equipado com CPU e GPU. Todo o processamento que ocorre em uma GPU deve ser iniciado a partir de uma CPU, que a controla e invoca seus *kernels* (funções que executam na GPU, visto na Seção 3.3). Assim, surge o modelo de computação heterogêneo CPU/GPU, buscando obter o melhor de cada elemento, com a CPU realizando o controle e as restrições sequenciais do programa,

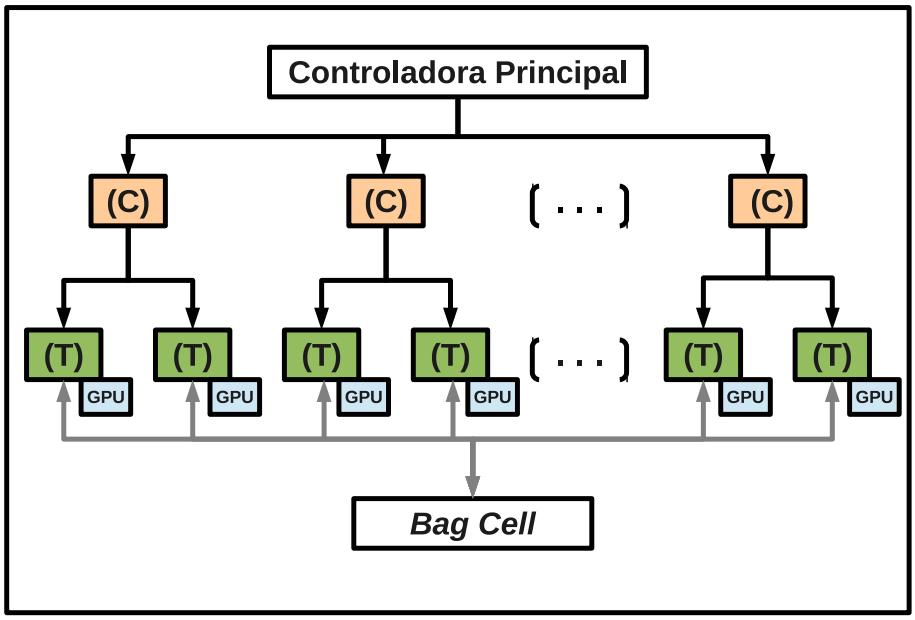


Figura 5.1: Arquitetura Heterogênea da implementação *Gamma-GPU*.

e a GPU sendo utilizada para o processamento intensivo de cálculos simultâneos. Na Figura 5.1, podemos ver a arquitetura heterogênea da nossa nova implementação *Gamma-GPU*, na qual as CPUs ficam responsáveis pelo controle das células e pela orquestração do fluxo de execução, através do uso de troca de mensagens empregando MPI, e as GPUs, ficam encarregadas de processar os cálculos das reações do programa Gamma, no âmbito das células trabalhadoras.

5.3 Compilador *Gamma-GPU*

Como vimos na seção anterior, a arquitetura da nossa implementação seguiu os mesmos moldes arquiteturais da implementação *Gamma-Base*. Consequentemente, o compilador *Gamma-GPU*, utilizou o compilador Gamma já existente (descrito na Seção 4.3) como base para sua construção, e realizou as extensões necessárias a fim de prover suporte à geração de código para executar nas GPUs. Assim sendo, todas as funcionalidades e capacidades presentes no compilador Gamma da implementação base, continuam válidas para o compilador atual, incluindo o suporte a todas as palavras reservadas e operadores previamente citados. Na realidade, as fases iniciais de análise léxica, sintática, e gramatical, continuam exatamente as mesmas, e isso significa que os programas escritos em Gamma, submetidos como entrada ao compilador, permanecem com a mesma sintaxe de antes, de modo que o programador Gamma não precisa aprender nenhum novo conceito para utilizar nossa implementação *Gamma-GPU*. As mudanças no compilador ocorrem na fase interna de geração de código, pois agora o código-fonte gerado é compatível com um ambiente heterogêneo CPU/GPU, e isto ocorre de forma totalmente transparente

ao programador, que irá apenas se beneficiar com a capacidade de processamento paralelo das GPUs, sem a necessidade de conhecer os detalhes de baixo nível envolvidos na programação das mesmas, como por exemplo, conceitos de *kernels*, blocos e *threads*.

Assim como no compilador base, o novo compilador permanece trabalhando em duas etapas (Figura 5.2), contudo, agora temos novas características presentes. A entrada do compilador continua sendo um código-fonte escrito em Gamma (*source.gm*), porém, foi adicionada a possibilidade de passar ao compilador o parâmetro de compilação `[--gpu]`, o qual sinaliza o desejo de gerar código com ou sem suporte a GPUs. Se o parâmetro não for passado, então o compilador funciona normalmente como na versão base, gerando como saída da primeira etapa apenas o arquivo com código em linguagem C (*run.c*). Porém, caso o parâmetro seja passado ao compilador, serão gerados como saída da primeira etapa dois arquivos, ao invés de apenas um. O primeiro continua sendo o arquivo em linguagem C (*run.c*), mas, seu conteúdo agora é modificado de modo a incluir todo o código necessário para realizar a preparação e as chamadas aos *kernels* que executarão nas GPUs. Já o segundo, é um arquivo de código-fonte escrito na linguagem CUDA[1] (*run_gpu.cu*), que contém os *kernels* que serão executados inúmeras vezes pelas *threads* criadas na chamada dos mesmos. Nesse caso, com o suporte a GPUs ativo, são os *kernels* CUDA os responsáveis por realizar o processamento das reações do programa Gamma no escopo de cada célula trabalhadora, podendo se beneficiar de uma execução altamente paralela, propiciada pelas centenas de *CUDA-Cores*[2] existentes no hardware das GPUs.

A segunda etapa da compilação também sofreu alterações, pois foi necessária a criação de novos arquivos escritos em linguagem C e em linguagem CUDA, para compor o *framework* da implementação e dar suporte ao novo modelo. Além disso, o compilador usado nesta etapa passou a ser o NVCC (*NVIDIA's CUDA Compiler*), necessário para poder compilar os arquivos escritos em CUDA. Como já vimos na Seção 3.3, um programa em CUDA é composto por uma mistura de dois tipos de código: o código que executará na CPU (*host*), que chamamos de *host-code*, e o código que será processado pela GPU (*device*), denominado *device-code*. Existem palavras reservadas especiais em CUDA para identificar quais os trechos de código que serão executados no *device*. O NVCC processa o programa escrito em CUDA usando as palavras reservadas especiais para separar o *host-code* e o *device-code*, e então realiza a compilação do *device-code*, gerando a parte do executável relativa à GPU. O *host-code* é delegado pelo NVCC ao compilador padrão do *host* (GCC), que o compila para gerar a parte do executável relativa à CPU. Ao final desta

[1] CUDA fornece integração com várias linguagens de programação, e neste trabalho, a opção utilizada foi o uso de CUDA em conjunto com a linguagem C, denominada CUDA C.

[2] Denominação dada aos *cores* presentes nos multiprocessadores de *streaming* das GPUs da NVIDIA.

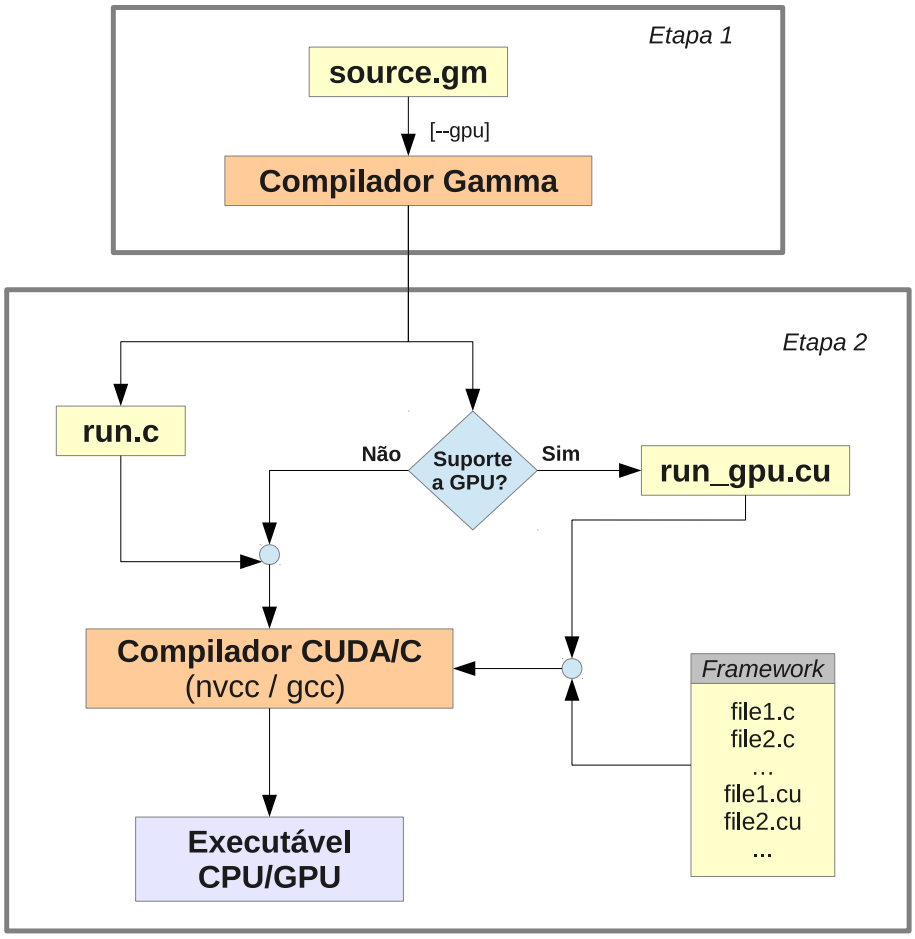


Figura 5.2: Etapas de compilação da implementação *Gamma-GPU*.

etapa, temos como saída um arquivo executável heterogêneo CPU/GPU, que pode ser executado em um ambiente distribuído onde cada nó possui uma GPU, como por exemplo, um *cluster* de GPUs. No Apêndice A, são mostrados os códigos-fonte gerados pelo compilador *Gamma-GPU* para uma aplicação escrita em Gamma (além de códigos gerados pela versão *Gamma-Base*, e por uma versão sequencial de Gamma chamada *Gamma-Seq*, da qual falaremos brevemente no capítulo de experimentos e resultados).

Para encerrar esta seção, ressaltamos que conforme expusemos na Subseção 3.3.3, existem outras linguagens para programação de GPUs além de CUDA, mas nossa escolha se deu pelo fato desta tecnologia ser amplamente aceita e largamente utilizada nos dias atuais em aplicações de HPC (*High-Performance Computing*) que utilizam as GPUs como aceleradoras de processamento.

5.4 Detalhes de Implementação

Agora que já tivemos uma visão geral sobre a arquitetura heterogênea, e um entendimento sobre o processo de compilação e geração de código presentes em *Gamma-*

GPU, iremos nos aprofundar um pouco mais nos detalhes da implementação realizada. Veremos como se dá a dinâmica de processamento de um programa Gamma em nosso modelo, e como ocorre a divisão de tarefas entre CPU e GPU durante a execução, analisando precipuamente a estratégia adotada para processar as reações sobre o multiconjunto na GPU.

A fim de facilitar a compreensão desses detalhes de implementação, utilizaremos como caso de estudo ao longo de toda a presente seção, o seguinte programa escrito em Gamma, que calcula o maior elemento presente no multiconjunto:

/* maximo.gm */

```
max { 8,2,5,9,1,4,7,3 }
where
max = replace x,y
      by x
      if (x >= y)
```

O programa possui apenas uma reação, chamada `"max"`, a qual dizemos possuir tamanho dois, pois são dois os elementos reagentes (`"x"` e `"y"`) presentes na cláusula `"replace"`. Tais elementos são combinados para a verificação da condição de reação `'x >= y'`, contida na cláusula `"if"`. No caso da condição ser verdadeira, a ação correspondente é realizar a reescrita no multiconjunto, eliminando o elemento `"y"` e mantendo o elemento `"x"` (`"replace x,y by x"`). Isso se repete até que reste apenas um elemento no multiconjunto, que será a resposta do programa, ou seja, o maior elemento. O multiconjunto utilizado no programa é formado apenas por elementos simples (valores numéricos), porém, outros programas podem utilizar em seus multiconjuntos elementos compostos (tuplas), ou até mesmo, multiconjuntos mistos, que possuem uma mistura de tuplas e elementos simples. O tipo do elemento, simples ou tupla, que aparece na primeira posição da combinação dos elementos reagentes da cláusula `"replace"`, é importante para a estratégia de escalonamento do multiconjunto a ser processado pela GPU, como veremos adiante.

5.4.1 Dinâmica de Processamento

Como já sabemos, a arquitetura distribuída da nossa implementação funciona criando células para realizar as funcionalidades expressas no programa Gamma. Duas células são sempre criadas, independente do programa, que são a Controladora Principal e a *Bag-Cell* (explicadas na Seção 4.2). Em programas compostos por mais de

uma reação, são criadas as células do tipo controladora, que ficam responsáveis pela execução da semântica dos operadores de composição. Para cada reação, é criada uma célula do tipo trabalhadora, que fica responsável por executar as reações propriamente ditas sobre o multiconjunto. O que chamamos de executar uma reação, significa na verdade testar a condição de reação para cada combinação possível entre os elementos do multiconjunto, e aplicar a ação correspondente (reescrita do multiconjunto) para aquelas combinações cujas condições forem avaliadas como satisfatórias, ou seja, que cumprirem a condição. No nosso programa exemplo há apenas uma reação (chamada `"max"`), e portanto, nesse caso é criada apenas uma única célula trabalhadora, que fica responsável por executá-la. O processamento na célula trabalhadora ocorre em várias iterações, as quais se iniciam através do recebimento do multiconjunto, enviado pela *Bag-Cell*, modificam o mesmo caso alguma combinação de elementos reaja, e se encerram com a devolução do multiconjunto à *Bag-Cell*.

Na implementação *Gamma-Base*, este processamento no escopo de cada célula trabalhadora é feito de forma sequencial, através de *loops* aninhados que testam a condição de reação para cada possível combinação de elementos do multiconjunto. Durante estas verificações, quando se encontra uma combinação que satisfaça a condição de reação, a ação correspondente é executada, e a célula interrompe os *loops*, devolvendo o multiconjunto já reescrito para a *Bag-Cell*, dando a oportunidade para que outras células trabalhadoras recebam o multiconjunto, e executem suas respectivas reações. Já na implementação *Gamma-GPU* desenvolvida neste trabalho, é nesse trecho do processamento que as maiores mudanças ocorrem. Na arquitetura heterogênea do nosso novo modelo, partes da computação de cada reação, no escopo da sua célula trabalhadora, são transmitidas para processamento na GPU, e a dinâmica das combinações de elementos para verificação da condição de reação é alterada, buscando tirar o máximo proveito do paralelismo oferecido pelos *many-cores* presentes neste tipo de unidade de processamento.

Na *Gamma-GPU*, a dinâmica de processamento que ocorre em uma célula trabalhadora segue os seguintes passos:

1. [CPU] O multiconjunto é requisitado e recebido da *Bag-Cell*;
2. [CPU] O multiconjunto é transformado em um formato de *array* (seu formato padrão é uma lista encadeada) para se adequar ao processamento nas GPUs;
3. [CPU] É realizada a alocação de memória no *device* e o multiconjunto em formato de *array* é copiado para a GPU (cópia de memória *host-to-device*);
4. [CPU] São calculados o número de blocos e *threads*, e o *Kernel* CUDA é invocado;

5. [**GPU**] Cada *thread* realiza a combinação do seu elemento-base (será definido adiante) com os demais elementos do multiconjunto, testando a condição de reação;

6. [**GPU**] As *threads* nas quais alguma combinação reage, marcam os elementos utilizados na combinação como já processados, preenchem uma estrutura adicional de resposta a ser devolvida à CPU, e interrompem seu processamento;

7. [CPU] Quando todas as *threads* na GPU terminam sua execução, o processamento retorna para CPU, que realiza a cópia da estrutura de resposta da memória da GPU para sua memória local (cópia de memória *device-to-host*), e libera a memória da GPU;

8. [CPU] As ações relativas aos elementos que reagiram são aplicadas (reescrita do multiconjunto), baseando-se na estrutura de resposta preenchida pelas *threads* da GPU;

9. [CPU] O multiconjunto é devolvido para a *Bag-Cell*, para que outras células trabalhadoras possam executar.

Os passos descritos estão também ilustrados na Figura 5.3, e nos permitem ter uma visão macro da dinâmica de processamento no escopo de uma célula trabalhadora, nos dando uma percepção da cooperação existente entre CPU e GPU, e mostrando quais tarefas são executadas pelo *host* e quais são transmitidas para o *device*. Nas próximas seções, faremos o detalhamento de alguns dos principais passos exibidos.

5.4.2 Execução dos *Kernels* CUDA

Nesta subseção focaremos nas estratégias adotadas relativas ao emprego da GPU no processamento das reações dos programas Gamma. Iniciaremos falando sobre como o multiconjunto foi explorado de forma paralela nos *kernels* CUDA, e depois abordaremos alguns outros pontos importantes, como o tratamento a multiconjuntos com tipos compostos, a estrutura na qual as *threads* da GPU armazenam os resultados das reações, e finalmente, algumas otimizações implementadas.

5.4.2.1 Escalonamento do Multiconjunto na GPU

O escalonamento do multiconjunto diz respeito aos métodos utilizados para a exploração paralela da estrutura durante a execução dos programas. A implementação *Gamma-Base* possui um escalonamento simples, que se dá em tempo de compilação, através da criação de um número fixo de células trabalhadoras para processar as

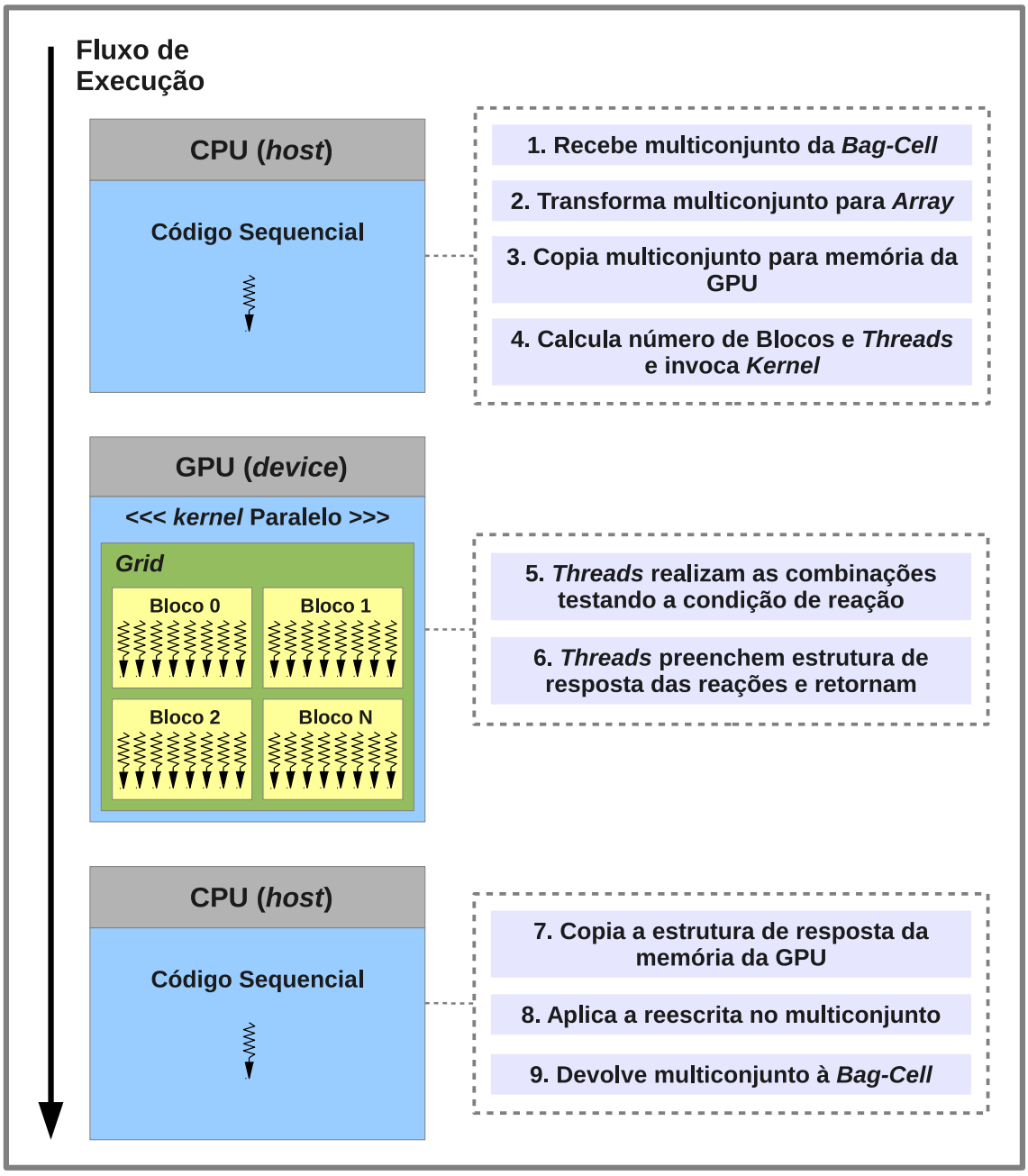


Figura 5.3: Dinâmica de processamento no escopo de uma célula trabalhadora na implementação *Gamma-GPU*.

reações. A simplicidade dos escalonadores reside no fato de não suportarem execução concomitante entre várias reações, ou ainda pelo fato de não haver uma exploração adequada do potencial paralelismo no acesso ao multiconjunto, mesmo no escopo de uma única reação. A proposta de escalonador que citamos na Seção 4.5, buscava resolver principalmente a primeira deficiência, através do suporte a várias reações atuando simultaneamente sobre o mesmo multiconjunto. Diferentemente dessa proposta, nosso escalonador focou na resolução da segunda deficiência, e atacou o problema de exploração do paralelismo no âmbito de cada reação individualmente, pois esse é o caminho mais compensador e natural para adequar o modelo Gamma ao

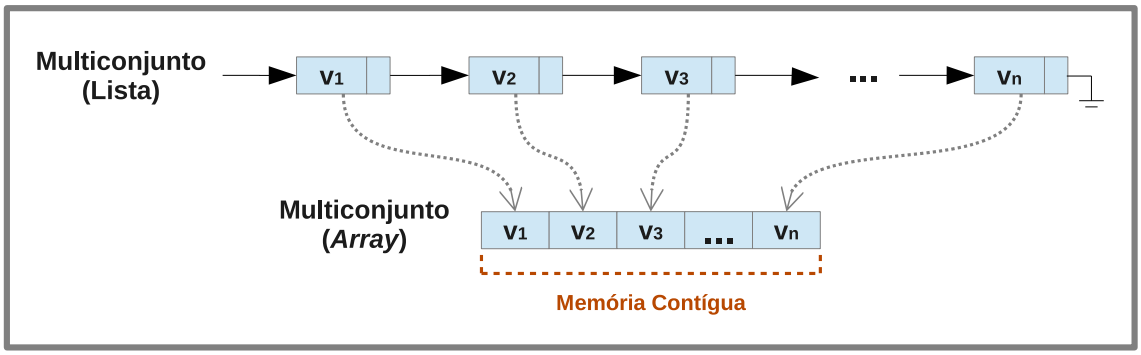


Figura 5.4: Transformação do multiconjunto de lista encadeada para *array*.

processamento nas GPUs. Nosso escalonamento é dinâmico e acontece em tempo de execução, uma vez que o número de *threads* que acessam o multiconjunto é variável, ajustado de acordo com as características dos elementos e com o tamanho no qual o multiconjunto se encontra em cada iteração, como veremos adiante.

Após ter recebido o multiconjunto, o passo subsequente no processamento consiste na transformação do formato do mesmo. A estrutura de dados originalmente utilizada para a representação do multiconjunto é uma lista encadeada, a qual não é adequada para o processamento nas GPUs. Logo, a lista encadeada sofre uma transformação para o formato de *array*, de modo que os dados fiquem arranjados de maneira contígua na memória (Figura 5.4).

Com o multiconjunto já em formato de *array*, entramos na fase que chamamos de pré-processamento de invocação do *kernel*, que consiste em realizar a alocação de espaço de memória na GPU, seguida da cópia do multiconjunto (*array*) da memória da CPU para a memória recém-alocada na GPU. Para realizar esta etapa, foram utilizadas as seguintes chamadas de função disponibilizadas pela API do CUDA *Runtime*:

```
cudaMalloc((void**) d_array_data, RunTimeBagArray->size * sizeof(int));

cudaMemcpy(*d_array_data, RunTimeBagArray->data,
           RunTimeBagArray->size * sizeof(int), cudaMemcpyHostToDevice);
```

O último parâmetro da chamada `"cudaMemcpy"` destaca o sentido da cópia entre as memórias, no caso, da CPU (*host*) para a GPU (*device*).

Agora que o multiconjunto já se encontra na memória da GPU, o próximo passo é a invocação do *kernel* CUDA, para que realize seu trabalho. Entretanto, antes de podermos invocá-lo de fato, é preciso que seja determinado o número de *threads* que serão criadas, e a divisão das mesmas em blocos dentro do *grid* do

kernel (conforme visto na Seção 3.3). Para que possamos explicar como se chega no número de *threads* a serem criadas, é necessário que façamos a introdução do conceito de "elemento-base".

Elemento-Base: "um elemento qualquer do multiconjunto será considerado um elemento-base se possuir o mesmo tipo e tamanho do primeiro elemento reagente contido na cláusula `"replace"` da definição da reação no programa Gamma."

Para esclarecer esta definição, vejamos um exemplo. Considere o seguinte trecho de um programa hipotético escrito em Gamma:

```
R { [3,3,3],[2,2],5,9,[1,7],[4,7,6],8,[9,9] }
where
R = replace ([x,y], [a,b,c], v)
    by ...
    if ...
```

Neste programa temos um multiconjunto misto, composto por três elementos simples, `"5,9,8"`, por três tuplas de tamanho dois, `"[2,2],[1,7],[9,9]"`, e por duas tuplas de tamanho três `"[3,3,3],[4,7,6]"`. Existe apenas uma reação, chamada de `"R"`, que é ternária, pois as combinações dos elementos são realizadas de três em três, e cada combinação é composta de: uma tupla de tamanho dois, uma tupla de tamanho três, e um elemento simples, nesta ordem, como exibido pela cláusula `"replace ([x,y], [a,b,c], v)"`. Teríamos então neste exemplo, como elementos-base, todos os elementos do multiconjunto que sejam do tipo tupla com tamanho igual a dois, pois é este o tipo de elemento que aparece como primeiro reagente da cláusula `"replace"`. Os elementos-base do multiconjunto seriam os três destacados a seguir:

{ [3,3,3], [2,2] ,5,9, [1,7] ,[4,7,6],8, [9,9] }

No programa Gamma `"maximo.gm"`, que está sendo utilizado como exemplo desta seção, o multiconjunto é composto apenas por elementos simples, e o primeiro elemento reagente da cláusula `"replace"` também é um elemento simples. Sendo assim, todos os oito elementos do multiconjunto são classificados como elementos-base, conforme destacado a seguir:

{ 8, 2, 5, 9, 1, 4, 7, 3 }

Agora que já esclarecemos o conceito de elemento-base, podemos retornar ao ponto da invocação do *kernel* CUDA, e definir o número de *threads* e blocos que serão criados. Simultaneamente à transformação do multiconjunto para o formato de *array*, é realizada uma contagem do número de elementos-base presentes no mesmo. O resultado desta contagem, ou seja, o número de elementos-base existentes, é justamente o número de *threads* que serão criadas na chamada ao *kernel*, de modo que cada *thread* ficará responsável por realizar o processamento relativo a um elemento-base, conforme detalharemos adiante. No modelo de processamento adotado em CUDA, é preciso também dizer o número de blocos nos quais as *threads* serão distribuídas quando o *kernel* é chamado. Esta possibilidade de organização em blocos permite que as *threads* consigam obter dentro dos *kernels* índices únicos, bastando fazer um cálculo simples, entre o índice do bloco a qual pertence, e seu próprio índice dentro do bloco. Além disso, CUDA permite que o número de blocos e *threads* por bloco sejam especificados via variáveis tridimensionais do tipo `dim3`, com a dimensão que melhor se adequar às características do problema. Entretanto, este não é nosso caso, e a especificação do número de blocos e *threads* foi realizada de forma unidimensional. Visto que nosso trabalho trata-se de um modelo computacional no qual qualquer problema pode ser processado, desde que expresso como um programa escrito em Gamma, e não é focado na resolução de algum problema em particular, não há nenhuma necessidade de organização em especial para o sistema de índices dos blocos e *threads*. Desta forma, decidimos por criar blocos apenas na medida necessária para conter o total de *threads* a serem criadas, uma vez que os blocos possuem um número máximo de *threads* que podem suportar, e isso é diretamente dependente do hardware utilizado, variando de acordo com o modelo específico de GPU. Para realizar o cálculo do número de blocos e de *threads* por bloco, utilizamos as seguintes linhas de código:

```
int numBlocks = ceil(num_elements_base / GPU_MAX_THREADS_PER_BLOCK);

int threadsPerBlock = ceil(num_elements_base / numBlocks);
```

Primeiro é calculado o número de blocos `"numBlocks"`, fazendo a divisão entre o número de elementos-base `"num_elements_base"` (que significam o número total de *threads*), e a constante `"GPU_MAX_THREADS_PER_BLOCK"` que representa o

número máximo de *threads* suportadas em um bloco. A função matemática `"ceil"` é utilizada para arredondar o quociente da divisão para cima. Depois, também com o uso de `"ceil"`, o número de *threads* por bloco `"threadsPerBlock"` é calculado através de uma divisão do número total de *threads* pelo número de blocos. Isto nos dá uma distribuição igualitária das *threads* pelos blocos, o que em alguns casos, pode ocasionar a criação de algumas *threads* em excesso no último bloco[3] (*threads* sem elemento-base associado), fato que causaria problemas de acessos inválidos a memória dentro do *kernel*. Tal situação é facilmente contornada com a adição de uma verificação de *overflow* no índice da *thread*, inserida dentro do *kernel*, a qual evita que *threads* excessivas realizem qualquer computação. Para obter a constante `"GPU_MAX_THREADS_PER_BLOCK"`[4], utilizamos a seguinte chamada de função CUDA:

```
cudaDeviceGetAttribute(&GPU_MAX_THREADS_PER_BLOCK,
                       cudaDevAttrMaxThreadsPerBlock, gpu);
```

Uma vez calculados os valores de blocos e *threads* por bloco, podemos invocar o *kernel*, da seguinte maneira:

```
cell_reaction_kernel_fnc <<<numBlocks,threadsPerBlock>>> (d_array_data);
```

É através da chamada ao *kernel*, nomeada `"cell_reaction_kernel_fnc"`, que o processamento é transmitido da CPU para a GPU, a qual inicia a execução das *threads* em paralelo. A referência ao multiconjunto (`"d_array_data"`), que já encontra-se na memória do *device*, é passada por parâmetro na chamada do *kernel*, para que as *threads* possam acessar seus elementos durante a execução. Dentro do *kernel*, cada *thread* realiza a combinação do seu elemento-base com os demais elementos do multiconjunto, testando a condição de reação. Quando uma reação pode ocorrer, a *thread* marca os elementos utilizados na combinação como já reagidos, preenche uma estrutura adicional de resposta a ser devolvida à CPU (veremos mais detalhes em 5.4.2.3), que informa quais elementos reagiram, e interrompe seu processamento. Isto impede que elementos utilizados por uma determinada *thread* em uma reação, sejam indevidamente utilizados por outras *threads*.

Como já dissemos, cada *thread* precisa realizar processamento baseado no seu elemento-base associado. Para que ela saiba em qual posição do *array* tal elemento

[3] casos nos quais o resultado da divisão tem resto diferente de zero.

[4] Os valores de 512 ou 1024 são os mais tipicamente encontrados para este atributo nas GPUs modernas. Nas GPUs da NVIDIA, esse valor é de 512 ou 1024 na arquitetura *Fermi*, e 1024 nas arquiteturas *Kepler* e *Maxwell*.

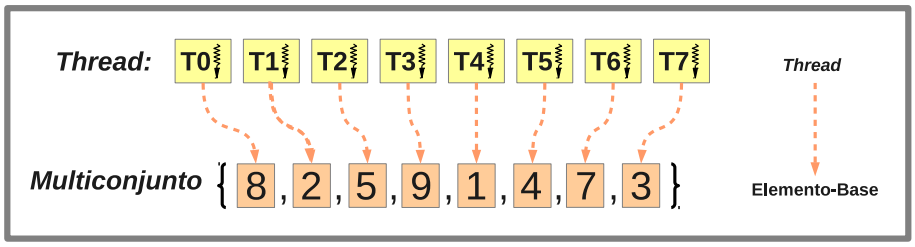


Figura 5.5: Threads e elementos-base no programa `"maximo.gm"`.

está localizado, é necessário calcular o seu índice único de identificação, que servirá como indexador do *array* que representa o multiconjunto. O índice da *thread* é calculado da seguinte maneira:

```
int thread_index = (blockDim.x * blockIdx.x) + (threadIdx.x);
```

O cálculo é feito com o uso de variáveis automaticamente fornecidas pelo ambiente CUDA no contexto dos *kernels*, que são: `"blockDim.x"`, `"blockIdx.x"`e `"threadIdx.x"`. Estas variáveis nos dão informações da dimensão do bloco (número de *threads* em um bloco), índice do bloco, e índice da *thread* dentro do bloco, respectivamente. O uso das variáveis através do acesso ao campo `"x"`, como em `"threadIdx.x"`, significa que estamos usando a primeira dimensão da mesma, que no nosso caso, é a única utilizada.

Para um melhor entendimento de tudo que foi exposto, voltaremos agora ao nosso programa exemplo `"maximo.gm"`, e vamos demonstrar a dinâmica do escalonamento do multiconjunto na GPU que ocorre neste caso. O processamento da reação `"max"` sobre o multiconjunto ocorre através de várias iterações, onde cada uma destas iterações corresponde a uma chamada do *kernel* CUDA, composto de várias *threads* atuando paralelamente. Depois de cada iteração, o multiconjunto diminui de tamanho, pois a reação `"max"` realiza a remoção de dois elementos do multiconjunto para comparação, e reescreve apenas o maior deles. Bem, na primeira iteração, temos o multiconjunto ainda completo, e já vimos que todos os seus oito elementos são classificados como elementos-base, logo, para a primeira chamada ao *kernel*, são criadas oito *threads*, todas pertencentes a um mesmo bloco. Cada *thread*, da T0 até a T7, é associada a um elemento-base, como podemos ver na Figura 5.5.

Tendo conhecimento de seu elemento-base, cada *thread* inicia sua computação em busca de reações que possam ocorrer, realizando o teste da condição de reação `"x >= y"`, entre o seu elemento-base, cujo valor está sempre em `"x"`, e os demais elementos do multiconjunto, que serão carregados em `"y"`. As possíveis combinações de elementos realizadas por cada *thread* podem ser vistas na Figura 5.6.

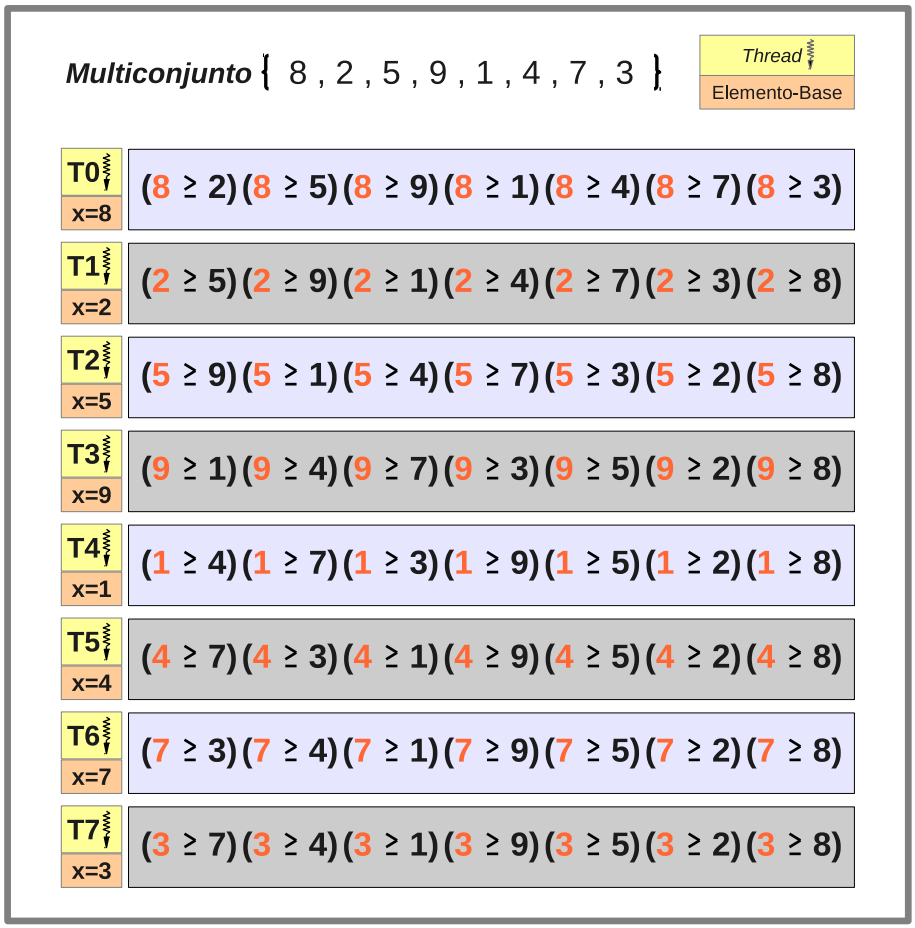


Figura 5.6: Possíveis combinações de elementos em cada *thread* no programa `"maximo.gm"`.

Em alguns programas, pode ocorrer de nenhuma combinação de elementos atender à condição de reação, o que significa que o programa chegou ao fim, pois o multiconjunto atingiu um estado estável e não pode mais ser modificado. Por outro lado, nos casos onde uma *thread* encontra uma combinação de elementos que cumpre a condição de reação, ela marca os elementos utilizados como já reagidos, e interrompe seu processamento. Devemos lembrar que um elemento não pode reagir em duas *threads* distintas ao mesmo tempo, por exemplo, caso a *thread* T0 reagisse o seu elemento-base `"8"` com o elemento do multiconjunto de valor `"2"`, ela interromperia sua execução e marcaria os dois elementos como já reagidos, o que automaticamente inviabilizaria qualquer reação de ocorrer na *thread* T1, visto que seu elemento-base é o `"2"`, e que ele já reagiu em outra *thread*. Logo, a *thread* T1 ao verificar que tal situação ocorreu, também interromperia seu processamento. Isto garante a coerência e corretude do processamento sobre o multiconjunto.

Depois que todas as *threads* criadas terminam suas execuções, seja porque reagiram um par de elementos, ou porque seu elemento-base já reagiu em outra *thread*, ou ainda porque nenhuma combinação de elementos reagiu, o *kernel* é encerrado

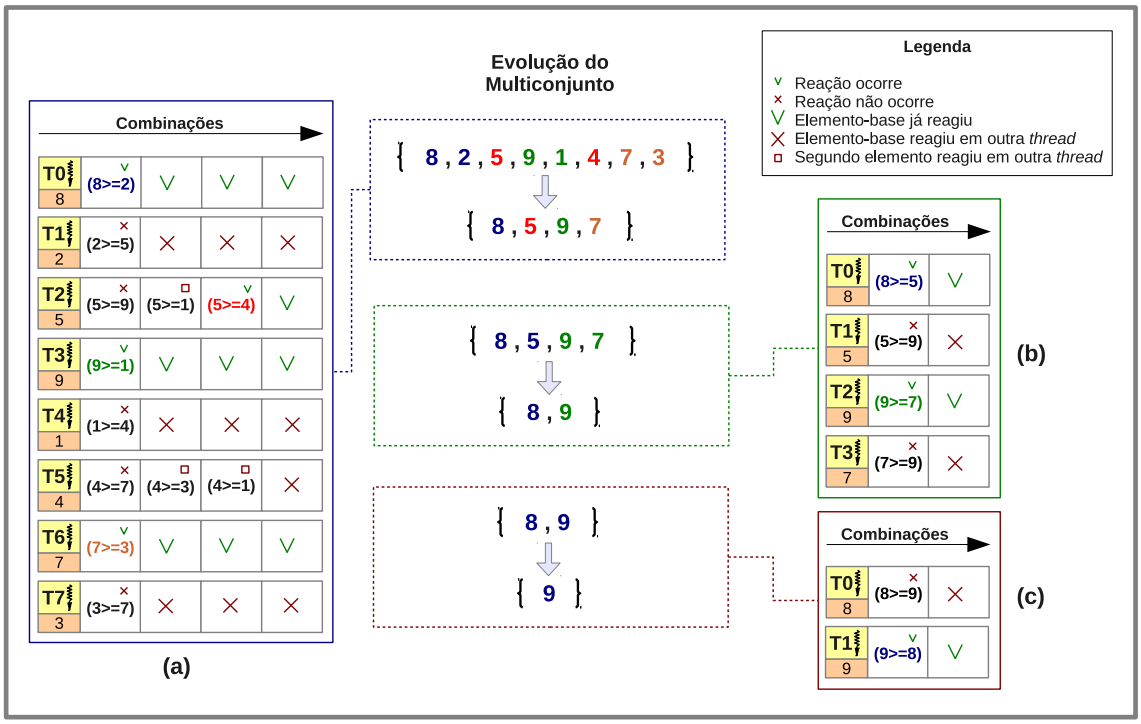


Figura 5.7: Iterações e escalonamento do multiconjunto no programa `"maximo.gm"`. **(a)** Iteração 1, **(b)** Iteração 2, **(c)** Iteração 3.

e o controle retorna para a CPU, que aplica as ações correspondentes às reações que cumpriram a condição de reação, modificando o multiconjunto e encerrando a primeira iteração do processamento. Como o multiconjunto ainda possui elementos suficientes para mais reações, e o programa ainda não atingiu sua resposta, a próxima iteração se inicia, com o *kernel* sendo invocado novamente, composto de um número de *threads* que varia de acordo com o número de elementos-base presentes no multiconjunto no momento da invocação. O processo se repete pelo número de iterações necessárias para que a resposta do programa seja obtida, ou seja, até que o multiconjunto atinja um estado no qual não há mais reações possíveis. A Figura 5.7 demonstra as três iterações que ocorrem até que a resposta do programa `"maximo.gm"` seja obtida, que neste caso, é o valor `"9"`. Nela, podemos observar a evolução do multiconjunto entre as iterações, bem como, o *status* de cada *thread*, ou seja, se realizou alguma reação ou se interrompeu seu processamento sem realizar nenhuma.

5.4.2.2 Tratando Tipos Compostos

Já vimos anteriormente que as *threads* criadas para a execução dos *kernels* na GPU, realizam seus processamentos baseadas nos chamados elementos-base do multiconjunto. Vimos também, que a dinâmica básica do processamento em cada *thread* consiste em realizar as combinações entre seu elemento-base e os demais elemen-

tos do multiconjunto, testando a condição de reação. Para que possa acessar um elemento qualquer do multiconjunto, incluindo o próprio elemento-base, a *thread* precisa conhecer a posição que ele ocupa no *array* de elementos, que como bem sabemos, é o formato no qual o multiconjunto é transformado para ser submetido ao processamento na GPU. Nos casos onde o multiconjunto é formado apenas por elementos do tipo simples, como por exemplo no programa `"maximo.gm"`, este acesso pode ser feito de maneira simples e direta, uma vez que cada elemento do multiconjunto, ocupa apenas uma posição no *array*, como podemos ver a seguir:

```
multiconjunto { 8, 2, 5, 9, 1, 4, 7, 3 }
                ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
        array ⟨ 8, 2, 5, 9, 1, 4, 7, 3 ⟩
```

Nesse caso, uma indexação direta resolveria o problema do acesso, uma vez que temos oito elementos no multiconjunto e também oito posições no *array*. Supondo que o elemento-base de uma *thread* fosse o quarto elemento do multiconjunto, de valor `"9"`, bastaria acessar a quarta posição do *array* (de índice = 3), para obter tal valor, lembrando que a indexação no *array* inicia em zero.

Todavia, este comportamento não se repete para os casos nos quais o multiconjunto é formado por elementos de tipos compostos (tuplas) ou por uma mistura de elementos simples e compostos. Nessas situações, o número de elementos presentes no multiconjunto é necessariamente menor que o número de posições no *array* resultante, visto que as tuplas são consideradas como um único elemento no multiconjunto, mas ocupam mais de uma posição quando são representadas sequencialmente na forma de *array*. Esta característica adiciona um nível de indireção na tarefa das *threads* de acessar corretamente os elementos, tornando-a mais dispendiosa, pois não há mais uma relação direta de indexação entre a posição do elemento no multiconjunto e a posição no *array*. Desta forma, para que as *threads* pudessem realizar o devido acesso aos elementos nesta classe de multiconjuntos, foram criados dois *arrays* auxiliares, um chamado de *offset* e o outro de *elem-size*. A partir de agora, usaremos o termo *bag-array*, para nos referirmos ao *array* utilizado para representar o multiconjunto, a fim de evitar confusões.

O *array offset* serve para armazenar os índices do *bag-array* no qual um determinado elemento se encontra, ao passo que o *array elem-size*, é utilizado para guardar os tamanhos dos elementos compostos presentes no multiconjunto. Ambos possuem tamanho igual ao número de elementos existentes no multiconjunto, os quais, ainda que sejam do tipo composto, contam como apenas um elemento. Ou seja, a indexação destes dois *arrays*, o *offset* e o *elem-size*, é realizada de maneira

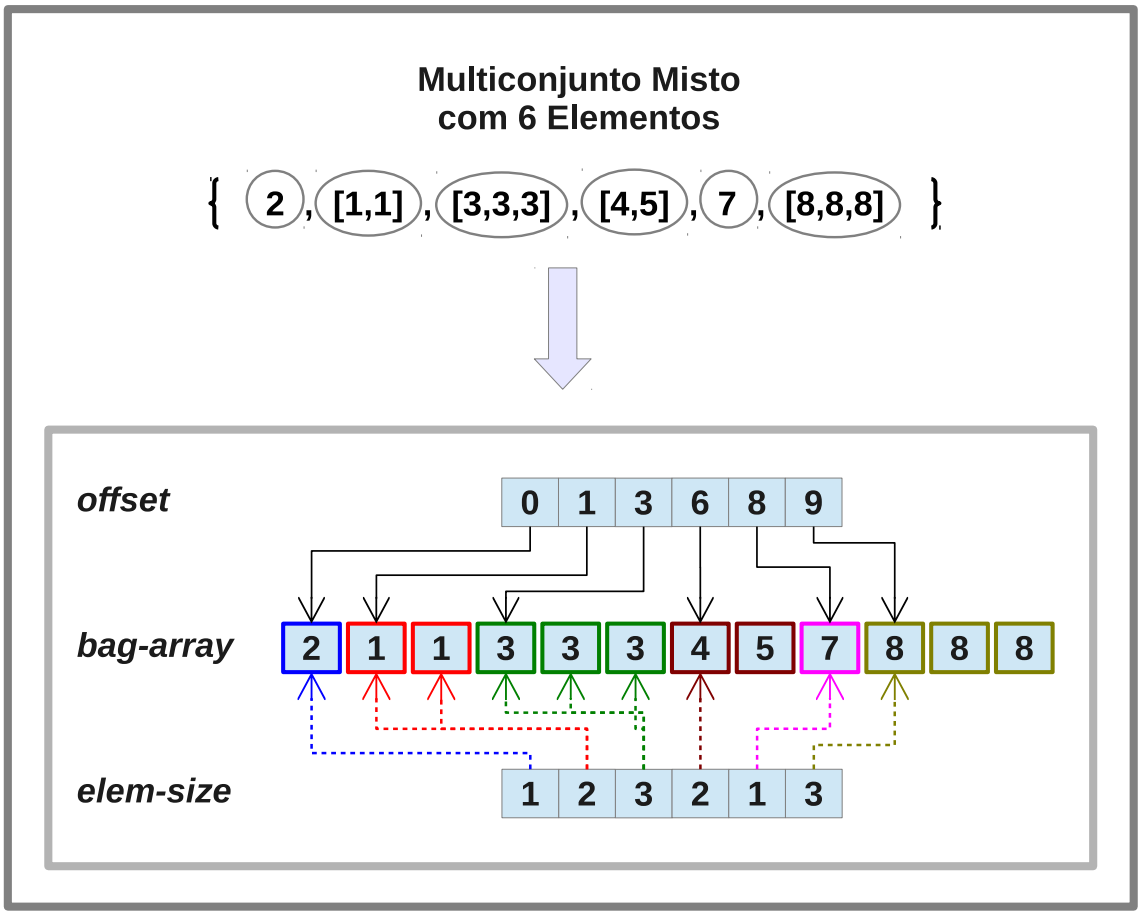


Figura 5.8: Exemplo de multiconjunto misto e dos *arrays* associados para o correto acesso aos seus elementos.

direta de acordo com a posição do elemento no multiconjunto. Em outras palavras, uma *thread* que deseje acessar um elemento do *bag-array*, deve primeiro realizar uma indexação direta no *array offset*, para obter a posição correta de armazenamento, e depois, deve realizar outra indexação direta, desta vez no *array elem-size*, para descobrir qual o tamanho do elemento que está sendo acessado, podendo assim, obter os sucessivos valores em posição e quantidade correta, para os casos de tipos compostos. A Figura 5.8 nos demonstra um exemplo de multiconjunto misto, e os *arrays* usados para sua representação principal (*bag-array*), e para permitir o acesso de maneira correta (*offset* e *elem-size*).

Suponhamos que uma *thread* quisesse acessar o quarto elemento (índice = 3) do multiconjunto mostrado, que é a tupla `"[4,5]"`. O primeiro passo seria realizar uma indexação direta no índice três do *array offset*, e armazenar este valor em uma variável para uso futuro, chamemos-na de `"idx"`:

```
offset ⟨ 0, 1, 3, 6, 8, 9 ⟩
```

```
idx = offset[3];
  ↓
idx = 6;
```

Depois, a *thread* deveria da mesma forma, indexar diretamente o *array elem-size*, obtendo o tamanho (`"tam"`) do elemento sendo buscado:

```
elem-size ⟨ 1, 2, 3, 2, 1, 3 ⟩
```

```
tam = elem-size[3];
  ↓
tam = 2;
```

Finalmente, ela poderia realizar o acesso ao elemento pretendido no *bag-array*, utilizando o índice `"idx=6"` para indexá-lo, e sabendo que o elemento possui tamanho `"tam=2"`, ou seja, dois valores discretos para serem lidos:

```
bag-array ⟨ 2, 1, 1, 3, 3, 3, 4, 5, 7, 8, 8, 8 ⟩
```

```
valor_1 = bag-array[idx];
valor_2 = bag-array[idx+1];
  ↓
valor_1 = bag-array[6];
valor_2 = bag-array[7];
  ↓
valor_1 = 4;
valor_2 = 5;
```

Na implementação realizada, além das estruturas mencionadas, a representação do multiconjunto no formato de *array* utilizou algumas outras variáveis úteis para seu funcionamento, como por exemplo, um *array* mantendo os índices dos elementos-base. A seguir podemos ver a *struct* em linguagem C que foi criada e utilizada para manter as informações do multiconjunto:

```
/* Run-Time bag-array, para processamento na GPU */
struct rt_bag_array {
  int size;               //número de valores discretos no array
  int *data;              //valores discretos do array
  int reaction_size;      //número de elementos em uma reação
  int reaction_num_vals; //número de valores discretos em uma reação
  int num_elements;       //número de elementos no array
  int *offset;            //array de offsets dos elementos
  int *elem_size;         //array de tamanho dos elementos
  int num_base;           //número de elementos-base no array
  int *ind_base;          //array de índices dos elementos-base
};
```

5.4.2.3 Estrutura de Resultado das Reações

Quando as *threads* que executam os *kernels* na GPU realizam uma verificação de condição de reação que se avalia como verdadeira, a reação pode ocorrer entre os elementos combinados. Nesse instante, o comportamento da *thread* consiste em armazenar a informação sobre quais elementos participaram da verificação bem sucedida, marcá-los como já reagidos, para que não mais possam fazer parte de outras reações em outras *threads* e interromper seu processamento. Assim que todas as *threads* na GPU terminam suas execuções, tais informações são obtidas pela CPU para dar sequência ao programa, através da execução das ações correspondentes. Para realizar esta comunicação, é necessária uma estrutura que armazene estas informações de resultado das reações, servindo como uma resposta, ou um parâmetro de saída, da GPU para a CPU. Nossa implementação resolveu utilizar uma estrutura no formato de uma matriz bidimensional para tal tarefa, de modo que as linhas representam as *threads* e as colunas os índices dos elementos que participaram de uma reação. Ou seja, o número de linhas da matriz será sempre igual ao número de *threads* criadas na chamada do *kernel*, e o número de colunas, será a aridade da reação, que é o número de elementos participantes na mesma.

Assim, a *thread* que verifica uma reação que pode ocorrer, preenche a sua linha associada na matriz com os valores dos índices dos elementos reagentes, na ordem em que aparecem na cláusula `"replace"` dos programas Gamma. Devemos lembrar que o primeiro elemento reagente testado em uma condição de reação qualquer, é sempre o elemento-base da *thread* que ora o processa, enquanto os demais, podem ser quaisquer outros elementos existentes no multiconjunto. Ou seja, a primeira coluna da linha preenchida por uma certa *thread*, será sempre o índice do seu próprio elemento-base. A Figura 5.9 mostra esta estrutura matricial de resultado das reações

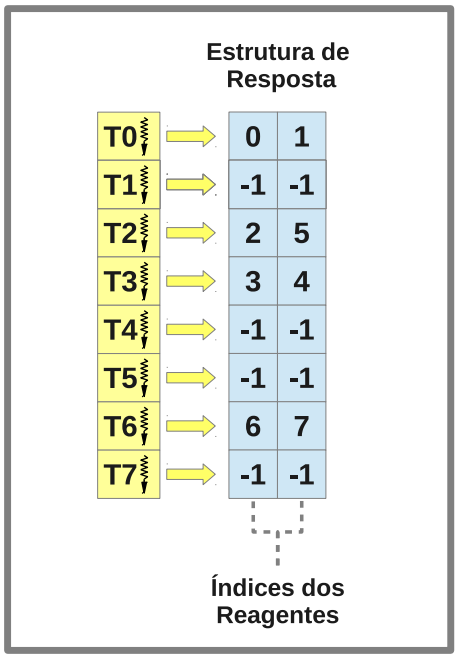


Figura 5.9: Estrutura matricial de resultado das reações da primeira iteração do programa `"maximo.gm"`.

preenchida com os valores relativos à primeira iteração (exibida na Figura 5.7) do nosso programa exemplo `"maximo.gm"`.

A estrutura de resposta é inicialmente inteira preenchida com o valor `-1"`, cujo significado é de reação não ocorrida. As *threads* que por algum motivo não verificarem reações que possam ocorrer, seja porque nenhuma combinação satisfez a condição, ou porque seus elementos-base já reagiram em outras *threads*, não preenchem sua linha correspondente na matriz de resposta, ficando a mesma com os valores padrão de inicialização, como é o caso das *threads* T1, T4, T5 e T7, da Figura 5.9. Já as linhas das *threads* T0, T2, T3 e T6, estão devidamente preenchidas com os índices dos elementos participantes das reações, respectivamente verificadas por cada uma delas.

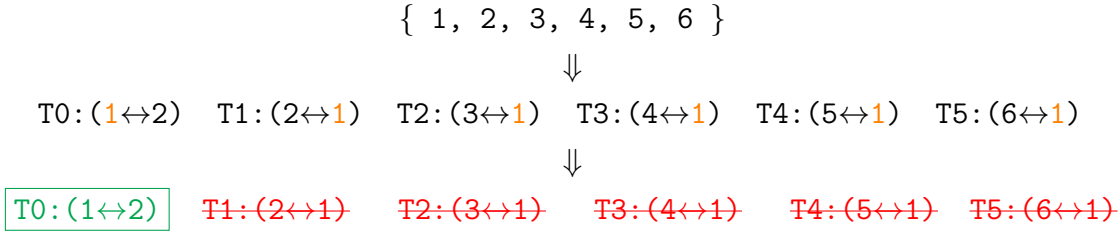
5.4.2.4 Otimizações

A semântica do formalismo Gamma tem como premissa básica estabelecer na descrição de programas o mínimo possível de restrições, sejam de controle ou de ordem de execução. Isto torna difícil a tarefa de alcançar um nível aceitável de eficiência em qualquer implementação da linguagem, pois em uma cenário de pior caso, há a necessidade de realizar as combinações entre todos os elementos do multiconjunto, causando a chamada "explosão combinatória" [51]. Contudo, essa dificuldade de implementação pode ser contornada com a adição de algumas restrições na execução do programa Gamma, as quais podem ser inseridas no código para impor um mínimo de controle sobre a ordem de execução, como no caso dos operadores de composição

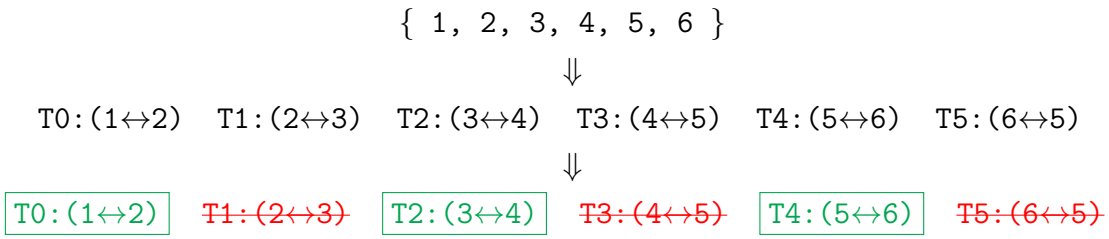
de programas, que já vimos anteriormente. Além disso, podemos também buscar inserir algum tipo de otimização nas comparações entre os elementos, visando evitar computações desnecessárias e improdutivas.

Devemos nos lembrar que a dinâmica das reações em um programa Gamma consiste em testar a condição de reação para cada possível combinação de elementos do multiconjunto, e que quanto maior a aridade da reação, maior o número de combinações possíveis. Na implementação *Gamma-Base*, este processamento é feito de forma sequencial, através de *loops* aninhados que realizam tais combinações. Já na nova implementação *Gamma-GPU*, fomos capazes de mitigar o custo computacional da "explosão combinatória", pois muitas das combinações podem ser realizadas em paralelo pelas *threads* na GPU.

Como pudemos notar na Figura 5.6, exibida na Subsubseção 5.4.2.1, a ordem das combinações testadas em uma certa *thread*, inicia sempre pelo elemento imediatamente posterior ao elemento-base, avança até o último elemento do multiconjunto, e depois regride a partir do elemento imediatamente anterior ao elemento-base, até chegar ao primeiro elemento do multiconjunto. Esta foi uma heurística implementada visando otimizar o processo dos testes das condições de reação inter-*threads*, pois potencializamos a chance de encontrar simultaneamente reações que não compartilhem um elemento comum, o que inviabilizaria uma delas de ocorrer. Se tivéssemos adotado uma ordem de comparações mais trivial, como por exemplo, todas as *threads* iniciam comparando seu elemento-base com o primeiro elemento do multiconjunto, e prosseguem sequencialmente até o último elemento, teríamos já na primeira rodada, todas as *threads* testando combinações que envolvem um elemento em comum, no caso, o primeiro do multiconjunto. Supondo que todos estes testes fossem avaliados como verdadeiros, teríamos apenas um que de fato poderia ocorrer, uma vez que um mesmo elemento, não pode participar de mais de uma reação dentro da mesma iteração de processamento. Fica claro que isto leva a uma computação mais onerosa, pois um grande número de comparações foi realizada inutilmente. Este cenário pode ser visto a seguir, no qual consideramos um multiconjunto com seis elementos, sobre o qual seis *threads* (T0 a T5) iniciariam testando uma condição de reação qualquer usando as combinações triviais, e apenas uma poderia realmente se concretizar:



Vemos que nesse caso o elemento "1" foi testado simultaneamente por todas as *threads*, e por isso, apenas uma delas, a T0 (poderia ser qualquer outra), poderá de fato concretizar a reação sendo executada. Dizemos que a eficiência da rodada de testes neste exemplo é de 2/6 (33%), visto que dois dos seis elementos do multiconjunto ("1"e "2"), puderam ser utilizados em alguma combinação de forma exitosa. Agora vejamos o mesmo exemplo com o uso da otimização implementada:



Percebemos que não há mais um elemento sendo o gargalo das comparações, e assim, podemos obter uma eficiência de 6/6 (100%), visto que todos os seis elementos do multiconjunto puderam participar de alguma combinação com sucesso.

Na prática, em um programa qualquer, pode ocorrer que algumas *threads* não sejam capazes de verificar em tempo hábil a ocorrência de uma disputa no uso de um elemento em comum, o que resultaria em reações distintas sendo validadas, ainda que compartilhassem um mesmo elemento. Este comportamento está diretamente ligado à forma de execução das *threads* no ambiente CUDA. Como vimos na Seção 3.3, o modelo de processamento adotado na arquitetura das GPUs cria um *grid* de *threads* quando um *kernel* é invocado. Essas *threads* são organizadas em blocos, que podem entre eles, executar em qualquer ordem. Já as *threads* dentro de um mesmo bloco, são executadas em grupos, chamados de *warps*, que são compostos de 32 *threads*. Em mais baixo nível, os *warps* são executados nos multiprocessadores de *streaming* através de um hardware do tipo SIMT (*Single-Instruction, Multiple-Thread*), o qual é composto de uma única unidade de controle, e várias de processamento, de modo que a mesma instrução é aplicada simultaneamente para todas as *threads* de um mesmo *warp*. Voltando ao nosso problema, isso explica o porquê de algumas *threads* não conseguirem detectar que um elemento já foi reagido em outra *thread*, pois as verificações ocorrem precisamente no mesmo instante de tempo, uma vez que só há um sinal de controle sendo enviado às múltiplas unidades de processamento, as quais executam a mesma instrução. Todavia, caso isto aconteça, tal falha semântica é corrigida em um ponto mais adiante no processamento, quando as ações sobre o multiconjunto são aplicadas, como veremos na Subseção 5.4.3.

Outra otimização que implementamos, bastante simples, está relacionada ao uso de mais de uma GPU no mesmo nó de processamento, ou seja, nos casos onde uma

mesma CPU possui duas ou mais GPUs disponíveis para colaborar na computação. Como bem sabemos, na nossa implementação *Gamma-GPU*, cada reação presente no programa Gamma executa em uma célula trabalhadora. O processo associado a esta célula é submetido para execução em um *core* de alguma CPU presente no ambiente de hardware disponível, a qual pode fazer uso de uma GPU, caso exista, para acelerar sua computação nos trechos já exibidos. É fácil perceber que em CPUs *multicore*, podemos ter múltiplos processos executando suas células trabalhadoras correspondentes, e cada um deles, poderá solicitar o auxílio de uma GPU. Nesse caso, se houver mais de uma GPU disponível no contexto da referida CPU, é interessante que possamos distribuir o uso das mesmas da maneira mais equilibrada possível, de forma que os processos não acabem todos associados a uma única GPU, causando potenciais contenções no acesso ao recurso, enquanto outras GPUs sejam subutilizadas, passando boa parte do tempo ociosas. Diante disto, criamos uma solução para associar as GPUs aos processos, baseada no número da célula trabalhadora no ambiente distribuído, como podemos ver nas linhas de código a seguir:

```
cudaGetDeviceCount(&n_gpus);
gpu = cell_number % n_gpus;
cudaSetDevice(gpu);
```

Primeiro, descobrimos quantas GPUs existem no contexto da CPU em pauta, através da chamada `"cudaGetDeviceCount"`, e depois usamos o resultado do resto da divisão do número da célula trabalhadora (`"cell_number"`) pelo número de GPUs existentes (`"n_gpus"`), para selecionar a GPU que será utilizada pelo processo, usando a chamada `"cudaSetDevice"`. O resto da divisão sempre gera um número compreendido entre zero e `"n_gpus - 1"`, que é a faixa das GPUs existentes no nó, uma vez que a numeração atribuída pelo ambiente CUDA às GPUs disponíveis inicia-se em zero. Num cenário com duas GPUs presentes, e quatro processos executando em *cores* da mesma CPU, teríamos as seguintes GPUs selecionadas em cada célula:

```
Célula 0  →  GPU = 0%2  →  GPU = 0
Célula 1  →  GPU = 1%2  →  GPU = 1
Célula 2  →  GPU = 2%2  →  GPU = 0
Célula 3  →  GPU = 3%2  →  GPU = 1
```

Como podemos notar, cada GPU fica associada a dois processos, resultando numa melhor utilização dos recursos disponíveis.

Para encerrar, devemos lembrar que as otimizações exibidas relativas ao uso das GPUs, são limitadas pela característica generalista do nosso trabalho, uma vez que o mesmo trata-se da construção de um compilador, que pode gerar código para os mais diversos tipos de problemas, expressos na linguagem Gamma. Isto contrasta com trabalhos que buscam desenvolver soluções para problemas específicos e bem definidos usando GPUs, pois nesse caso, é possível entender os fatores limitantes e as restrições de uso de recursos intrínsecos à aplicação em particular, o que possibilita implementar otimizações mais elaboradas e complexas, mas que por outro lado, se restringem unicamente à solução do problema alvo. Dependendo do nível de necessidade por melhor desempenho, algumas soluções chegam ao ponto de implementar otimizações específicas para uma determinada classe ou família de GPUs, ficando altamente dependentes do hardware usado, e consequentemente, implicando na perda de generalidade. Isso não é admissível no nosso caso, pois a proposta é que tenha-se um compilador de programas Gamma capaz de beneficiar-se do uso das GPUs, sem gerar tamanha dependência do hardware subjacente.

5.4.3 Execução das Ações e Terminação

Depois que todas as *threads* de um determinado *kernel* CUDA encerram seus processamentos na GPU, a estrutura contendo os resultados das reações testadas já encontra-se devidamente preenchida, e armazenada em espaço de memória do *device*. Neste momento, o controle do fluxo de execução retorna para a CPU, e entramos na fase chamada de pós-processamento de invocação do *kernel*, a qual consiste em realizar a cópia da estrutura de resposta da memória do *device* para a memória do *host*, seguida da liberação de espaço de memória da GPU, como podemos ver nas chamadas CUDA a seguir:

```
cudaMemcpy(h_response, d_response, num_elements_base *
           RunTimeBagArray->reaction_size * sizeof(int),
           cudaMemcpyDeviceToHost);

cudaFree(d_array_data);
cudaFree(d_response);
```

O último parâmetro da chamada `"cudaMemcpy"` destaca o sentido da cópia entre as memórias, que agora, ocorre da GPU (*device*) para a CPU (*host*). São então

liberadas todas as memórias na GPU, como o *array* que armazenava o multiconjunto e a matriz usada para guardar a estrutura de respostas (além de outras não exibidas, como por exemplo as estruturas adicionais para tratamento a tipos compostos).

De posse da estrutura de respostas, a CPU pode dar continuidade à dinâmica de processamento, entrando na fase de execução das ações correspondentes às reações que tiveram suas condições de reação avaliadas como verdadeiras pelas *threads* da GPU. Esta é o momento no qual ocorre a reescrita do multiconjunto, pois o mesmo sofre modificações através da remoção dos elementos que reagiram, e da adição de novos elementos que são criados pelas ações. De fato, este é o comportamento expresso pelo par de cláusulas `"replace"` e `"by"`, presentes obrigatoriamente em todo programa expresso na linguagem Gamma. Sabemos que um determinado elemento do multiconjunto não pode ter reagido mais de uma vez na mesma chamada a um *kernel*, ou seja, não deve aparecer mais de uma vez na matriz de respostas. Contudo, em algumas situações particulares, podem ocorrer certas falhas semânticas durante o processamento das reações nas *threads* da GPU, fazendo que a matriz de respostas contenha duplicações errôneas, conforme explicamos na Subsubseção 5.4.2.4, que tratou das otimizações.

Para contornar este problema, e garantir que a CPU possa aplicar as ações devidas sobre o multiconjunto, mantendo a consistência e corretude do mesmo, implementamos neste ponto uma simples conferência sobre os elementos que já reagiram uma vez, não permitindo que tomem parte em quaisquer outras reações. Usamos para tanto, um *array* de tamanho igual ao número de elementos existentes no multiconjunto, de modo que cada posição representa um elemento, e pode ser preenchida com um valor *booleano* para indicar se este elemento já reagiu. Na realidade, esta mesma ideia foi utilizada no contexto das *threads* da GPU, para que pudessem controlar quais elementos já haviam reagido em outras *threads*, porém, como foi exposto, este mecanismo não foi capaz de resolver o caso da execução das *threads* de um mesmo *warp*, uma vez que a verificação sobre o *array* ocorre simultaneamente por todas as *threads*, pois todas são controladas por um único sinal de controle. Tal comportamento é corrigido na etapa descrita nesta seção, pela CPU, através do *array* de valores *booleanos*, conforme citado previamente.

Finalmente, após ter aplicado as ações sobre o multiconjunto, a CPU encerra a iteração corrente do processamento, devolvendo o mesmo para a *Bag-Cell*, o que irá possibilitar outros processos de células trabalhadoras executarem suas respectivas reações, dando sequência à computação do resultado do programa Gamma. A terminação do programa ocorre quando nenhuma reação pode ocorrer em nenhum processo de célula trabalhadora existente, ou seja, chegou-se a um estado estável do multiconjunto que não pode mais ser modificado. Nesta ocasião, a célula Controladora Principal envia mensagens no padrão MPI sinalizando todas as células

do ambiente distribuído para encerrarem seus processamentos, e o multiconjunto estável resultante é a resposta final do programa Gamma.

5.5 Análise de Complexidade

Uma característica muito importante de qualquer algoritmo é o seu tempo de execução, que pode ser obtido naturalmente de forma empírica através da sua execução e medida do tempo gasto sobre uma determinada arquitetura em particular (utilizaremos esta abordagem no Capítulo 6). Por outro lado, existem também métodos analíticos que possibilitam a obtenção de uma ordem de grandeza do tempo de execução do algoritmo, via determinação de expressões matemáticas que traduzam seu comportamento. A vantagem do método analítico é que ele nos provê uma aferição de tempo independente de fatores específicos da arquitetura de hardware, linguagens de programação, ou compiladores utilizados. Para medir o tempo de um algoritmo analiticamente, usamos o número de passos efetuados pelo mesmo, que pode ser interpretado como o número de execuções de uma determinada operação nele constante, geralmente sendo eleita uma operação principal como dominante para a medida. A complexidade de tempo de pior caso corresponde ao número de passos que o algoritmo efetua no seu pior caso de execução, ou seja, no cenário mais desfavorável de entrada. A representação da complexidade de tempo de um algoritmo pode então ser descrita com o auxílio de um operador matemático, que nos dá um limite superior para valores assintóticos da função que representa o número de passos do mesmo [52]. Tal operador é descrito pela chamada notação O, a qual será utilizada para as análises procedidas nesta seção.

Pois bem, faremos agora a análise de complexidade do algoritmo que realiza as possíveis combinações entre os elementos do multiconjunto, comparando as duas implementações já citadas, ou seja, a *Gamma-Base* e a *Gamma-GPU*. Como sabemos, no pior caso, para que se conclua que nenhuma condição é satisfeita para uma certa reação, todas as combinações entre os elementos do multiconjunto devem ser testadas, e quanto maior a aridade da reação, maior o número de combinações possíveis. Assim, a operação dominante para a contagem do número de passos do algoritmo neste caso, será a operação de combinação entre elementos do multiconjunto, ou seja, o número total de combinações efetuadas será a variável que nos dará a complexidade de tempo do algoritmo. Considere as seguintes variáveis:

$\mathbf{NC_{base}}$: número de combinações da *Gamma-Base*
$\mathbf{NC_{gpu}}$: número de combinações da *Gamma-GPU*
$\mathbf{NC_{thread}}$: número de combinações de uma *thread*
n : número de elementos no multiconjunto (tamanho da entrada)
r : aridade da reação (número de elementos reagentes)
CC : número de *CUDA-Cores* para processamento paralelo

1ª Análise: *Gamma-Base*

Na implementação *Gamma-Base*, as combinações de uma determinada reação são todas realizadas por uma única linha de execução, de maneira sequencial, através de *loops* aninhados. O número de níveis de *loops* (ou profundidade do *loop*) é baseado na aridade da reação, sendo criado um nível para cada elemento reagente. Considerando um multiconjunto com n elementos, e uma reação de aridade r, podemos aplicar a Equação 5.1 para cálculo de um Arranjo Simples e obter o número de combinações de pior caso, uma vez que o Arranjo Simples nos dá o número existente de combinações de n elementos tomados r a r.

$$A_r^n = \frac{n!}{(n-r)!} \tag{5.1}$$

Assim, temos:

$$\begin{aligned} NC_{base} &= \frac{n!}{(n-r)!} \\ &= \frac{n \times (n-1) \times (n-2) \times \ldots \times (n-(r-1)) \times \cancel{(n-r)!}}{\cancel{(n-r)!}} \\ &= n \times (n-1) \times (n-2) \times \ldots \times (n-(r-1)) \\ &= \prod_{i=0}^{r-1} (n-i) \end{aligned}$$

Considerando valores muito grandes de n e também sabendo que $n \lll r$, todas as r parcelas (0 até $r-1$) do produtório podem ser aproximadas para n, o que nos dá:

$$NC_{base} \simeq n^r \tag{5.2}$$

Ou seja, a complexidade de tempo de pior caso é da ordem de n^r:

$$\boxed{NC_{base} = O(n^r)} \tag{5.3}$$

1ª Análise: *Gamma-GPU*

Na implementação *Gamma-GPU*, as combinações de uma reação não são mais realizadas de forma sequencial por uma única linha de execução, pois o novo modelo permite que o trabalho seja dividido para execução paralela em múltiplas *threads*, que executam nas centenas de *CUDA-Cores* presentes nas GPUs. Considerando o mesmo cenário de um multiconjunto com n elementos, e uma reação de aridade r, o que ocorre agora é que cada *thread* fica responsável por realizar apenas as combinações que possuem seu elemento-base envolvido, de modo que seus *loops* possuem um nível a menos que a aridade da reação, uma vez que um dos reagentes é fixo. Assim, cada *thread* realiza no pior caso a combinação de $n-1$ elementos tomados de $r-1$ em $r-1$ (A_{r-1}^{n-1}), o que nos dá:

$$\begin{aligned}
NC_{thread} &= \frac{(n-1)!}{(n-1)-(r-1)!} \\
&= \frac{(n-1)\times((n-1)-1)\times\ldots\times((n-1)-(r-2))\times \cancel{(n-1)-(r-1)!}}{\cancel{(n-1)-(r-1)!}} \\
&= (n-1)\times((n-1)-1)\times\ldots\times((n-1)-(r-2)) \\
&= \prod_{i=0}^{r-2}((n-1)-i)
\end{aligned}$$

Usando novamente o mesmo raciocínio, para valores muito grandes de n e também sabendo que $n \ggg r$, todas as $r-1$ parcelas (0 até $r-2$) do produtório podem ser aproximadas para n, o que nos dá:

$$NC_{thread} \simeq n^{r-1} \tag{5.4}$$

Agora que já temos o número de comparações que cada *thread* realiza individualmente, precisamos obter o número total de combinações de todas as *threads* somadas. Devemos ressaltar que como estamos interessados em obter complexidade de tempo, as combinações realizadas em paralelo por *threads* distintas são contadas como apenas uma combinação, pois são passos que ocorrem simultaneamente gastando apenas uma unidade de tempo. Assim sendo, é neste ponto que o paralelismo

do modelo entra em ação, pois as *threads* podem executar simultaneamente sobre o hardware da GPU, até o limite do número de *CUDA-cores* existentes (variável CC). Ou seja, das n *threads* existentes (uma para cada elemento do multiconjunto), executam de fato em paralelo apenas CC delas, devido à limitação de recursos de hardware, e desta forma, são necessárias várias execuções em grupos de CC *threads* para finalizar o processamento. É fácil perceber que o número de execuções de grupos de *threads* (chamemos $nExec$) é o quociente da divisão do número total de *threads* pelo tamanho do grupo que pode executar em paralelo:

$$nExec = \frac{n}{CC} \tag{5.5}$$

Assim, podemos obter o número total de combinações de *Gamma-GPU*, em termos de complexidade de tempo, multiplicando o número de execuções de grupos de *threads* pelo trabalho de cada *thread*:

$$NC_{gpu} = nExec \times NC_{thread} \tag{5.6}$$

Logo:

$$\begin{aligned} NC_{gpu} &= \frac{n}{CC} \times n^{r-1} \\ &= \frac{n^r}{CC} \end{aligned}$$

Vemos que o número de comparações na abordagem *Gamma-GPU* ficou similar ao da *Gamma-Base*, mas dividido pelo número de *cores* existentes no hardware das GPUs. Na prática, o número de *cores* nas GPUs atuais costuma chegar na casa das centenas e até mesmo milhares de núcleos, mas ainda assim, é um número bem menor que o tamanho da entrada, a qual estamos considerando muito grande, em consonância com o viés assintótico da análise. Por isso, podemos considerá-lo como uma constante e retirá-lo da complexidade de tempo final, que fica então:

$$\boxed{NC_{gpu} = O(n^r)} \tag{5.7}$$

Podemos notar que as equações 5.3 e 5.7 são equivalentes em termos assintóticos, ou seja, quando o tamanho da entrada cresce muito, percebemos que as duas implementações tem a mesma ordem de complexidade de tempo de pior caso. Este resultado não parece muito animador, por isso vamos detalhar um pouco mais o impacto prático dos múltiplos *cores* presentes nas GPUs sobre a complexidade de tempo do algoritmo de comparações, procedendo a uma segunda etapa de análises.

2ª Análise: *Gamma-Base*

Nesta segunda fase de análise, vamos tentar refinar um pouco mais o resultado da complexidade de tempo obtida. Para isso, iremos considerar o tamanho da entrada n em formato de notação científica de expoente k:

$$n = 1O^k \tag{5.8}$$

Agora, pretendemos buscar um expoente x que seja mais preciso para representar o número de comparações em função do tamanho da entrada n (ou 10^k):

$$NC_{base} = n^x \tag{5.9}$$

Assim, substituindo a Equação 5.8 na 5.2, e igualando à 5.9, podemos resolver a equação exponencial e obter o expoente:

$$n^x = (10^k)^r$$

$$\cancel{(10^k)^x} = \cancel{(10^k)^r}$$

$$x = r$$

Logo:

$$\boxed{NC_{base} = n^r} \tag{5.10}$$

Como podemos notar, no caso da *Gamma-Base* acabamos obtendo o mesmo resultado da primeira análise, ou seja, a ordem de complexidade de tempo de pior caso continua sendo $O(n^r)$.

2ª Análise: *Gamma-GPU*

Seguindo a mesma ideia de refinamento, vamos buscar agora um expoente x que seja mais preciso para o caso da implementação *Gamma-GPU*:

$$NC_{gpu} = n^x \tag{5.11}$$

Primeiro, vamos refinar a Equação 5.4, que nos informa o número de comparações de cada *thread*:

$$NC_{thread} \simeq n^{r-1}$$

$$\simeq (10^k)^{r-1}$$

$$NC_{thread} \simeq 10^{kr-k} \tag{5.12}$$

Agora refinaremos a Equação 5.5, que nos informa o número de execuções de grupos de *threads*. O número de *CUDA-Cores* representado anteriormente pela variável CC, passará a ser expresso também em forma de notação científica, com expoente cc:

$$nExec = \frac{n}{CC}$$

$$= \frac{10^k}{10^{cc}}$$

$$nExec = 10^{k-cc} \tag{5.13}$$

Substituindo as equações 5.12 e 5.13 na 5.6, obtemos:

$$NC_{gpu} = 10^{k-cc} \times 10^{kr-k}$$

$$= 10^{\cancel{k}-cc+kr-\cancel{k}}$$

$$NC_{gpu} = 10^{kr-cc} \tag{5.14}$$

Podemos agora igualar as equações 5.11 e 5.14, para obtermos o expoente x:

$$n^x = 10^{kr-cc}$$

$$(10^k)^x = 10^{kr-cc}$$

$$\cancel{10}^{kx} = \cancel{10}^{kr-cc}$$

$$kx = kr - cc$$

$$\boxed{x = \frac{kr - cc}{k}} \tag{5.15}$$

A equação 5.15 nos permite obter o expoente para a complexidade de tempo da implementação *Gamma-GPU* de forma mais precisa, em função do tamanho da entrada, da aridade da reação, e do número de *CUDA-Cores* existentes. Quando o tamanho da entrada tende a infinito, a complexidade continua sendo $O(n^r)$, como

constatamos a seguir, através da obtenção do limite da função x:

$$\begin{aligned} x &= \lim_{k\to\infty}\left(\frac{kr-cc}{k}\right) \\ &= \lim_{k\to\infty}\left(r\times\left(\frac{k}{k}-\frac{cc}{kr}\right)\right) \\ &= r\times\lim_{k\to\infty}\left(1-\left(\frac{cc}{kr}\right)\right) \\ &= r\times\left(\lim_{k\to\infty}(1)-\lim_{k\to\infty}\left(\frac{cc}{kr}\right)\right) \\ &= r\times(1-0) \\ x &= r \end{aligned} \tag{5.16}$$

Como esperado, o expoente realmente tende a r quando a entrada tende ao infinito, o que nos dá o mesmo resultado da primeira análise. Contudo, valendo-se da Equação 5.15, podemos proceder algumas demonstrações para casos reais, supondo valores para o tamanho de entrada, a aridade da reação, e o número de *cores*. Verifiquemos o que ocorre para alguns tamanhos de entrada, fixando a aridade da reação em três ($r = 3$), e o número de *CUDA-Cores* em mil ($CC = 1000$, e $cc = 3$ pois $CC = 10^{cc}$), que são valores bastante razoáveis e condizentes com a realidade. Para efeito de comparação, devemos relembrar que para uma reação de aridade igual a três, a complexidade da implementação *Gamma-Base*, recorrendo à Equação 5.3, seria igual a $O(n^3)$ para qualquer tamanho de entrada. Vejamos para a implementação *Gamma-GPU*:

Entrada 1: k = 12

Nesta entrada, $n = 10^{12}$ elementos, temos:

$$\begin{aligned} x &= \frac{kr-cc}{k} \\ &= \frac{12\times 3-3}{12} \\ x &= 2,75 \end{aligned}$$

Donde, pela Equação 5.11, temos:

$$\boxed{NC_{gpu} = n^{2,75}}$$

Portanto, a complexidade de tempo no pior caso para esta configuração na implementação *Gamma-GPU* é $O(n^{2,75})$, contra $O(n^3)$ da implementação *Gamma-Base*. Ou seja, mesmo para uma entrada ainda muito grande, já percebemos a vantagem obtida pela exploração do paralelismo sobre as GPUs.

Entrada 2: k = 6

Nesta entrada, $n = 10^6$ elementos, temos:

$$x = \frac{kr - cc}{k}$$

$$= \frac{6 \times 3 - 3}{6}$$

$$x = 2,5$$

Donde, pela Equação 5.11, temos:

$$\boxed{NC_{gpu} = n^{2,5}}$$

Portanto, a complexidade de tempo no pior caso para esta configuração na implementação *Gamma-GPU* é $O(n^{2,5})$, contra $O(n^3)$ da implementação *Gamma-Base*. Podemos perceber que com a entrada menor que no exemplo anterior, porém ainda bastante expressiva em termos de tamanho de problemas reais, a vantagem do uso dos *many-cores* das GPUs sobressaiu-se ainda mais.

Entrada 3: k = 3

Nesta entrada, $n = 10^3$ elementos, temos:

$$x = \frac{kr - cc}{k}$$

$$= \frac{3 \times 3 - 3}{3}$$

$$x = 2$$

Donde, pela Equação 5.11, temos:

$$\boxed{NC_{gpu} = n^2}$$

Portanto, a complexidade de tempo no pior caso para esta configuração na im-

plementação *Gamma-GPU* é $O(n^2)$, contra $O(n^3)$ da implementação *Gamma-Base*. Neste último exemplo, supusemos uma entrada pequena, de apenas mil elementos, o que acarretou num ganho de complexidade de uma ordem de magnitude no expoente, pois enquanto na implementação *Gamma-Base* temos uma complexidade cúbica, na *Gamma-GPU* obtivemos uma complexidade quadrática. Um detalhe interessante neste caso é perceber que o número de elementos da entrada é igual ao número de *CUDA-Cores* existentes, o que naturalmente implica em uma execução totalmente paralela, pois os recursos de hardware foram suficientes para atender todas as *threads* criadas simultaneamente.

5.6 Trabalhos Correlatos

Até o momento da escrita desta dissertação foi encontrado apenas um trabalho que tenha utilizado, de alguma forma, a arquitetura e capacidade de processamento das GPUs, aliado ao uso do paradigma Gamma de programação. Os demais trabalhos correlatos se relacionam com o nosso de modo parcial, pois, ou realizam implementações de Gamma em arquiteturas que não contemplam as GPUs, ou são propostas de abstrações para programação de GPUs que nada tem a ver com o paradigma Gamma. Desta forma, tais trabalhos serão divididos em duas subseções distintas. A primeira, expõe trabalhos que realizaram implementações do formalismo Gamma e suas extensões, e a segunda, trabalhos que buscaram desenvolver abstrações para a programação de GPUs. Mas antes disso, falaremos sobre o único trabalho encontrado que mais se aproxima da proposta do nosso modelo *Gamma-GPU*.

O referido trabalho [53] tem como principal proposição a implementação de uma extensão *high-order* de Gamma, chamada de *Gamma Calculus* (ou γ-*calculus*), utilizando um *middleware* da IBM baseado em linguagem Java, o *TSpaces*. Tal ferramenta consiste em um modelo de comunicação cliente/servidor baseado em um "espaço de tuplas", que foi inicialmente desenvolvido para suportar o modelo de computação da linguagem Linda, porém adequou-se bem à semântica operacional do modelo de reações químicas de Gamma, segundo os autores. Foram demonstrados no trabalho técnicas de conversão de programas escritos em γ-*calculus* (através de dois exemplos: maior elemento e *sort*) para o ambiente *TSpaces*, focando principalmente em problemas de sincronização, e alguns experimentos foram realizados em um *cluster*. Foi defendida a ideia de que o desenvolvimento de um sistema de conversão automático é exequível, mas não foi implementado.

A segunda parte do trabalho (que não possui relação com sua primeira e principal parte), é a que possui maior afinidade com a presente dissertação, consistindo na proposta de implementar programas em Gamma utilizando a API do *OpenCL*,

de modo a explorar o processamento das GPUs em um ambiente heterogêneo. Foram exibidas duas implementações de programas exemplo, um para achar o maior elemento do multiconjunto, e outro para somar os elementos do multiconjunto. Sem maiores detalhes da arquitetura subjacente, a ideia foi basicamente realizar sucessivas invocações do *kernel* na GPU, realizando as operações em paralelo, até que se atingisse a resposta do programa. Os autores afirmam que experimentos mostraram um desempenho promissor do modelo, mas não foram exibidos resultados de tempo nem comparações.

Apesar de ter proposto a integração entre o paradigma Gamma e o uso de GPUs, o referido trabalho fez apenas um estágio inicial da solução, e limitou-se a realizar duas implementações específicas de programas exemplo, de modo manual, ou seja, foi adequada a ideia do comportamento de um programa Gamma usando diretamente a linguagem *OpenCL* para modelar os *kernels*. Características do código como número de parâmetros do *kernel*, além da dinâmica das reações de acordo com a aridade das mesmas, foram expressas de forma fixa, compondo uma solução voltada apenas para os exemplos em particular. Ao contrário disto, o nosso modelo *Gamma-GPU* possui um escopo bem mais amplo de atuação, pois trata-se de um compilador de programas escritos diretamente em linguagem Gamma, o qual gera automaticamente código em CUDA compatível para execução e exploração do paralelismo das GPUs. Nossa implementação livra o programador da necessidade de conhecer os detalhes de baixo nível das GPUs, e fornece esta camada transparente de abstração, permitindo que o mesmo expresse seus programas naturalmente através do paradigma Gamma. Podemos dizer que o nosso trabalho realizou os demais estágios da solução, oferecendo uma real integração entre Gamma e GPUs de maneira genérica e transparente, sendo portanto, a primeira implementação conhecida deste tipo.

5.6.1 Relacionados a Implementações do Paradigma Gamma

Na Seção 2.4 já vimos alguns exemplos de implementações do paradigma Gamma realizadas, que são correlatas a este trabalho. Veremos agora mais alguns trabalhos que propuseram diferentes implementações para Gamma.

Em [20], foram investigados vários esquemas de implementação de operações sobre multiconjuntos em arquiteturas MIMD (*Multiple-Instruction, Multiple-Data*). Algumas técnicas foram demonstradas para buscar aumentar a eficiência de algoritmos relacionados aos problemas típicos de Gamma. Os experimentos foram executados em uma arquitetura de memória compartilhada, com uma máquina dotada de seis processadores simétricos, e compararam problemas com multiconjuntos peque-

nos, de apenas seis elementos, com multiconjuntos grandes, de dez mil elementos, explorando os impactos no desempenho obtido. Esta não foi uma implementação de Gamma propriamente dita, mas explorou técnicas diretamente aplicáveis sobre o multiconjunto, que é a principal estrutura no modelo de reações químicas.

Já em [54], foram exibidos dois modelos síncronos de execução para Gamma, e discutiu-se como implementá-los em uma máquina SIMD massivamente paralela (*MasPar MP-1*). Apenas reações binárias foram consideradas, e a comunicação entre os processadores foi restrita apenas entre seus vizinhos imediatos. A ideia foi distribuir um elemento por processador, que então realiza os seguintes passos:

- testa a condição de reação entre o par de elementos formado pelo seu elemento local e pelo elemento de um de seus vizinhos;
- se a condição é verdadeira, aplica a ação e atribui os elementos gerados a outros processadores; se a condição é falsa, faz uma nova comparação com o elemento do outro vizinho, ou troca os elementos com os vizinhos.

O principal desafio investigado foi como realizar todas as comparações no multiconjunto através de dois algoritmos voltados para execução na máquina SIMD. Foram conduzidos alguns experimentos e os resultados foram comparados com os trabalhos de [19], [18] e [20]. A conclusão foi de que a implementação SIMD no *MasPar MP-1* obteve vantagem sobre as demais, e que outras aplicações em Gamma poderiam ser eficientemente implementadas no modelo proposto. A grande limitação do referido trabalho é o fato de ter considerado apenas reações binárias, além de outras restrições de tipo dos elementos nos argumentos da condição de reação e da ação resultante, em contraste com o nosso trabalho, que não impõe tais barreiras. Além disso, tal implementação foi feita diretamente na linguagem suportada pelo *MasPar MP-1*, não sendo portanto, uma abordagem genérica como o nosso compilador de programas Gamma.

No trabalho desenvolvido por [55], foi proposto um modelo de implementação paralela e distribuída de Gamma chamada *Global Gamma*, no qual a ideia foi utilizar como nós de processamento, os computadores com tempo ocioso conectados na internet (*Global Computing*). A proposta trata de uma solução completa para criar e executar programas em Gamma sobre o ambiente distribuído, através da criação de um compilador que processa um código em Gamma e gera código em Java para execução sobre a rede de computadores. Foram abordados temas como segurança, tolerância a falhas, e escalabilidade no ambiente distribuído, focando principalmente no escalonamento do multiconjunto para o processamento das reações. A ideia básica foi dividir o multiconjunto, enviando cada pedaço para um nó de processamento, que executa todas as possíveis combinações entre os elementos recebidos.

O referido trabalho tem um certo grau de semelhança com a implementação que utilizamos como base para o nosso modelo, pois propõe um escalonador para programas Gamma. Foram consideradas reações n-árias, mas não foram considerados programas compostos com mais de uma reação, como é o caso do nosso trabalho. Além disso, os autores concluem o trabalho dizendo que estavam ainda em processo de implementação da ideia, ou seja, não foram apresentados resultados de nenhum experimento prático sobre o modelo.

5.6.2 Relacionados a Abstrações para Programação de GPUs

Do mesmo modo que nossa implementação *Gamma-GPU* tem uma vertente que visa prover uma abstração para o uso de GPUs, possibilitando que a programação seja expressa em uma linguagem de alto nível de forma mais natural sem a necessidade de que se conheçam detalhes de baixo nível do hardware, inúmeros outros trabalhos também propõem abstrações nesse sentido, visando oferecer ao programador uma interface mais amigável para acesso a estes dispositivos. Isto porque as linguagens mais tradicionais, CUDA e OpenCL, para programação de GPUs, requerem um nível considerável de conhecimento de baixo nível de técnicas para movimentação de dados entre memórias, sincronização e escalonamento da computação. Na maioria dos casos, os trabalhos que veremos a seguir utilizam CUDA ou OpenCL somente como linguagens intermediárias, exibindo ao usuário apenas uma linguagem mais abstrata e sintaticamente mais simples, poupando-o de detalhes que não estão diretamente ligados à resolução do problema em si.

Em [56] é apresentada uma linguagem chamada CGiS, que visa aumentar a acessibilidade às GPUs através de um elevado nível de abstração. O foco principal foi permitir que programadores de problemas científicos, sem experiência em manipulação de GPUs, pudessem se beneficiar do ganho de desempenho oferecido de forma transparente. A linguagem permite que se escreva o algoritmo completo, que é então submetido a um compilador que gera código em C++ e diretamente em *assembly* para GPUs. Um programa em CGiS é composto por três seções: INTERFACE na qual são declaradas variáveis escalares e de fluxo; CODE onde definem-se as funções sobre os fluxos; e CONTROL que define a interação entre as funções.

O trabalho descrito em [57], introduz também uma nova linguagem para programação de GPUs, chamada EPGPU (*Expressive Programming for GPU*), a qual utilizou OpenCL embutido no ambiente de programação de C++. Uma das principais abstrações providas é chamada de FILL *kernel*, que permite que as escritas na memória da GPU sejam feitas de modo controlado em tempo de execução, eliminando a chance de aparecimento de problemas como condições de disputa a regiões

de memória compartilhada.

Uma linguagem de programação declarativa para GPUs foi proposta em [58], batizada de *Harlan*. A principal ideia é que o programador possa expressar apenas o que fazer, e não como fazer. A linguagem foi integrada com um outro trabalho prévio dos mesmos autores, possibilitando seu uso em *clusters* de GPUs. O compilador construído para a linguagem traduz um programa em *Harlan* para código CUDA ou OpenCL, e suporta algumas facilidades sintáticas, como a determinação automática da configuração de blocos e *threads* para um certo *kernel*, divisão de um único *kernel* em *Harlan* para vários *kernels* na linguagem de mais baixo nível, além de prover um escalonamento mais eficiente entre *kernels* de acordo com um grafo ponderado de dependência de dados dos mesmos.

Em [59] foi criada uma linguagem para domínios específicos chamada *Chestnut* para programação de GPUs, voltada para aplicações sobre dados no formato de *arrays* multidimensionais (*grids*), como por exemplo soluções finitas de equações diferenciais parciais, aplicáveis em problemas como modelagem climática, dinâmica dos fluidos e difusão de calor. O compilador *Chestnut* foi escrito em *Python* e traduz o código de alto nível para código CUDA. A sintaxe para programar em *Chestnut* é similar à linguagem C, e a ideia foi criar uma forma muito simples de indicar os trechos de código que devem executar em paralelo nas GPUs, bastando envolvê-los pelas declarações *foreach* e *end*, que delimitam as operações a serem aplicadas sobre um conjunto de *arrays* em paralelo. Alguns experimentos mostraram que programas escritos em *Chestnut* obtiveram um desempenho bem próximo àqueles escritos diretamente em CUDA.

Na proposta contida em [60], a ideia foi criar uma abstração baseada no paradigma orientado a objetos para acesso às GPUs, através de uma nova linguagem chamada de *Fusion*, a qual é uma extensão sintática sobre a linguagem Java. A principal destas extensões é a possibilidade de criar objetos aceleradores, instâncias de uma classe aceleradora, os quais terão seus métodos executados pelas GPUs. O compilador *Fusion* gera dois códigos separados como saída, um em Java puro, que contém as chamadas aos *kernels*, e o outro em CUDA, contendo os *kernels* que são as implementações dos métodos dos objetos aceleradores.

Outros trabalhos que seguem linhas similares podem ser vistos em [61], [62], [63] e [64].

Capítulo 6

Experimentos e Resultados

Depois de termos discorrido nos capítulos anteriores sobre as soluções de implementação envolvidas no trabalho, vamos agora dedicar o presente capítulo para demonstrar os experimentos práticos realizados com as mesmas, nos quais almejamos verificar a corretude da nova implementação *Gamma-GPU*, bem como analisar o desempenho do modelo quando executado em uma plataforma distribuída de hardware.

6.1 Metodologia Experimental

Utilizamos uma metodologia empírica para proceder os testes do trabalho, na qual foram selecionadas algumas aplicações escritas em Gamma para serem executadas sobre as implementações *Gamma-Base* e *Gamma-GPU*, a fim de que pudéssemos comparar as respostas dos programas e também medir o tempo total de execução dos mesmos. A comparação entre as respostas de cada programa nas duas implementações, serve para nos garantir que nossa nova implementação *Gamma-GPU* comporta-se de maneira adequada, emitindo resultados corretos para os problemas processados com o auxílio das GPUs. Já a medida dos tempos de execução, nos ajuda a proceder uma análise comparativa de desempenho entre as duas implementações, demonstrando o poder de aceleração de processamento inserido com o uso das GPUs.

6.1.1 Aplicações de Teste

Foram selecionadas para a realização dos experimentos cinco aplicações de teste (*benchmarks* de aplicação) a serem executadas, cada uma com características próprias, de modo a podermos avaliar de forma mais completa o comportamento das implementações. Vejamos a seguir quais foram estas aplicações e uma breve explicação sobre a ideia de cada uma delas:

1 - Números Primos

/* primos.gm */

```
primos { 2,3,4,...,N }
where
primos = replace x1, x2
         by x2
         if ((x1 % x2) == 0)
```

Este programa processa o multiconjunto eliminando todos os números entre `"2"` e `"N"` que possuem divisores, restando no final como resposta, apenas os números que são primos.

2 - Algoritmo de Ordenação

/* sort.gm */

```
init ; sort { N elementos }
where
init = replace x
       by [0,x]
       if true
sort = replace [i,x], [j,y]
       by [i,x], [j+1, y]
       if (x <= y and i == j)
```

Neste caso, temos um programa Gamma que realiza a ordenação (*sort*) dos elementos do multiconjunto. A primeira reação transforma todos os elementos simples, em tuplas de tamanho dois, nas quais o primeiro elemento representa o índice de ordenação (inicialmente todos os índices recebem o valor zero), e o segundo elemento é o próprio valor que estava no multiconjunto inicial. A segunda reação realiza a alteração dos índices das tuplas, de acordo com o valor de seus elementos. No final, o multiconjunto resultante consiste em um conjunto de tuplas de dois elementos cada, que estão ordenadas pelos valores de seu segundo elemento, seguindo a ordem sequencial de seus índices: $\{\ [0, x_1], [1, x_2], \ldots, [(N-1), x_n]\ \}\ \mid\ (x_1 \leq x_2 \leq \ldots \leq x_n)$.

3 - *Fibonacci*

/* fibonacci.gm */

```
gera ; soma { N }
where
gera  = gera1 | gera2
gera1 = replace n
        by n-1, n-2
        if (n > 1)
gera2 = replace n
        by 1
        if (n == 0)
soma  = replace x,y
        by x+y
        if true
```

Este programa calcula o n-ésimo termo da série de *fibonacci*. O termo que se deseja calcular é a entrada inicial do multiconjunto, ou seja, o valor `"N"`. Existem duas reações conectadas pelo operador sequencial `";"`, `"gera"` e `"soma"`. A primeira se divide em duas outras reações conectadas pelo operador paralelo `"|"`, `"gera1"` e `"gera2"`, as quais realizam a expansão do multiconjunto através da decomposição do termo inicial, até que restem apenas valores iguais a `"1"`. Depois, a reação `"soma"` realiza o somatório no multiconjunto, obtendo a resposta do programa.

4 - Fusão de Dados para Acompanhamento de Alvos

/* fusao.gm */

```
R1;R2;R3;R4;R5;R6;R7;R8;R9;R10;R11;R12;R13;R14;R15;R16
{ N tuplas sensor 1, N tuplas sensor 2, ..., N tuplas sensor Y }
where

R1 =   replace [ids,ida,idb],[ids1,ida1,idb1]
       by      [ids1,ida1,idb1]
       if      ids==ids1 and ida==idb and ida!=1
```

```
R2 =   replace [ids,idv,x,y,tempo,marca],
               [ids1,idv1,x1,y1,tempo1,marca1],
               [ids2,ida,idb]
       by      [ids,idv,x,y,tempo,marca],
               [ids1,idv1,x1,y1,tempo1,marca1]
       if      ids==ids1 and ids1==ids2 and ida==idv and idb==idv1
               and tempo1<=tempo

...

R16 =  replace [ids,idv,x,y,tempo,marca], [ids1,aux]
       by      [ids,idv,x,y,tempo,marca]
       if      ids==ids1
```

Esta aplicação teste é um programa Gamma com um grau de complexidade bem mais elevado que os anteriores. Este código faz parte de um conjunto maior de programas escritos em Gamma, os quais integram um trabalho que dedicou-se a modelar um problema real de acompanhamento de alvos através do uso de fusão de dados em Gamma, desenvolvido por Rui Rodrigues no ano de 2015 [11]. Os elementos do multiconjunto são representações de nós e arestas de grafos, que são manipulados nas reações de modo a se chegar no resultado pretendido. O referido trabalho abordou um problema encontrado no âmbito científico militar-naval (Marinha do Brasil), no qual um sistema de cálculos táticos monitora uma série de parâmetros dos contatos (ou alvos) presentes no cenário de operações navais. A ideia principal é ser capaz de utilizar dados provenientes de várias fontes de captação de sinais (sensores), e aplicar técnicas de fusão sobre estes dados, de modo a aprimorar a detecção do possível trajeto percorrido pelos contatos.

Em outras palavras, o trabalho utiliza grafos para acompanhar a trajetória de alvos através da aquisição de dados de diversos sensores (ambiente *Multi Sensor*). Para tanto, o algoritmo desenvolvido baseia-se numa solução em dois estágios onde no primeiro são processados os dados referentes a cada sensor e em um segundo estágio os dados selecionados de todos os sensores são unificados visando a busca de um caminho ótimo que corresponde à melhor hipótese de caminho para determinado alvo. Dessa forma, no primeiro estágio, é gerado um grafo para cada conjunto de dados recebidos de cada sensor. Os vértices destes grafos correspondem a informações sobre o posicionamento de um alvo ou algum tipo de ruído advindo destes sensores. Para a geração das arestas destes grafos são considerados fatores de consistência

cinemática do alvo monitorado. Uma vez que os grafos estejam prontos para cada sensor, um algoritmo de otimização é aplicado visando a escolha dos pontos que pertencem a melhor hipótese de caminho para o conjunto de dados em questão. Desta forma, o resultado do primeiro estágio é um conjunto de vértices (*plots*) selecionados de cada sensor. Com este conjunto de *plots* escolhidos, inicia-se o segundo estágio que, de forma análoga ao primeiro, gera um grafo e aplica um algoritmo de otimização visando a escolha do melhor caminho, com a diferença de que neste estágio os vértices iniciais são aqueles selecionados no primeiro estágio e, portanto, advindos de mais de um sensor. Finalmente, o resultado deste último estágio representa o caminho composto pelos *plots* que pertencem à melhor hipótese de caminho para o alvo monitorado. Maiores informações para um melhor entendimento a respeito desta aplicação, com descrições mais detalhadas sobre as reações, podem ser vistas em [11].

5 - Aplicação de Cálculo Matricial

/* calcMatricial.gm */

```
calcMatricial { N tuplas de tamanho 10 }
where
calcMatricial = replace

                [m_0_0, m_1_0, m_2_0, m_3_0, m_4_0,
                 m_5_0, m_6_0, m_7_0, m_8_0, m_9_0],

                [m_0_1, m_1_1, m_2_1, m_3_1, m_4_1,
                 m_5_1, m_6_1, m_7_1, m_8_1, m_9_1]

                by 1
                if (determinante(m_0_0,...,m_9_0,
                                 m_0_1,...,m_9_1,
                                 ...,
                                 m_0_9,...,m_9_9) != 0)
```

Nesta última aplicação, resolvemos criar um programa Gamma hipotético que realiza cálculo com matrizes. A ideia foi criar um programa cuja condição de reação fosse computacionalmente mais custosa de se calcular, uma vez que é neste trecho de código que o paralelismo da implementação com GPUs é mais

explorado, podendo consequentemente, obter maiores benefícios de aceleração. O multiconjunto inicial é formado por `"N"` tuplas de dez valores cada, as quais assumirão o papel de colunas de uma matriz quadrada de ordem dez durante o processamento. O programa contém uma única reação chamada `"calcMatricial"`, que funciona selecionando no multiconjunto pares de tuplas, e testando a condição de reação entre elas, que por sua vez, consiste em proceder o cálculo do determinante da matriz quadrada de ordem dez (pelo método de triangularização), onde as duas primeiras colunas são as duas tuplas selecionadas, e as demais colunas são preenchidas com valores fixos durante a computação. Caso a condição de reação for verdadeira, ou seja, o determinante calculado for diferente de zero, então a ação é aplicada, a qual apenas consome as duas tuplas selecionadas e reescreve no multiconjunto o valor `"1"`.

Como a ideia neste exemplo foi apenas a de podermos analisar o comportamento das implementações quando o programa possui uma condição de reação mais complexa, criamos um programa em Gamma, que da maneira com está expresso, não é suportado pela atual sintaxe do modelo, uma vez que a obtenção do determinante envolve cálculos complicados. Então, o que fizemos foi substituir no código em Gamma a condição de reação por um simples `"if true"`, e depois modificamos diretamente o código em C gerado pelo compilador, a fim de inserir os trechos de código para o cálculo do determinante. Vale ressaltar que o código modificado manualmente foi apenas o trecho da condição de reação, de modo que todo o restante é o gerado automaticamente pelo compilador Gamma.

6.1.2 Configuração e Execução

Foram escolhidos para a execução de cada aplicação teste, três tamanhos distintos de entrada, como podemos ver na Tabela 6.1.

Tabela 6.1: Configurações das entradas para as aplicações teste.

Aplicação	**Unidade de Entrada**	**Entrada 1**	**Entrada 2**	**Entrada 3**
Números Primos	de 2 até N elementos	N=5000	N=7500	N=10000
Ordenação	N elementos	N=300	N=450	N=600
Fibonacci	N-ésimo termo	N=17	N=18	N=20
Fusão de Dados	N sensores	N=3	N=6	N=9
Cálculo Matricial	N tuplas	N=1000	N=5000	N=10000

Todas as aplicações tiveram então seus três casos de entrada executados sobre as implementações *Gamma-Base* e *Gamma-GPU*, e, a fim de que pudéssemos obter

uma medida mais confiável para o tempo total de execução, usamos a média dos tempos de dez execuções para cada caso considerado. A configuração de hardware no ambiente distribuído, ou seja, o número de nós e de *cores* de CPU utilizados pelas aplicações em cada uma das duas implementações foi exatamente o mesmo, variando logicamente, apenas o uso ou não das GPUs, pois a implementação *Gamma-Base* não as suporta, enquanto que a *Gamma-GPU* tem como principal propósito justamente sua utilização.

Os resultados obtidos para cada aplicação serão exibidos na Seção 6.2, através de gráficos e tabelas que sumarizam as informações de cada teste, contendo informações como a média dos tempos de execução, o desvio padrão, e também o *speedup* possivelmente atingido. Esta última medida, o *speedup*, nos informa a aceleração obtida na implementação *Gamma-GPU* em relação à implementação *Gamma-Base*, e será calculada da seguinte forma:

$$speedup = \frac{T_{base}}{T_{gpu}}$$

onde:

$\mathbf{T_{base}}$: tempo de execução sobre a implementação *Gamma-Base*
$\mathbf{T_{gpu}}$: tempo de execução sobre a implementação *Gamma-GPU*

Ou seja, o *speedup* nos diz quantas vezes mais rápida foi a execução de *Gamma-GPU* se comparada com *Gamma-Base*.

6.1.3 Ambiente de Testes

Como já sabemos, as duas implementações, *Gamma-Base* e *Gamma-GPU*, possuem a necessidade de uma plataforma de hardware composta de múltiplos processadores (ou ao menos uma CPU *multicore*) para que possam executar, e a última precisa obviamente de GPUs disponíveis como coprocessadores de aceleração. Um simples computador dotado de uma CPU *multicore* e de uma GPU, já seria capaz de nos dar uma boa perspectiva a respeito do comportamento dos modelos, todavia, um ambiente mais poderoso e completo cumpre melhor esta função, fazendo com que os resultados obtidos sejam mais representativos. Diante disso, as aplicações de teste neste trabalho foram executadas em um ambiente distribuído *multi-GPU*, ou seja, um *cluster* de GPUs, detalhado adiante.

Instituição de Apoio

A instituição que forneceu apoio para a realização dos experimentos neste trabalho foi o CENAPAD-UFC[1], que é um dos centros integrantes do Sistema Nacional de Processamento de Alto Desempenho (SINAPAD), o qual reúne nove centros de todo o Brasil para a formação de uma grade computacional com o objetivo de fornecer os recursos de Processamento de Alto Desempenho (PAD) necessários para o desenvolvimento científico e tecnológico do país. O referido centro possui instalado um supercomputador na arquitetura de *cluster*, composto por vários servidores em *blade*, podendo atingir um processamento teórico na casa de 14,2 *TeraFlops* [65].

Recursos de Hardware

O *cluster* conta com a seguinte configuração de hardware:

- 48 *Blades Bull* 500, cada uma possuindo:
  - 2x processadores *Intel*® *Westmere*® X5650 EP;
  - 6x4GB DDR3 1333MHz de memória RAM;
  - 1x 250GB SATA II 1,8"HDD;

    Total: 576 núcleos de processamento e 1152GB de RAM.
- 3 *Blades Bullx* B505, cada uma possuindo:
  - 1x processador *Intel*® *Xeon*® CPU E5-2470;
  - 96GB DDR3 de Memória RAM;
  - **1x GPU NVidia Tesla K20m com 2496 CUDA-Cores e 5GB DRR5 de memória;**

    Total: 48 núcleos de processamento, 288GB de RAM e 15GB de GPU.

Além disso, para atuar em conjunto do *cluster* para processamento de alto desempenho, o CENAPAD-UFC está equipado com *storage NetApp FAS* com capacidade de armazenamento bruta de 100 *TeraBytes*.

Recursos de Software

O *cluster* conta com os seguintes recursos de software:

[1] "Pesquisa desenvolvida junto ao Centro Nacional de Processamento de Alto Desempenho, instalado na Universidade Federal do Ceará".

- Sistema Operacional Linux *Red Hat* EL 6.4;
- Ambientes de compilação e execução MPI;
- Compilador CUDA nvcc;
- Gerenciador de recursos SLURM (*Simple Linux Utility for Resource Management*).

6.2 Resultados Obtidos

A seguir serão exibidos os resultados obtidos para as cinco aplicações de teste. Será mostrado para cada uma delas, uma tabela de informações sobre a execução contendo: a média dos tempos de execução em segundos (`"Tempo(s)"`) para cada implementação (`"Base"` e `"GPU"`); o desvio padrão da média dos tempos (`"Desvio Padrão"`); e o *speedup* obtido na versão com GPUs *Gamma-GPU* em relação à versão sem GPUs *Gamma-Base* (`"Speedup"`). Serão exibidos também, um gráfico contendo as curvas de tempo de execução para cada implementação, e um gráfico com os *speedups* obtidos. A análise e a discussão sobre os resultados serão realizadas na Seção 6.3.

6.2.1 Números Primos

Tabela 6.2: Resultados obtidos na aplicação `"Números Primos"`.

	Entrada (2 até N elementos)					
	N=1000		**N=5000**		**N=10000**	
	Base	*GPU*	*Base*	*GPU*	*Base*	*GPU*
Tempo (s)	61,91	3,63	192,32	4,00	421,83	4,12
Desvio Padrão	2,11	0,45	1,66	0,46	2,06	0,53
Speedup	17,05		48,13		102,44	

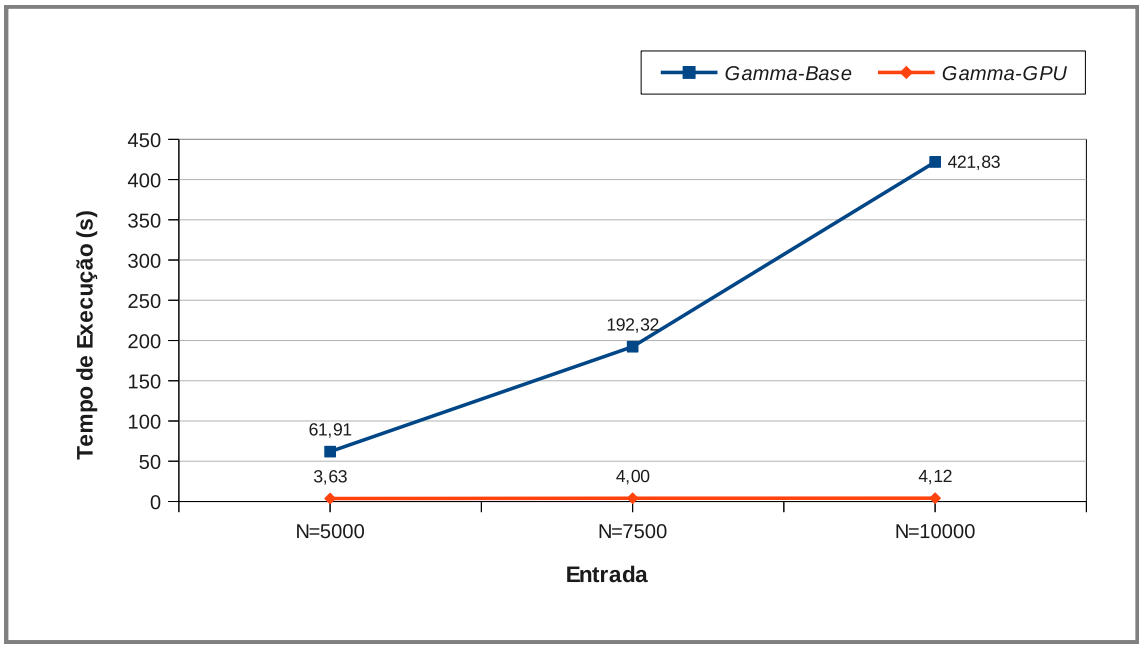


Figura 6.1: Gráfico dos tempos de execução para as três entradas da aplicação `"Números Primos"` nas implementações *Gamma-Base* e *Gamma-GPU*.

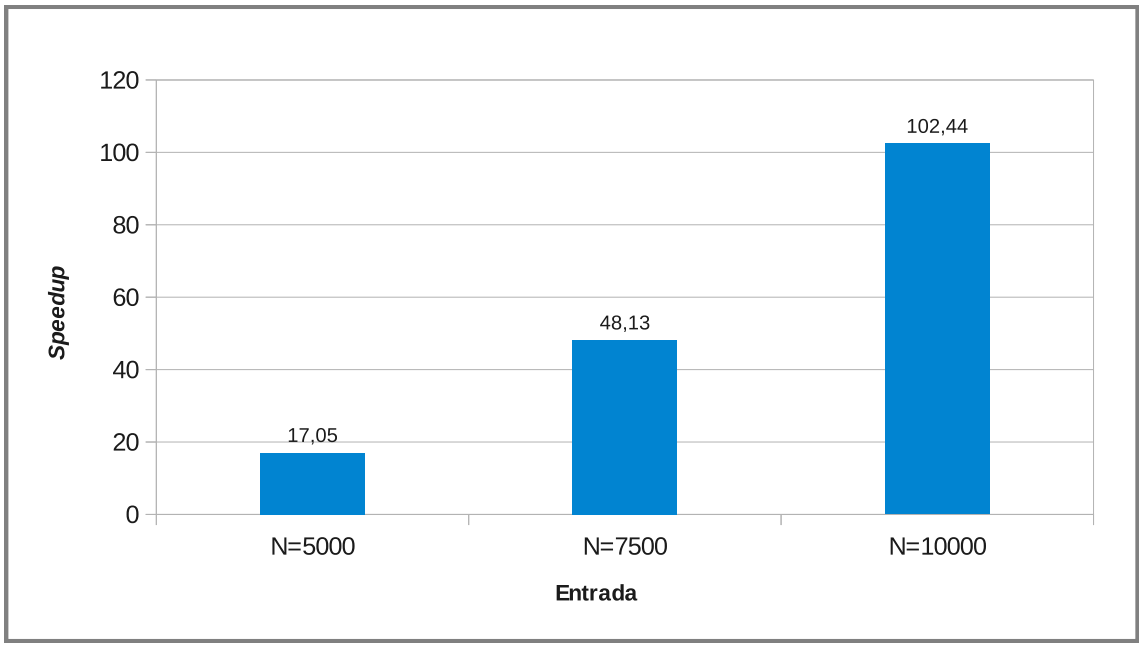


Figura 6.2: Gráfico dos *speedups* obtidos com a implementação *Gamma-GPU* em relação à *Gamma-Base* para as três entradas da aplicação `"Números Primos"`.

6.2.2 Algoritmo de Ordenação

Tabela 6.3: Resultados obtidos na aplicação `"Ordenação"`.

	Entrada (N elementos)					
	N=300		**N=450**		**N=600**	
	Base	*GPU*	*Base*	*GPU*	*Base*	*GPU*
Tempo (s)	59,38	7,23	198,73	8,50	465,00	11,29
Desvio Padrão	0,56	0,46	4,50	0,13	4,36	0,16
Speedup	8,22		23,37		41,17	

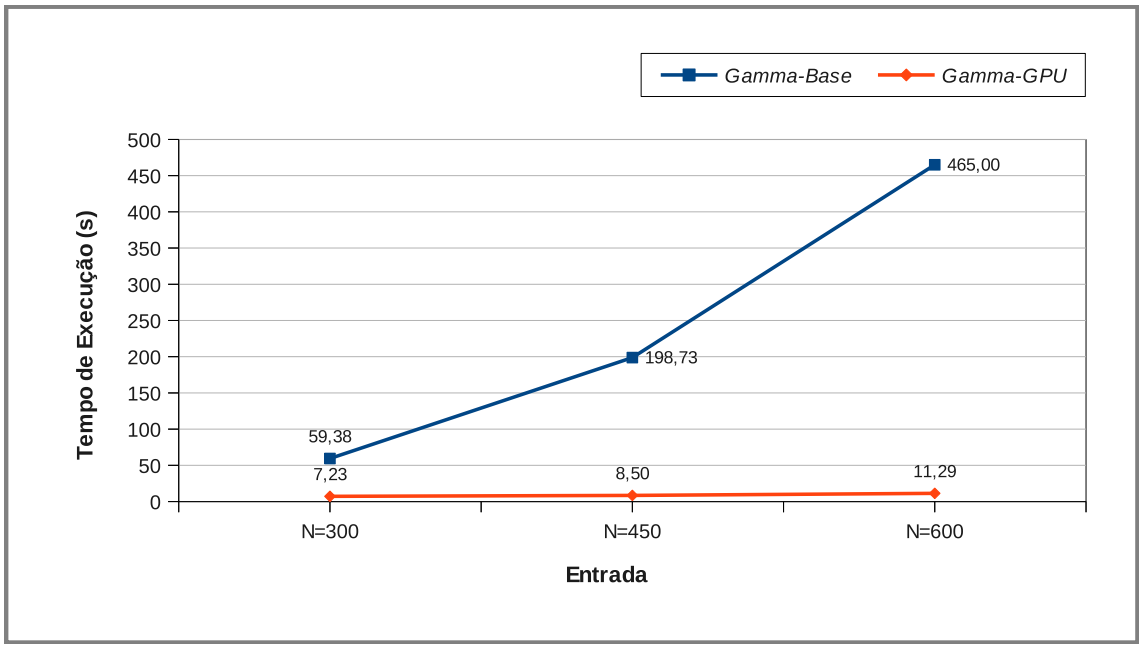


Figura 6.3: Gráfico dos tempos de execução para as três entradas da aplicação `"Ordenação"` nas implementações *Gamma-Base* e *Gamma-GPU*.

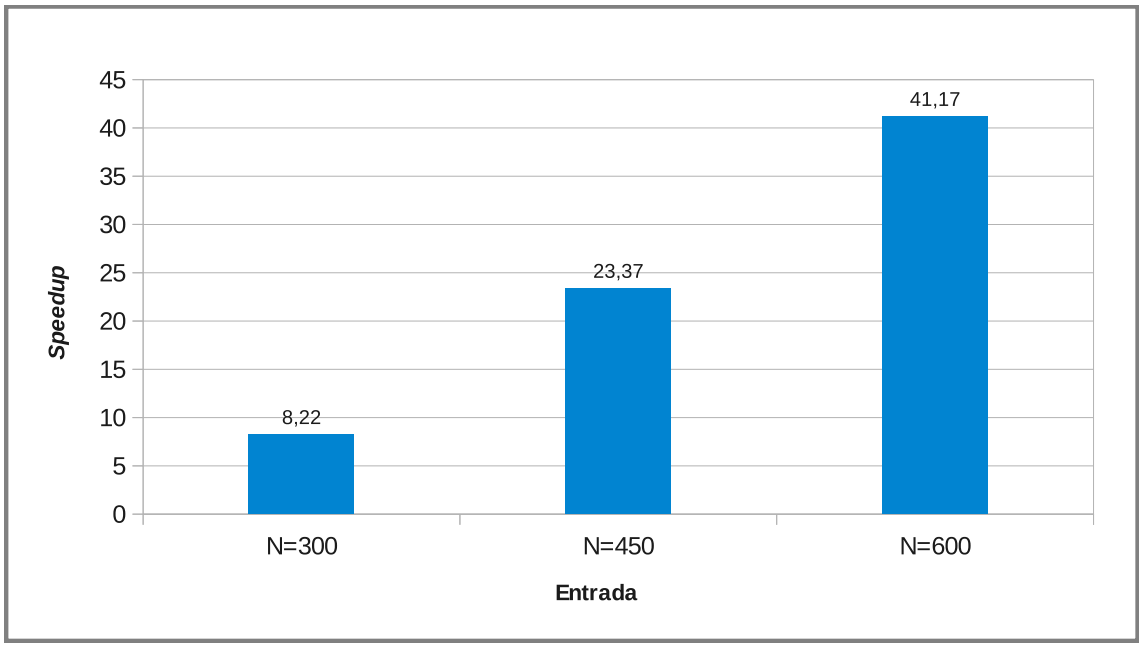


Figura 6.4: Gráfico dos *speedups* obtidos com a implementação *Gamma-GPU* em relação à *Gamma-Base* para as três entradas da aplicação `"Ordenação"`.

6.2.3 *Fibonacci*

Tabela 6.4: Resultados obtidos na aplicação `"Fibonacci"`.

	Entrada (N-ésimo termo)					
	N=17		**N=18**		**N=20**	
	Base	*GPU*	*Base*	*GPU*	*Base*	*GPU*
Tempo (s)	13,96	5,46	32,64	5,91	224,75	6,26
Desvio Padrão	1,14	0,09	2,24	0,52	7,34	0,40
Speedup	2,56		5,52		35,88	

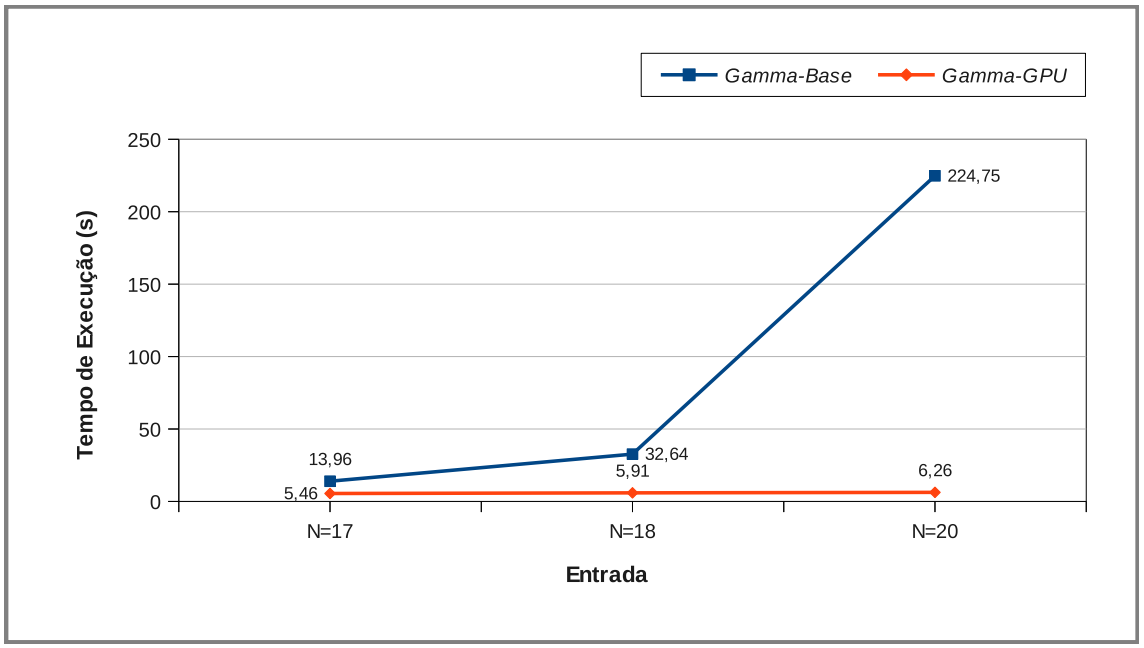


Figura 6.5: Gráfico dos tempos de execução para as três entradas da aplicação `"Fibonacci"` nas implementações *Gamma-Base* e *Gamma-GPU*.

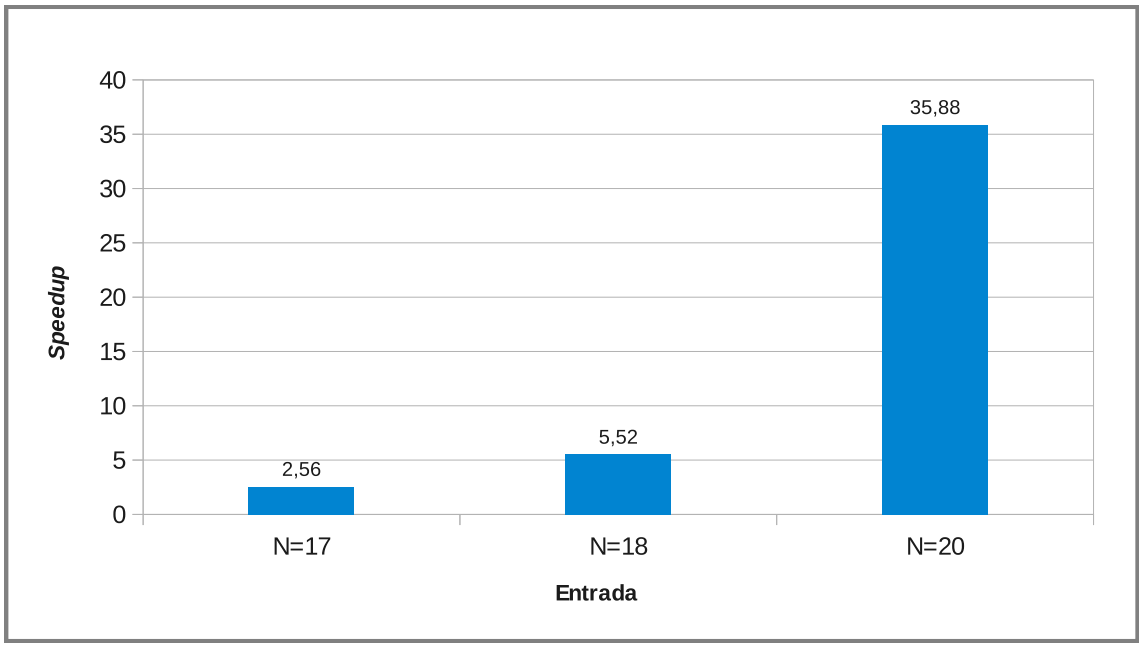


Figura 6.6: Gráfico dos *speedups* obtidos com a implementação *Gamma-GPU* em relação à *Gamma-Base* para as três entradas da aplicação `"Fibonacci"`.

6.2.4 Fusão de Dados para Acompanhamento de Alvos

Tabela 6.5: Resultados obtidos na aplicação `"Fusão de Dados"`.

	Entrada (N sensores)					
	N=3		**N=6**		**N=9**	
	Base	*GPU*	*Base*	*GPU*	*Base*	*GPU*
Tempo (s)	69,15	97,62	543,74	300,61	2001,80	810,38
Desvio Padrão	0,72	2,60	22,43	26,58	22,77	27,71
Speedup	0,71		1,81		2,47	

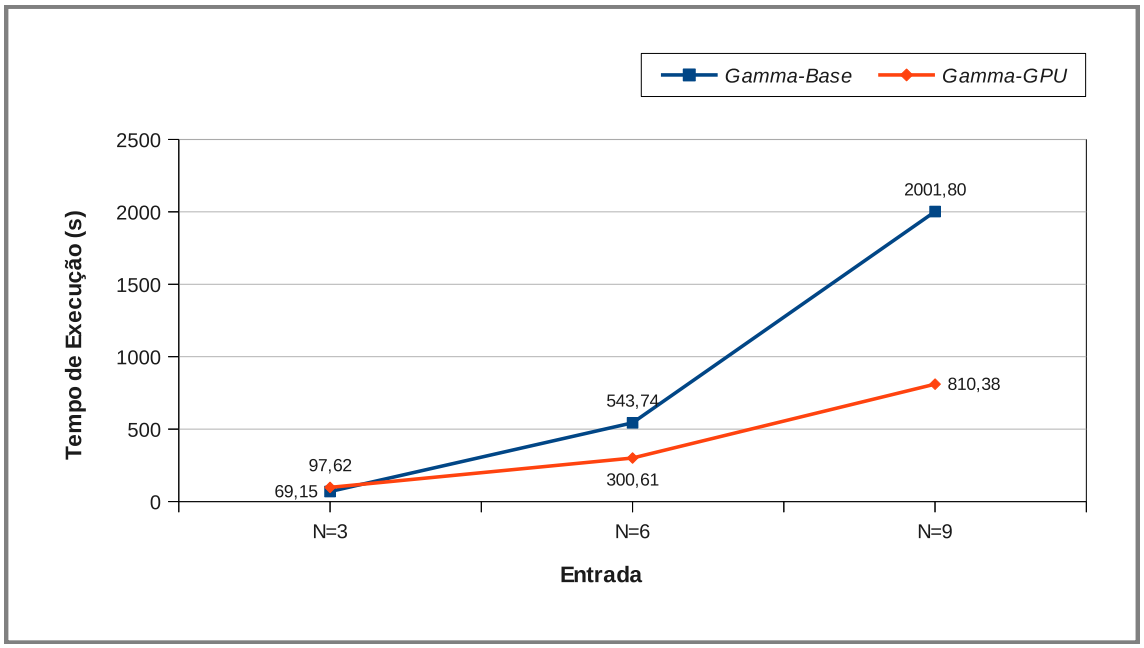


Figura 6.7: Gráfico dos tempos de execução para as três entradas da aplicação `"Fusão de Dados"` nas implementações *Gamma-Base* e *Gamma-GPU*.

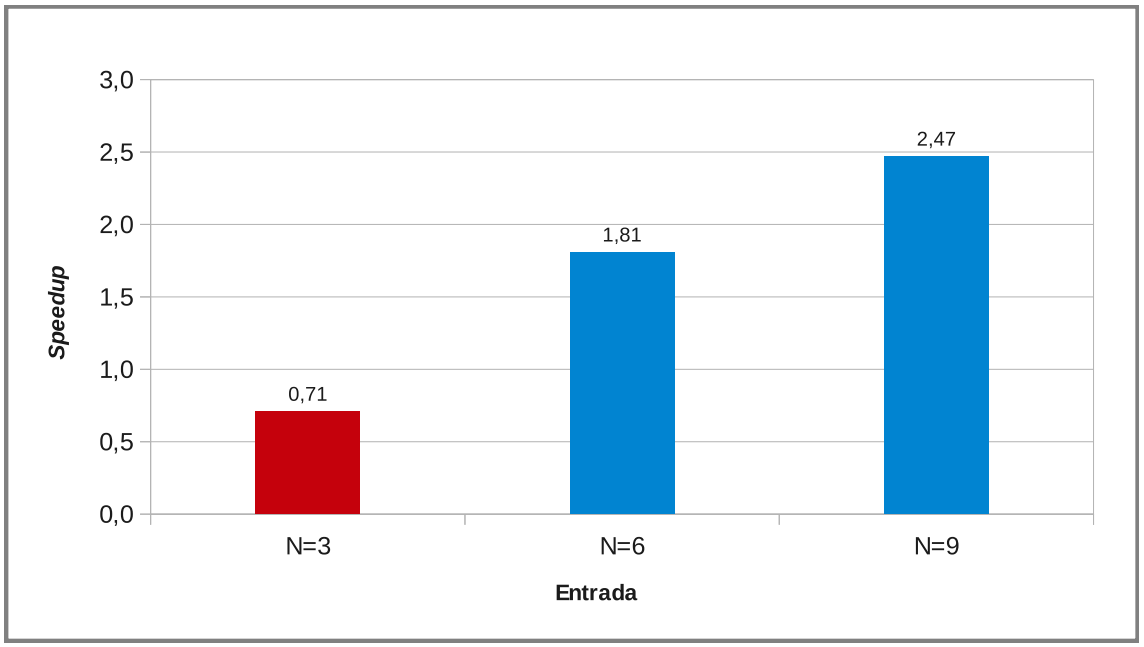


Figura 6.8: Gráfico dos *speedups* obtidos com a implementação *Gamma-GPU* em relação à *Gamma-Base* para as três entradas da aplicação `"Fusão de Dados"`.

6.2.5 Aplicação de Cálculo Matricial

Tabela 6.6: Resultados obtidos na aplicação `"Cálculo Matricial"`.

	Entrada (N tuplas)					
	N=1000		**N=5000**		**N=10000**	
	Base	*GPU*	*Base*	*GPU*	*Base*	*GPU*
Tempo (s)	4,06	3,33	96,18	3,86	388,98	4,47
Desvio Padrão	0,03	0,19	1,71	0,57	4,43	0,10
Speedup	1,22		24,94		87,10	

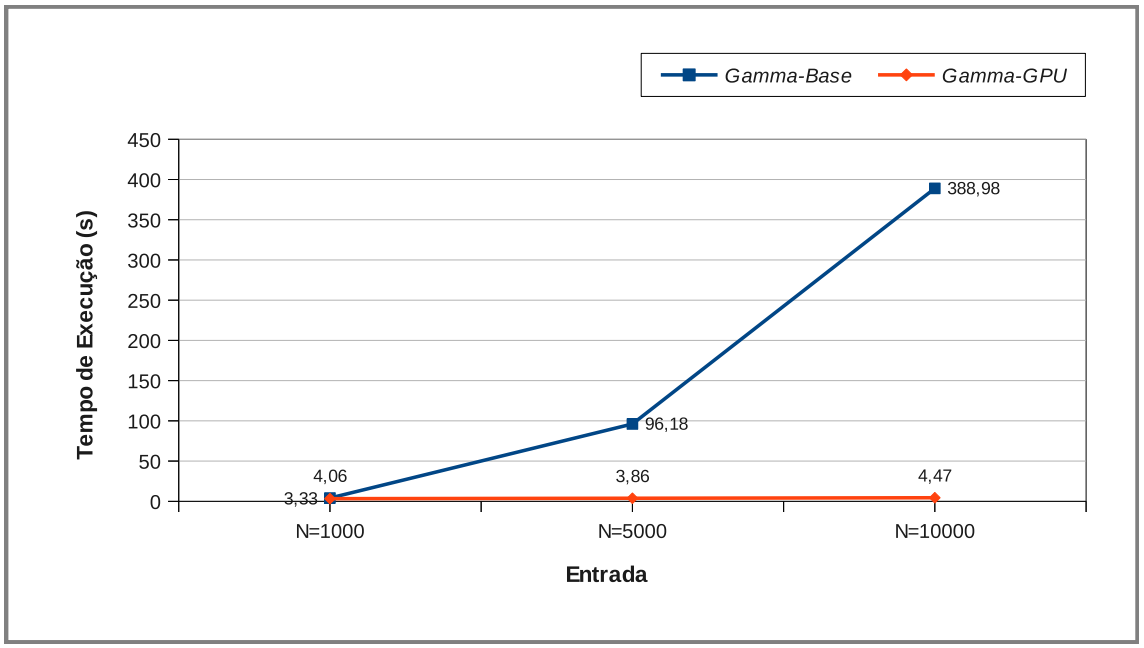


Figura 6.9: Gráfico dos tempos de execução para as três entradas da aplicação `"Cálculo Matricial"` nas implementações *Gamma-Base* e *Gamma-GPU*.

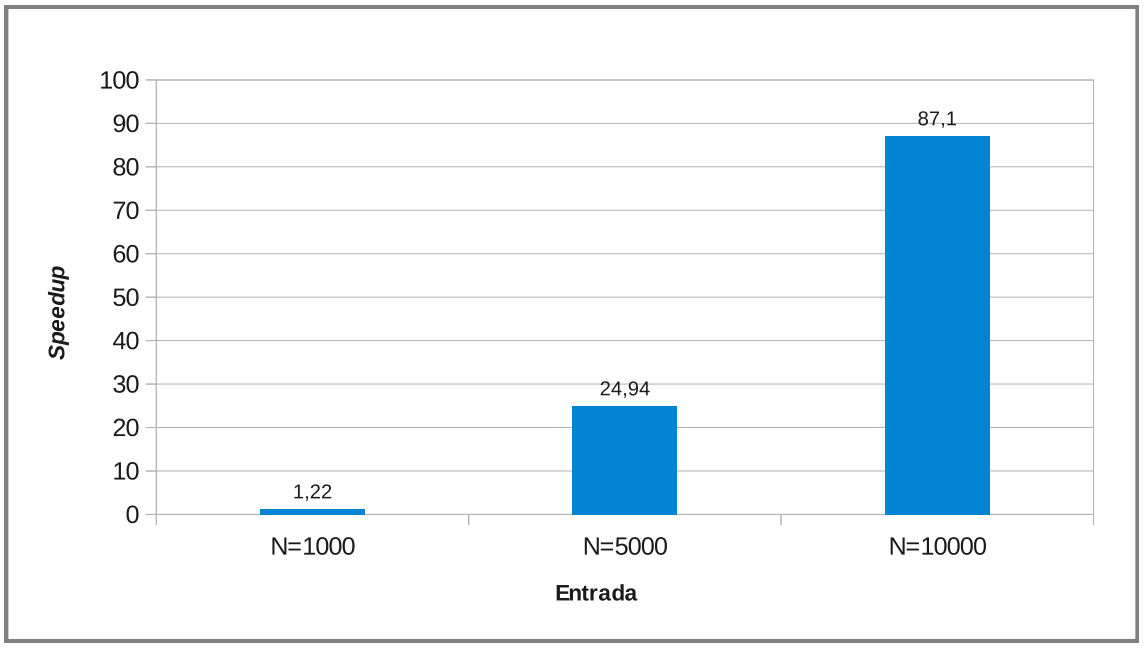


Figura 6.10: Gráfico dos *speedups* obtidos com a implementação *Gamma-GPU* em relação à *Gamma-Base* para as três entradas da aplicação `"Cálculo Matricial"`.

6.2.5.1 Resultados com uma Implementação Sequencial

Neste ponto dos testes, resolvemos utilizar a aplicação de cálculo matricial para verificar o desempenho da implementação *Gamma-GPU* frente a uma implementação sequencial de Gamma. Tal implementação, que chamaremos *Gamma-Seq*, consiste em utilizar o modelo de computação de Gamma em apenas um único processador,

de modo que todo o processamento ocorre de forma sequencial, e naturalmente não existe ambiente distribuído nem troca de mensagens durante a execução. Utilizamos entradas de 30, 60, e 100 mil tuplas nesta ocasião, valores bem maiores do que os anteriormente usados na comparação entre *Gamma-GPU* e *Gamma-Base*. Veremos o porquê disto na seção de análise dos resultados. A seguir podemos ver os resultados obtidos nesta configuração de teste.

Tabela 6.7: Resultados obtidos na aplicação `"Cálculo Matricial"` - experimento com versão sequencial.

	Entrada (N tuplas)					
	N=30000		**N=60000**		**N=100000**	
	Sequencial	*GPU*	*Sequencial*	*GPU*	*Sequencial*	*GPU*
Tempo (s)	8,99	4,84	41,77	5,26	158,00	5,83
Desvio Padrão	0,21	0,13	0,50	0,10	1,38	0,09
Speedup	1,86		7,94		27,08	

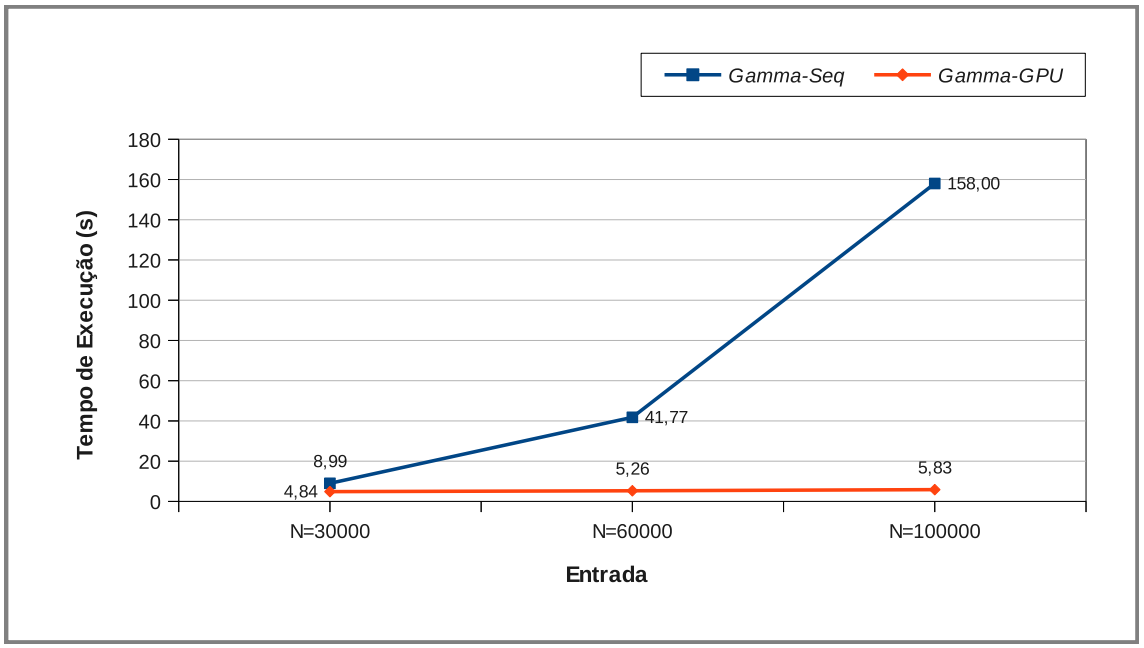


Figura 6.11: Gráfico dos tempos de execução para entradas de 30, 60, e 100 mil tuplas da aplicação `"Cálculo Matricial"` nas implementações *Gamma-Seq* e *Gamma-GPU*.

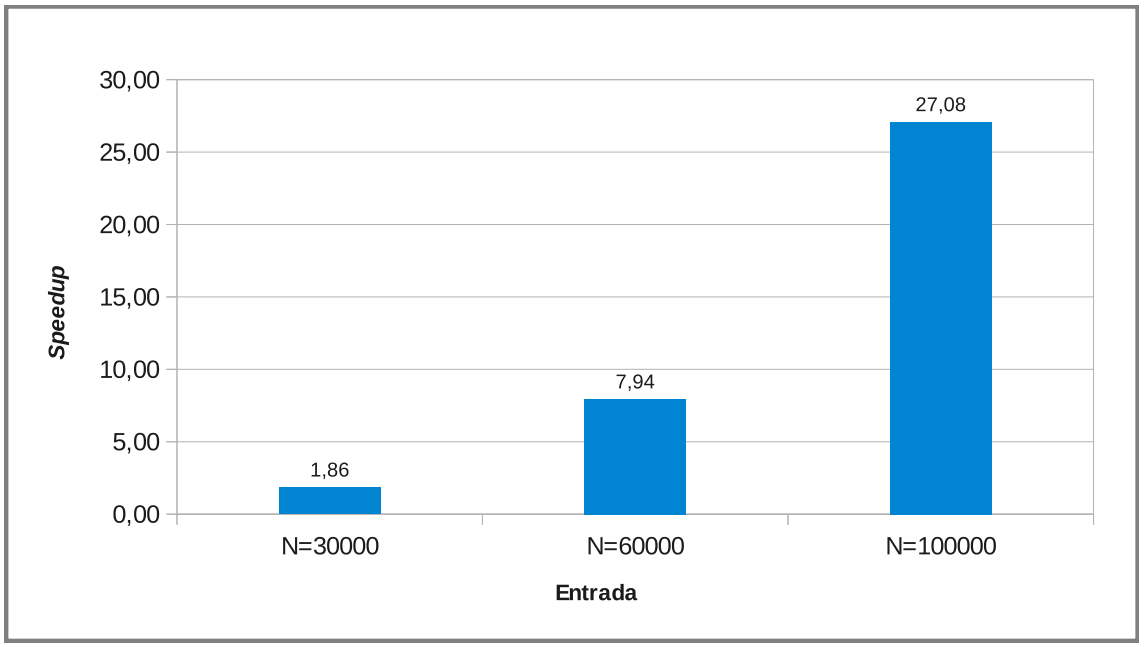


Figura 6.12: Gráfico dos *speedups* obtidos com a implementação *Gamma-GPU* em relação à *Gamma-Seq* para entradas de 30, 60, e 100 mil tuplas da aplicação `"Cálculo Matricial"`.

6.3 Análise dos Resultados

A primeira constatação que fizemos com a execução dos experimentos foi qualitativa, e diz respeito à corretude da nova implementação *Gamma-GPU*. Todas as aplicações de teste tiveram suas respostas comparadas entre as duas implementações, e pudemos verificar que os resultados foram os mesmos, o que nos confirmou que nosso novo modelo computacional foi implementado corretamente, sendo capaz de adicionar as GPUs ao cenário de processamento de forma coerente e confiável. A segunda parte da análise será sobre aspectos quantitativos da implementação, na qual faremos a comparação entre o desempenho da *Gamma-GPU* contra a *Gamma-Base* através de seus tempos de execução. Levando em consideração as características intrínsecas de cada aplicação, analisaremos o porquê dos ganhos de desempenho obtidos, bem como as limitações destes, quando usamos o modelo com GPUs.

Podemos notar que na primeira aplicação, `"Números Primos"`, a implementação *Gamma-GPU* obteve ótimos *speedups* sobre a *Gamma-Base* para os três casos de entrada, chegando a uma aceleração superior a cem vezes mais rápida na execução da maior entrada, com dez mil elementos. Vemos que o tempo de execução da implementação com GPUs permanece praticamente constante a medida que a entrada aumenta de tamanho, ao passo que a implementação base sofre duramente os impactos desta variação no número de elementos do multiconjunto. Na *Gamma-GPU*, o que ocorre é que a cada iteração do programa, a condição de reação se verifica

como verdadeira para várias das combinações entre os elementos do multiconjunto, fazendo com que o mesmo diminua vertiginosamente de tamanho a cada passo, e levando a um ganho bastante significativo de desempenho atrelado ao processamento paralelo na GPU. Mesmo com o aumento no tamanho da entrada, os milhares de *CUDA-Cores* das GPUs continuam conseguindo processar de maneira muito eficiente o multiconjunto, a ponto do tempo de execução sofrer impactos mínimos. Já na implementação *Gamma-Base*, a redução do multiconjunto acontece lentamente, a medida que apenas uma condição de reação é testada a cada iteração, e devido a isso, o número de mensagens trocadas entres as células de processamento no ambiente distribuído torna-se demasiadamente alto, visto que a cada iteração o multiconjunto é devolvido à *Bag-Cell*, para só depois ser novamente recebido e dar prosseguimento à execução. Isto acarreta um alto *overhead* de comunicação no sistema, devido ao grande número de mensagens trafegadas na rede, o que acaba por representar uma fatia dominante no tempo de execução final da aplicação.

Na segunda aplicação, `"Algoritmo de Ordenação"`, podemos perceber nos resultados um comportamento similar ao apresentado na primeira aplicação `"Números Primos"`, pois os *speedups* também aumentam a medida que o tamanho da entrada aumenta, porém são um pouco menores que no caso anterior. Isto está ligado às características próprias do programa de ordenação, pois sua segunda reação, chamada `"sort"`, possui uma condição de reação um pouco mais restritiva que a do caso dos números primos, levando a um número menor de combinações que reagem a cada iteração, e consequentemente, não resultando em uma exploração adequada de todo o paralelismo oferecido pela GPU. Vale ressaltar que a primeira reação do programa, chamada `"init"`, pode se beneficiar amplamente das verificações em paralelo realizadas nas *threads* da GPU, pois a condição de reação é sempre verdadeira (`"if true"`), o que faz com que a transformação inicial do multiconjunto possa ser realizada em uma única iteração de processamento.

O programa que calcula o n-ésimo termo de *Fibonacci*, nossa terceira aplicação teste, possui complexidade exponencial em relação ao termo que se deseja encontrar. Esta característica fica bastante evidente nos resultados dos tempos de execução da implementação *Gamma-Base*, na qual a terceira entrada, que é o cálculo do vigésimo termo, gasta um tempo bem mais alto que as anteriores. Com o uso das GPUs, fomos capazes de mitigar este custo exponencial e obter tempos bem mais baixos de execução para os casos testados, mantendo-se praticamente constante para as três entradas. Novamente, isto ocorre porque as condições de reação das três reações presentes no programa (`"gera1"`, `"gera2"` e `"soma"`) são avaliadas como verdadeiras para praticamente a totalidade do multiconjunto em cada iteração, fazendo com que tanto a expansão quanto a posterior redução do mesmo, ocorram em uma taxa bastante acelerada, e a reposta do programa seja obtida mais rapidamente.

A quarta aplicação teste foi a de fusão de dados para acompanhamento de alvos, e neste experimento surgiram resultados em alguns aspectos distintos dos analisados até agora. O que podemos perceber é que a medida que o tamanho da entrada (neste caso representada pelo número de sensores considerados como fontes de sinais no contexto da aplicação) aumenta, temos um aumento na eficiência da implementação *Gamma-GPU* relativamente à implementação *Gamma-Base*, sendo este um comportamento comum às outras aplicações já discutidas previamente. Todavia, analisando a magnitude dos *speedups* obtidos, vemos que estes ficaram na casa das unidades, ao passo que até então, estávamos observando para todas as aplicações a obtenção de acelerações na casa de dezenas de vezes mais rápidas. Mais que isso, chamamos a atenção para o caso observado na primeira entrada testada, com três sensores, onde obteve-se um *"speedup"* de 0,71, ou seja, na realidade o que tivemos neste caso foi o efeito contrário, chamado de *slowdown*, pois a implementação *Gamma-GPU* foi mais lenta que a *Gamma-Base*, e ao invés de acelerá-la, diminuiu sua eficiência.

O que ocorre neste cenário é que o *overhead* adicionado pela invocação dos *kernels* e principalmente pelas cópias de dados entre as memórias da CPU e GPU, presentes na implementação *Gamma-GPU*, é maior do que o ganho de computação útil disponibilizado pela exploração do paralelismo na unidade de processamento gráfico. A explicação para este fenômeno está relacionada diretamente com as características intrínsecas e com a dinâmica do programa Gamma para o problema em particular, no qual o número de reações que podem potencialmente executar em paralelo é muito reduzido, de forma que a GPU não consegue contribuir para a melhoria de desempenho, causando apenas uma sobrecarga adicional de comunicação entre os processadores heterogêneos, sem incorrer nos ganhos computacionais associados. Imaginamos que outros programas Gamma com reações de características similares às do problema de fusão de dados podem sofrer impactos negativos semelhantes, e por isso, uma solução para mitigar a possível ineficiência das GPUs nestes casos, será discutida nas propostas de trabalhos futuros (Seção 7.3).

Finalmente, testamos a aplicação de cálculo matricial, e como esperado, os resultados nos demonstram ganhos de desempenho obtidos pela implementação *Gamma-GPU* sobre a *Gamma-Base*, que aumentam a medida que o tamanho da entrada aumenta. Neste exemplo, além de muitas reações terem potencial de executar em paralelo, a verificação da condição de reação é realizada através de um cálculo de determinante de uma matriz quadrada de ordem dez, o que é bem mais oneroso computacionalmente que as condições simples que vimos nas aplicações anteriores. Isso realça ainda mais o ganho que pode ser obtido com a paralelização das comparações através das *threads* nas GPUs, uma vez que o trecho de código paralelizável representa uma fatia maior na composição do tempo total de execução.

Esta aplicação de cálculo matricial é hipotética e não tem nenhum significado

por si só, contudo, podemos dizer que qualquer aplicação que necessite do cálculo de determinantes, como por exemplo, resolução de sistemas lineares, deve ter um comportamento similar ao deste programa, e mais genericamente, podemos inferir a partir dos resultados obtidos, que outras aplicações que possuam uma condição de reação custosa de se computar, também tendem a se beneficiar largamente com a introdução das GPUs no ambiente computacional.

6.3.1 Análise com uma Implementação Sequencial

Como mostrado na seção de resultados obtidos, realizamos para o caso da aplicação de cálculo matricial um experimento extra, a fim de verificar o desempenho da implementação *Gamma-GPU* contra uma implementação sequencial de Gamma, chamada *Gamma-Seq*. Isto porque já vimos que *Gamma-GPU* foi bem superior à *Gamma-Base* em praticamente todos os casos de teste, contudo, as duas implementações compartilham o mesmo arcabouço arquitetural de troca de mensagens, e este arcabouço sofre com problemas de desempenho devido ao fato de não haver implementado um escalonador apropriado como o proposto na Seção 4.5. Com isso, a implementação *Gamma-Base* acaba sempre sendo superada pela versão sequencial de Gamma, que não possui nenhum *overhead* de troca de mensagens. Em algumas aplicações de teste, principalmente aquelas que possuem um potencial menor de paralelismo, até mesmo a *Gamma-GPU* pode vir a ser superada pela implementação sequencial. Porém, quando aumentamos o tamanho da entrada para valores muito grandes, a implementação *Gamma-GPU*, mesmo sofrendo dos impactos da falta do escalonador no ambiente distribuído, é capaz de compensar esta limitação através do ganho obtido pelo processamento nas GPUs, e se sobressai em relação à *Gamma-Seq*.

Por isso resolvemos utilizar a aplicação de cálculo matricial com entradas maiores que as anteriormente empregadas, e procedemos os experimentos nas versões *Gamma-Seq* e *Gamma-GPU*. A versão *Gamma-Base* não foi testada nesta ocasião, pois como já falamos, é sempre superada até mesmo pela versão sequencial, e com os novos tamanhos de entrada, chegando a cem mil elementos, levaria muito tempo para executar e não contribuiria para a análise. Como podemos notar nos resultados obtidos, a implementação *Gamma-Seq* ficou próxima à *Gamma-GPU* em relação ao tempo de execução para a primeira entrada de 30 mil tuplas, entretanto, quando aumentamos a entrada para 60 e 100 mil tuplas, percebemos claramente os impactos sofridos pela versão sequencial, ao passo que a versão com GPUs mantém-se praticamente estável em seu tempo de execução, levando a *speedups* bastante significativos, chegando a 27 vezes mais rápido para a entrada de 100 mil tuplas.

Outras aplicações também tendem a demonstrar um comportamento similar quando a entrada utilizada é muito grande, mas cada uma com magnitudes de ganho

diferentes, de acordo com as características de paralelismo exibidas pelas mesmas. Como exemplo, no trabalho contemporâneo ao nosso que desenvolveu a aplicação de fusão de dados em Gamma [11] que utilizamos como um dos casos de teste neste trabalho, foram procedidos experimentos comparando as implementações *Gamma-Seq*, *Gamma-Base*, e *Gamma-GPU*. Como já discutimos, a aplicação de fusão de dados exibe um paralelismo limitado no tocante ao número de reações que podem ocorrer simultaneamente, e ainda assim, para a maior entrada testada no referido trabalho (base de dados com onze sensores), foi constatado que a implementação *Gamma-GPU* superou a versão sequencial *Gamma-Seq*, obtendo um *speedup* próximo a duas vezes mais rápido.

Capítulo 7

Conclusões

Neste último capítulo faremos a conclusão do trabalho, no qual falaremos sobre o cumprimento dos objetivos elencados no início da dissertação, e discutiremos a respeito dos pontos fortes e fracos do que foi desenvolvido, evidenciando as contribuições trazidas pela solução proposta, e citando limitações e possíveis melhorias através de trabalhos futuros.

7.1 Realização dos Objetivos

O principal objetivo desta dissertação foi prover uma nova implementação do paradigma Gamma que fornecesse suporte à execução de programas sobre a arquitetura das GPUs. Isso porque vemos em Gamma uma forma simples e natural de se expressar problemas, e acreditamos que seu modelo de computação se casaria muito bem ao modo de processamento adotado pelas GPUs, de maneira que a união destes dois conceitos poderia trazer um beneficio em dobro, com os programas Gamma sendo acelerados pelas GPUs, e com o acesso às GPUs sendo feito de forma transparente usando Gamma. Utilizando como base uma implementação paralela e distribuída já existente voltada para plataformas com processadores convencionais, fomos capazes de estendê-la com sucesso, mantendo o suporte a um ambiente computacional distribuído e adicionando à arquitetura do modelo as GPUs, o que resultou em uma plataforma heterogênea de processamento. Chamamos esta nova implementação de *Gamma-GPU*.

Pudemos verificar através de experimentos práticos realizados sobre um *cluster* de GPUs que a nossa nova implementação comportou-se de maneira correta em relação às respostas emitidas na execução de vários programas em Gamma, como o de cálculo de números primos, de ordenação de valores, de fusão de dados, entre outros. Além disso, mensuramos os tempos de execução das aplicações quando executadas sobre a implementação *Gamma-GPU* e os confrontamos com os tempos da implementação base que não suporta GPUs. Percebemos claramente o grande

benefício em termos de aceleração trazidos pelo novo modelo, com *speedups* que chegaram a valores bastante expressivos superando o modelo precedente em uma centena de vezes para alguns dos casos de teste. Para uma das aplicações de teste, a de cálculo matricial, realizamos um experimento extra com entradas maiores a fim de proceder uma comparação de desempenho de *Gamma-GPU* com uma implementação sequencial de Gamma, e comprovamos também nesse caso, que *Gamma-GPU* foi capaz de prover boas acelerações para o tempo de execução do programa em questão.

Desta forma, acreditamos ter cumprido de forma satisfatória os objetivos que identificamos no início do trabalho, tendo fornecido um estudo e uma implementação mais atuais relacionados ao paradigma Gamma, que deve motivar outros pesquisadores a estudarem e contribuírem com aspectos ligados ao tema. Cumprindo os objetivos, evidenciamos vários pontos fortes da solução desenvolvida, mas naturalmente, existem também limitações que surgiram ao longo do desenvolvimento da solução que ainda estão em aberto, e nos permitem vislumbrar uma série de incrementos ao modelo implementado, conforme veremos adiante.

7.2 Discussão

Através do compilador modificado na nova implementação *Gamma-GPU*, nosso modelo foi capaz de possibilitar o uso das GPUs através da linguagem Gamma de forma totalmente transparente ao programador, o qual não necessita conhecer os detalhes de baixo nível e complexidades das linguagens normalmente empregadas para programação das mesmas. Além disso, *Gamma-GPU* pôde proporcionar um aumento na eficiência da execução de programas escritos em Gamma, os quais são expressos de forma natural e concisa, concentrando-se apenas na resolução do problema em si, e sem se preocupar com artificialidades normalmente impostas pelos tradicionais paradigmas sequenciais, fornecendo desta forma um estilo de programação robusto e mais expressivo no tocante ao desenvolvimento de programas paralelos. Vale ressaltar que até onde nos é conhecido, nosso trabalho foi o primeiro a propor e implementar uma solução completa que unisse Gamma e GPUs em uma única plataforma de desenvolvimento. Como vimos na Seção 5.6, chegamos a identificar um trabalho passado que também propôs a integração entre estes dois conceitos, entretanto, o mesmo limitou-se a realizar dois testes de aplicações específicas, modelando a ideia de Gamma via implementações manualmente desenvolvidas sobre as GPUs. Em contrapartida, nossa solução é bem mais abrangente, e oferece uma efetiva integração entre Gamma e GPUs de forma genérica e transparente, por possuir um compilador que traduz os programas em Gamma e gera automaticamente código compatível com as GPUs.

Alguns pontos que ainda podem evoluir e ser melhorados foram aparecendo a medida em que o trabalho foi avançando durante seu desenvolvimento. Uma primeira limitação que identificamos no estado atual do nosso modelo diz respeito à sintaxe dos programas Gamma suportada pela implementação, uma vez que o compilador reconhece apenas os elementos previstos na concepção original do formalismo, não sendo capaz de trabalhar com todas as extensões linguísticas propostas para Gamma ao longo do tempo, como por exemplo, o suporte a novos tipos de dados presentes em Gamma Estruturada[1], ou ainda com as abordagens de alta ordem (*high-order*) de Gamma. Contudo, a extensão linguística que prevê o suporte a operadores de composição de programas Gamma de forma sequencial e paralela está presente na atual solução, o que já dá ao modelo uma capacidade aumentada de expressão sintática dos problemas.

Analisando a implementação sob a ótica de desempenho, podemos elencar outra limitação que acarreta em uma barreira aos ganhos obtidos, que é a falta de um escalonador adequado para os programas Gamma no ambiente distribuído. Isto ocorre porque o arcabouço arquitetural que herdamos da implementação base possui problemas importantes de desempenho, uma vez que trata-se de um escalonador muito simples, no qual não é suportada a execução concomitante entre várias reações de maneira eficiente. O principal ponto de limitação no desempenho do mesmo é a adoção de um modelo com controle centralizado, no qual temos uma célula que atua como regente do escalonamento, e acaba tornando-se o gargalo do sistema. Além disso, o esquema de execução sofre com um número muito elevado de mensagens trocadas entre as células na rede de interconexão do ambiente distribuído, o que faz o *tradeoff* comunicação/computação não ser vantajoso, pois a parcela de computação útil realizada a cada iteração é muito pequena se comparada com a quantia de tempo gasta com troca de mensagens. Este fato faz com que algumas aplicações executem mais rapidamente em uma implementação sequencial de Gamma do que nas versões distribuídas e paralelas, principalmente se as entradas não forem grandes o suficiente. Percebemos a intensidade desta limitação já em uma fase avançada do desenvolvimento da nova implementação *Gamma-GPU*, quando iniciamos os experimentos práticos utilizando entradas mais significativas. Considerando a complexidade de se implementar um modelo mais robusto de escalonamento com políticas mais elaboradas, constatamos que esta implementação demandaria muito tempo e por esta razão deixamos esta extensão como uma possibilidade de um trabalho futuro.

[1]Na realidade, como dissemos no Capítulo 4, já detemos uma implementação de Gamma Estruturada capaz de compilar tanto programas com tipos estruturados como também programas Gamma simples, contudo, ainda não integramos esta funcionalidade na versão com GPUs.

7.3 Trabalhos Futuros

Para finalizar esta dissertação, iremos expor agora algumas possibilidades de extensão do trabalho que identificamos ao longo do processo de desenvolvimento. Em termos de expressividade da linguagem Gamma, uma investida interessante seria ampliar a sintaxe suportada pelo compilador Gamma na implementação *Gamma-GPU*, de modo a possibilitar a interpretação de programas mais elaborados, seja pela utilização de mais de um multiconjunto por programa, pela presença de tipos estruturados propostos em Gamma Estruturada, pela presença de composições *High-Order* de Gamma, ou até mesmo por um conjunto de todas estas opções. Acreditamos que o desafio de adequar o processamento nas GPUs a estas extensões linguísticas de Gamma não seria pequeno, mas os resultados poderiam ser bastante significativos em termos de representatividade sintática dos algoritmos.

Em termos de desempenho, a solução para o comportamento descrito anteriormente como limitação seria realizar uma implementação mais elaborada de um escalonador para Gamma no ambiente distribuído nos moldes do proposto na Seção 4.5, para que seja usado em conjunto com a nossa implementação *Gamma-GPU*. Acreditamos que isso traria um grande ganho de desempenho ao modelo, pois o escalonador possivelmente resolveria o problema da não exploração adequada do paralelismo entre várias reações, uma vez que possibilitaria a execução concomitante destas atuando sobre o mesmo multiconjunto. Logo, esta seria a metade faltante para a melhoria de desempenho do modelo, visto que a outra metade, foi justamente o que atacamos neste trabalho, no qual aumentamos o paralelismo no contexto de uma única reação, valendo-se do poder computacional e dos *many-cores* presentes nas GPUs. Nesse mesmo contexto, seria interessante realizar mais experimentos a fim de comparar a implementação *Gamma-GPU* com a implementação sequencial de Gamma, para podermos obter resultados mais precisos e abrangentes quanto aos ganhos obtidos.

Ainda falando em termos de desempenho, outra possibilidade de evolução seria introduzirmos no contexto do uso das GPUs a aplicação das ações dos programas Gamma resultantes das reações que satisfizeram a condição de reação, de forma a explorar ainda mais o paralelismo sobre o multiconjunto no âmbito de uma única célula trabalhadora. Esta estratégia nos permitiria ampliar o potencial ganho de velocidade na execução dos programas Gamma, uma vez que englobaríamos uma parcela maior da computação a ser paralelizada, conforme preconizado pela "Lei de *Amdahl*"[26].

Conforme analisamos na Seção 6.3, em alguns programas escritos em Gamma pode ocorrer que o número de reações aptas a executar em paralelo seja muito reduzido, ocasião na qual o uso das GPUs sofre o risco de se tornar não profícuo,

causando mais sobrecarga do que ganho computacional. Para mitigar esta possível ineficiência no emprego das GPUs, identificamos uma outra extensão ao trabalho, que seria acrescentar uma nova estrutura sintática ao compilador que seja capaz de demarcar as reações específicas no código Gamma que devem fazer uso das GPUs. A ideia seria poder utilizar dentro da cláusula `where` dos programas Gamma, antes do identificador do nome da reação, uma nova palavra reservada modificadora '`gpu`' para indicar o uso das GPUs, como no exemplo a seguir:

```
...
where
R1 = replace ...  //reação sem GPU
gpu R2 = replace ... //reação com GPU
...
```

Nesse caso, a reação `R1` teria o processamento realizado em sua célula trabalhadora sem o auxílio das GPUs, ao passo que a reação `R2` estaria indicada para transmitir a computação às GPUs. Devemos enfatizar que esta solução é indicada somente nos casos onde o programador possui um amplo conhecimento sobre a configuração do problema que está sendo solucionado, entendendo a respeito do comportamento de cada reação e da quantidade e tipo de dados que serão manipulados no multiconjunto. Recorrendo novamente à Lei de *Amdahl*, podemos inferir que esta nova palavra modificadora deverá ser usada com cautela, pois sua ausência informa ao compilador para nunca executar uma determinada reação nas GPUs, o que consequentemente, diminuirá a parcela de código paralelizável no contexto das mesmas, acarretando numa perda do potencial ganho de desempenho que se poderia obter caso a palavra reservada modificadora fosse empregada.

Finalmente, outras futuras extensões que enxergamos como caminhos interessantes a serem investigados visando aumento de eficiência seriam, primeiro, aprimorar o código CUDA gerado pelo compilador, a fim de adotar práticas mais eficientes no uso da memória da GPU, através do uso de espaços de memória específicos para variáveis compartilhadas (*shared memory*) e constantes (*constant memory*), e segundo, buscar novas formas para dividir o trabalho entre as múltiplas *threads* na GPU, via implementação de novas heurísticas otimizadoras.

Referências Bibliográficas

[1] BANÂTRE, J.-P., LE MÉTAYER, D. *A New Computational Model and Its Discipline of Programming*. In: Rapport de Recherche 566, INRIA, France, 1986.

[2] BANÂTRE, J.-P., LE MÉTAYER, D. "The Gamma Model and Its Discipline of Programming", *Sci. Comput. Program.*, v. 15, n. 1, pp. 55–77, nov. 1990. ISSN: 0167-6423. doi: 10.1016/0167-6423(90)90044-E. Disponível em: <http://dx.doi.org/10.1016/0167-6423(90)90044-E>.

[3] BANÂTRE, J.-P., FRADET, P., MÉTAYER, D. L. "Gamma and the Chemical Reaction Model: Fifteen Years After". In: *Proceedings of the Workshop on Multiset Processing: Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View*, WMP '00, pp. 17–44, London, UK, UK, 2001. Springer-Verlag. ISBN: 3-540-43063-6. Disponível em: <http://dl.acm.org/citation.cfm?id=647269.721851>.

[4] SANDERS, J., KANDROT, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st ed. Boston, MA, USA, Addison-Wesley Professional, 2010. ISBN: 0131387685, 9780131387683.

[5] BURKS, A. W., GOLDSTINE, H. H., VON NEUMANN, J. "Perspectives on the Computer Revolution". Ablex Publishing Corp., cap. Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946), pp. 39–48, Norwood, NJ, USA, 1989. ISBN: 0-89391-369-3. Disponível em: <http://dl.acm.org/citation.cfm?id=98326.98337>.

[6] BANÂTRE, J.-P., LE MÉTAYER, D. "Programming by Multiset Transformation", *Commun. ACM*, v. 36, n. 1, pp. 98–111, jan. 1993. ISSN: 0001-0782. doi: 10.1145/151233.151242. Disponível em: <http://doi.acm.org/10.1145/151233.151242>.

[7] DIJKSTRA, E. W. *A Discipline of Programming*. 1st ed. Upper Saddle River, NJ, USA, Prentice Hall PTR, 1976. ISBN: 013215871X.

[8] BANÂTRE, J.-P., LE MÉTAYER, D. "Coordination Programming". Imperial College Press, cap. Gamma and the Chemical Reaction Model: Ten Years After, pp. 3–41, London, UK, UK, 1996. ISBN: 1-86094-023-4. Disponível em: <http://dl.acm.org/citation.cfm?id=270347.270348>.

[9] CHRISTIAN CREVEUIL, G. M. *Dérivation systématique d'un algorithme de segmentation d'images: un exemple d'application du formalisme GAMMA*. In: Rapport de Recherche 1049, INRIA, France, 1989.

[10] TANENBAUM, A. S., BOS, H. "Modern Operating Systems". 4th ed., cap. 6 - Deadlocks, Prentice Hall Press, 2014.

[11] DE MELLO JUNIOR, R. R. *Fusão de Dados em Gamma*. Dissertação de Mestrado, PESC/COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, 2015.

[12] HANKIN, C., LEMÉTAYER, D., SANDS, D. "A calculus of gamma programs". In: Banerjee, U., Gelernter, D., Nicolau, A., et al. (Eds.), *Languages and Compilers for Parallel Computing*, v. 757, *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 342–355, 1993. ISBN: 978-3-540-57502-3. doi: 10.1007/3-540-57502-2_57. Disponível em: <http://dx.doi.org/10.1007/3-540-57502-2_57>.

[13] SANDS, D. "A Compositional Semantics of Combining Forms for Gamma Programs". In: *INTERNATIONAL CONFERENCE ON FORMAL METHODS IN PROGRAMMING AND THEIR APPLICATIONS*, pp. 43–56. Springer-Verlag, 1993.

[14] MÉTAYER, D. L. "Higher-Order Multiset Programming". In: *Proc. of the DIMACS workshop on specifications of parallel algorithms*, Princeton, USA, 1994. American Mathematical Society.

[15] BANÂTRE, J. P., FRADET, P., RADENAC, Y. "Higher-Order Chemical Programming Style". In: *Proceedings of the 2004 International Conference on Unconventional Programming Paradigms*, UPP'04, pp. 84–95, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN: 3-540-27884-2, 978-3-540-27884-9. doi: 10.1007/11527800_7. Disponível em: <http://dx.doi.org/10.1007/11527800_7>.

[16] COHEN, D., FILHO, J. M. "Introducing a Calculus for Higher-Order Multiset Programming". In: *Proceedings of the First International Conference on Coordination Languages and Models*, COORDINATION '96, pp. 124–141,

London, UK, UK, 1996. Springer-Verlag. ISBN: 3-540-61052-9. Disponível em: <http://dl.acm.org/citation.cfm?id=647013.712992>.

[17] FRADET, P., LE MÉTAYER, D. "Structured Gamma", *Sci. Comput. Program.*, v. 31, n. 2-3, pp. 263–289, jul. 1998. ISSN: 0167-6423. doi: 10.1016/S0167-6423(97)00023-3. Disponível em: <http://dx.doi.org/10.1016/S0167-6423(97)00023-3>.

[18] CREVEUIL, C. "Implementation of Gamma on the Connection Machine". In: *Research Directions in High-Level Parallel Programming Languages*, pp. 219–230, London, UK, UK, 1992. Springer-Verlag. ISBN: 3-540-55160-3. Disponível em: <http://dl.acm.org/citation.cfm?id=646227.683816>.

[19] BANÂTRE, J.-P., COUTANT, A., METAYER, D. L. "A parallel machine for multiset transformation and its programming style", *Future Generation Computer Systems*, v. 4, n. 2, pp. 133 – 144, 1988. ISSN: 0167-739X. doi: http://dx.doi.org/10.1016/0167-739X(88)90012-X. Disponível em: <http://www.sciencedirect.com/science/article/pii/0167739X8890012X>.

[20] KUCHEN, H., GLADITZ, K. "Parallel Implementation of Bags". In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pp. 299–307, New York, NY, USA, 1993. ACM. ISBN: 0-89791-595-X. doi: 10.1145/165180.165226. Disponível em: <http://doi.acm.org/10.1145/165180.165226>.

[21] VIEILLOT, M. "Synthèse de programmes Gamma en logique reconfigurable", *Technique et science informatiques*, v. 14, n. 5, pp. 557–583, 1995.

[22] BANÂTRE, J.-P., FRADET, P., RADENAC, Y. "Generalised Multisets for Chemical Programming", *Mathematical. Structures in Comp. Sci.*, v. 16, n. 4, pp. 557–580, ago. 2006. ISSN: 0960-1295. doi: 10.1017/S0960129506005317. Disponível em: <http://dx.doi.org/10.1017/S0960129506005317>.

[23] MOORE, G. E. "Readings in Computer Architecture". Morgan Kaufmann Publishers Inc., cap. Cramming More Components Onto Integrated Circuits, pp. 56–59, San Francisco, CA, USA, 2000. ISBN: 1-55860-539-8. Disponível em: <http://dl.acm.org/citation.cfm?id=333067.333074>.

[24] PATTERSON, D. A., ASANOVIC, K., BODIK, R., et al. *The Landscape of Parallel Computing Research: A View from Berkeley.* Relatório Técnico

UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>.

[25] KIRK, D. B., HWU, W.-M. W. *Programming Massively Parallel Processors: A Hands-on Approach*. 2 ed. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2013. ISBN: 9780123914187.

[26] HENNESSY, J. L., PATTERSON, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th ed. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X, 9780123838728.

[27] PACHECO, P. *An Introduction to Parallel Programming*. 1st ed. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2011. ISBN: 9780123742605.

[28] FARBER, R. *CUDA Application Design and Development*. 1st ed. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123884268, 9780123884329.

[29] GASTER, B., HOWES, L., KAELI, D. R., et al. *Heterogeneous Computing with OpenCL*. 1st ed. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2011. ISBN: 0123877660, 9780123877666.

[30] DONGARRA, J., LASTOVETSKY, A. L. *High Performance Heterogeneous Computing*. New York, NY, USA, Wiley-Interscience, 2009. ISBN: 0470040394, 9780470040393.

[31] JEFFERS, J., REINDERS, J. *Intel Xeon Phi Coprocessor High Performance Programming*. 1st ed. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2013. ISBN: 9780124104143, 9780124104945.

[32] INTEL CORPORATION. "Intel® Xeon Phi™ Coprocessor 5110P". 2015. Disponível em: <http://ark.intel.com/pt-br/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1_053-GHz-60-core>.

[33] NVIDIA CORPORATION. *TESLA K20X GPU ACCELERATOR - Board Specification*, Novembro 2012. Disponível em: <http://www.nvidia.com/content/PDF/kepler/Tesla-K20X-BD-06397-001-v05.pdf>.

[34] NVIDIA CORPORATION. "Just The Facts". 2015. Disponível em: <http://www.nvidia.com/object/justthefacts.html>.

[35] FANG, J., VARBANESCU, A. L., IMBERN'ON, B., et al. "Parallel Computation of Non-Bonded Interactions in Drug Discovery: Nvidia GPUs vs. Intel Xeon Phi". In: *International Work-Conference on Bioinformatics and Biomedical Engineering (IWBBIO)*, 2014.

[36] KACZMAREK, O., SCHMIDT, C., STEINBRECHER, P., et al. ""HISQ inverter on Intel Xeon Phi and NVIDIA GPUs"", *ArXiv e-prints*, sep 2014.

[37] JEONG, H., LEE, W., PAK, J., et al. "Performance of Kepler GTX Titan GPUs and Xeon Phi System." *CoRR*, v. abs/1311.0590, 2013. Disponível em: <http://dblp.uni-trier.de/db/journals/corr/corr1311.html#JeongLPCPYKLL13>.

[38] STROHMAIER, E., DONGARRA, J., SIMON, H., et al. "TOP 500 LIST". novembro 2014. Disponível em: <http://www.top500.org/lists/2014/11/>.

[39] ADVANCED MICRO DEVICES INC. *AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE*. White paper, AMD, 2012. Disponível em: <https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf>.

[40] NVIDIA CORPORATION. *NVIDIA CUDA C Programming Guide*, Agosto 2014. Disponível em: <http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf>.

[41] NVIDIA CORPORATION. *PARALLEL THREAD EXECUTION ISA Application Guide*, Março 2015. Disponível em: <http://docs.nvidia.com/cuda/pdf/ptx_isa_4.2.pdf>.

[42] NVIDIA CORPORATION. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. White paper, NVIDIA Corporation, 2012. Disponível em: <http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf>.

[43] NVIDIA CORPORATION. "GPU Technology Conference". 2015. Disponível em: <http://blogs.nvidia.com/blog/2015/03/16/live-gtc/>.

[44] NVIDIA CORPORATION. *NVIDIA GeForce GTX 980*. White paper, NVIDIA Corporation, 2014. Disponível em: <http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF>.

[45] PAILLARD, G. A. L. *Uma Implementação Paralela e Distribuída de Gamma Estruturada*. Dissertação de Mestrado, PESC/COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, 1999.

[46] PAILLARD, G., FRANCA, F., FILHO, J. "A distributed implementation of Structured Gamma". In: *Parallel and Distributed Systems, 2001. ICPADS 2001. Proceedings. Eighth International Conference on*, pp. 445–450, 2001. doi: 10.1109/ICPADS.2001.934852.

[47] LEVINE, J., MASON, T., BROWN, D. *Lex & Yacc, 2Nd Edition*. Second ed. California, USA, O'Reilly, 1992. ISBN: 9781565920002.

[48] FRANCA, F., FILHO, J., PAILLARD, G. "Uma Proposta de um Escalonador para Gamma". In: *II Workshop em Sistemas Computacionais de Alto Desempenho*, pp. 47–54, Pirenópolis, GO, 2001.

[49] FRANÇA, F. M. G., FARIA, L. "Optimal Mapping of Neighbourhood-Constrained Systems". In: *Proceedings of the Second International Workshop on Parallel Algorithms for Irregularly Structured Problems*, IRREGULAR '95, pp. 165–170, London, UK, UK, 1995. Springer-Verlag. ISBN: 3-540-60321-2. Disponível em: <`http://dl.acm.org/citation.cfm?id=646009.676841`>.

[50] BARBOSA, V. C. *Massively Parallel Models of Computation: Distributed Parallel Processing in Artificial Intelligence and Optimisation*. Upper Saddle River, NJ, USA, Ellis Horwood, 1993. ISBN: 0-13-562968-3.

[51] MÉTAYER, D. L. "Higher-Order Multiset Programming". In: *in Proc. of the DIMACS workshop on specifications of parallel algorithms, American Mathematical Society, Dimacs series in Discrete Mathematics*. American Mathematical Society, 1994.

[52] SZWARCFITER, J., MARKENZON, L. *Estruturas de dados e seus algoritmos*. 3 ed. Rio de Janeiro, RJ, LTC, 2010. ISBN: 9788521617501.

[53] LIN, H., KEMP, J., GILBERT, P. "Computing Gamma Calculus on Computer Cluster", *Int. J. Technol. Diffus.*, v. 1, n. 4, pp. 42–52, out. 2010. ISSN: 1947-9301. doi: 10.4018/jtd.2010100104. Disponível em: <`http://dx.doi.org/10.4018/jtd.2010100104`>.

[54] HUANG, L., TONG, W., KAM, W., et al. "Implementation of GAMMA on a massively parallel computer", *Journal of Computer Science and Technology*, v. 12, n. 1, pp. 29–39, 1997. ISSN: 1000-9000. doi:

10.1007/BF02943142. Disponível em: <http://dx.doi.org/10.1007/BF02943142>.

[55] GHANEMI, S., DAMEGH, A. A. A. "Global GAMMA: A Distributed Implementation of GAMMA Using Global Computing", *Journal of King Saud University - Computer and Information Sciences*, v. 16, n. 0, pp. 67 – 83, 2004. ISSN: 1319-1578. doi: http://dx.doi.org/10.1016/S1319-1578(04)80009-0. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1319157804800090>.

[56] LUCAS, P., FRITZ, N., WILHELM, R. "The Development of the Data-Parallel GPU Programming Language CGiS". In: *6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part IV*, pp. 200–203, UK, 2006. Springer Berlin Heidelberg. ISBN: 978-3-540-34386-8.

[57] LAWLOR, O. "Embedding OpenCL in C++ for Expressive GPU Programming", *IPDPS WOLFHPC, May*, 2011.

[58] HOLK, E., BYRD, W., MAHAJAN, N., et al. "Declarative Parallel Programming for GPUs". In: *International Conference on Parallel Computing (ParCo)*, Ghent, Belgium, 2011.

[59] STROMME, A., CARLSON, R., NEWHALL, T. "Chestnut: A GPU Programming Language for Non-experts". In: *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '12, pp. 156–167, New York, NY, USA, 2012. ACM. ISBN: 978-1-4503-1211-0. doi: 10.1145/2141702.2141720. Disponível em: <http://doi.acm.org/10.1145/2141702.2141720>.

[60] PINHEIRO, A., DE CARVALHO JUNIOR, F., ARRUDA, N., et al. "Fusion: Abstractions for Multicore/Manycore Heterogenous Parallel Programming Using GPUs". In: Quintão Pereira, F. (Ed.), *Programming Languages*, v. 8771, *Lecture Notes in Computer Science*, Springer International Publishing, pp. 109–123, 2014. ISBN: 978-3-319-11862-8. doi: 10.1007/978-3-319-11863-5_8. Disponível em: <http://dx.doi.org/10.1007/978-3-319-11863-5_8>.

[61] LEE, S., CHAKRAVARTY, M. M. T., GROVER, V., et al. "GPU Kernels as Data-Parallel Array Computations in Haskell". In: *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPHAM)*, Seattle, WA, 2009.

[62] GROSSMAN, M., SBÎRLEA, A. S., BUDIMLIĆ, Z., et al. "CnC-CUDA: Declarative Programming for GPUs". In: *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, LCPC'10, pp. 230–245, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN: 978-3-642-19594-5. Disponível em: <http://dl.acm.org/citation.cfm?id=1964536.1964552>.

[63] HOLK, E., PATHIRAGE, M., CHAUHAN, A., et al. "GPU Programming in Rust: Implementing High-Level Abstractions in a Systems-Level Language". In: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pp. 315–324, Washington, DC, USA, 2013. IEEE Computer Society. ISBN: 978-0-7695-4979-8. doi: 10.1109/IPDPSW.2013.173. Disponível em: <http://dx.doi.org/10.1109/IPDPSW.2013.173>.

[64] SCHAETZ, S., UECKER, M. "A multi-GPU Programming Library for Real-time Applications". In: *Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ICA3PP'12, pp. 114–128, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN: 978-3-642-33077-3. doi: 10.1007/978-3-642-33078-0_9. Disponível em: <http://dx.doi.org/10.1007/978-3-642-33078-0_9>.

[65] "CENAPAD UFC". 2015. Disponível em: <http://www.cenapad.ufc.br/quem-somos/recursos>.

Apêndice A

Exemplo de Código Fonte Gerado pelo Compilador Gamma

A.1 Números Primos

Código Gamma

/* primos.gm */

```
primos { 2,3,4,...,N }
where
primos = replace x1, x2
         by x2
         if ((x1 % x2) == 0)
```

A.1.1 Gamma-GPU

Código Gerado (run.c)

```
1 # include <stdio.h>
2 # include <sys/types.h>
3 # include <sys/time.h>
4 # include <mpi.h>
5 # include "../include/gpu.h"
6 # include "../include/const.h"
7 # include "../include/types.h"
8 # include "../include/rtbag.h"
9 # include "../include/fnc.h"
```

```
10 # include "../include/msg.h"
11 # include "../include/vars.h"
12
13 extern void cell_0();
14 extern void cell_1();
15 extern int * cell_2_reaction_gpu_call_fnc(int, int);
16
17 void cell_2()
18 {
19   RT_VALUE *GenValueList();
20   int result;
21   register RT_VALUE *link, *linkhead;
22   MPI_Status status;
23   int false_buffer = FALSE;
24   int tag, ierr;
25
26   void ReceiveBag();
27   int RequestBag();
28
29   ierr = gpu_init(2);
30   if (ierr != GPU_SUCCESS) {
31     fprintf(stderr, "GPU initialisation error on cell 2");
32     exit(1);
33   }
34
35   for (;;) {
36     result = FALSE;
37     MPI_Recv(NULL, 0, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
           MPI_COMM_WORLD, &status);
38     tag = status.MPI_TAG;
39     if (tag == DIE)
40       return;
41     for (; !RequestBag(0););
42     ReceiveBag(0);
43
44     /* Code Section for GPU Processing */
45     int reaction_size = 2;
46     int reaction_num_vals = 2;
47     int base_type = TY_INT;
48     int base_type_size = 1;
49
50     BagToBagArray(RunTimeBag, reaction_size, reaction_num_vals,
           base_type, base_type_size);
51
52     /* num_elements = número de elementos da BAG
53      * num_elements_base = número de elementos base da BAG - será
           criada na GPU uma thread por elemento base */
```

```
54      int num_elements = RunTimeBagArray->num_elements;
55      int num_elements_base = RunTimeBagArray->num_base;
56
57      /* verifica se há elementos suficientes para testar ao menos
            uma reação*/
58      if (num_elements_base > 0 && num_elements >= reaction_size) {
59
60        /* chama função que invoca o kernel*/
61        int *response = (int *) cell_2_reaction_gpu_call_fnc(
              num_elements, num_elements_base);
62
63        int i, j, k;
64        /* Trata resposta recebida da GPU executando as Actions para
              cada Reacão cuja condicão foi satisfeita */
65        RT_BAG *actions_bag = NULL;
66        int *already_reacted = (int *) Alloc(num_elements * sizeof(
              int));
67        memset(already_reacted, FALSE, num_elements * sizeof(int));
68        RT_ELEMENT **ptr = (RT_ELEMENT **) Alloc(reaction_num_vals *
              sizeof(RT_ELEMENT*));
69
70        for (i = 0; i < num_elements_base; i++) {
71          if (ElementReacts(i, response, already_reacted,
                reaction_size) == TRUE) {
72            result = TRUE;
73            int idx, idx_base, ptr_idx;
74            ptr_idx = 0;
75            for (j = 0; j < reaction_size; j++) {
76              idx = getElement_h(response, i, j, reaction_size);
77              idx_base = RunTimeBagArray->offset[idx];
78              for (k = 0; k < RunTimeBagArray->elem_size[idx]; k++) {
79                ptr[ptr_idx++] = CreateRTElement(RunTimeBagArray->
                      data[idx_base + k]);
80              }
81            }
82            linkhead = link = GenValueList(2);
83            link->rtv_val = CopyRTElement(ptr[0]);
84            link->rtv_name = Alloc(3);
85            strcpy(link->rtv_name, "x1");
86            link = link->rtv_next;
87            link->rtv_val = CopyRTElement(ptr[1]);
88            link->rtv_name = Alloc(3);
89            strcpy(link->rtv_name, "x2");
90            link = link->rtv_next;
91            cell_2_action(linkhead, &actions_bag);
92            for (j = 0; j < reaction_num_vals; j++)
93              Free(ptr[j]);
```

```
94          }
95        }
96        Free(ptr);
97        if (result == TRUE) {
98          /* Remove nodes que reagiram da RunTimeBag */
99          RemoveReactedNodes(&RunTimeBag, already_reacted,
              num_elements);
100       }
101       /* Faz merge das listas RunTimeBag e actions */
102       MergeBags(&RunTimeBag, actions_bag);
103       Free(response);
104       Free(already_reacted);
105     }
106     DeleteBagArray();
107     /* End Code Section for GPU Processing */
108
109     MPI_Send(NULL, 0, MPI_INT, 0, BAG_SEND, MPI_COMM_WORLD);
110     SendBag(0);
111     MPI_Send(&result, 1, MPI_INT, 1, BR_RESPONSE, MPI_COMM_WORLD);
112   }
113 } /* end cell_2 */
114
115 cell_2_action(RT_VALUE *link, RT_BAG **rtb)
116 {
117   cell_2_action_0(link, rtb);
118   FreeListValues(link);
119 } /* end cell_2_action */
120
121 cell_2_action_0_e( x2)
122 {
123   register int Result = 0;
124   Result = x2;
125   return (Result);
126 } /* end cell_2_action_0_e */
127
128 cell_2_action_0(RT_VALUE *link, RT_BAG **rtb)
129 {
130   register int x2 = GetValue("x2", link);
131   register int result = 0;
132   result = cell_2_action_0_e(x2);
133   AddRTBagNode(TY_INT, result, rtb);
134 } /* end cell_2_action_0 */
135
136 void main(int argc, char *argv[])
137 {
138   void (*c[3])();
139   int mynumber, ierr;
```

```
140
141   c[0] = cell_0;
142   c[1] = cell_1;
143   c[2] = cell_2;
144
145   ierr = MPI_Init(&argc, &argv);
146   if (ierr != MPI_SUCCESS) {
147   fprintf(stderr, "MPI initialisation error");
148   exit(1);
149   }
150   MPI_Comm_rank(MPI_COMM_WORLD, &mynumber);
151   (c[mynumber])();
152   MPI_Finalize();
153 } /* end main */
```

Código Gerado (run_gpu.cu)

```
1  # include <stdio.h>
2  # include <stdlib.h>
3  # include <math.h>
4  # include "../include/const.h"
5  # include "../include/types.h"
6  # include "../include/rtbag.h"
7  # include "../include/vars.h"
8  # include "../include/gpu.h"
9
10 // Set a matrix element
11 __device__ void setElement(int *A, int row, int col, int stride, int
       value)
12 {
13   A[row * stride + col] = value;
14 }
15
16 // Get a matrix element
17 __device__ int getElement(const int *A, int row, int col, int stride
     )
18 {
19   return A[row * stride + col];
20 }
21
22 extern "C" {
23 int * cell_2_reaction_gpu_call_fnc(int, int);
24 }
```

```
25
26 __global__ void cell_2_reaction_kernel_fnc(int *, int *, int *, int
       *, int*, int*, int*, int*, int*);
27
28 int * cell_2_reaction_gpu_call_fnc(int num_elements, int
       num_elements_base)
29 {
30   int numBlocks = (int) ceil((float) num_elements_base /
       GPU_MAX_THREADS_PER_BLOCK);
31   int threadsPerBlock = (int) ceil((float) num_elements_base /
       numBlocks);
32
33   int *d_array_data = 0, *d_response = 0, *d_reaction_size = 0, *
       d_already_reacted = 0, *d_offset = 0, *d_elem_size = 0, *
       d_indice_base = 0, *d_num_elements = 0, *d_num_base = 0;
34
35   gpu_pre_reaction_call(num_elements, num_elements_base, &
       d_array_data, &d_reaction_size, &d_response, &
       d_already_reacted, &d_offset, &d_elem_size, &d_indice_base, &
       d_num_elements, &d_num_base);
36
37   cell_2_reaction_kernel_fnc<<<numBlocks, threadsPerBlock>>>(
       d_array_data, d_reaction_size, d_response, d_already_reacted,
       d_offset, d_elem_size, d_indice_base, d_num_elements,
       d_num_base);
38
39   int *h_response = gpu_post_reaction_call(num_elements_base, &
       d_array_data, &d_reaction_size, &d_response, &
       d_already_reacted, &d_offset, &d_elem_size, &d_indice_base, &
       d_num_elements, &d_num_base);
40
41   return h_response;
42 }
43
44 __device__ int cell_2_reaction_gpu_fnc(int, int);
45
46 __global__ void cell_2_reaction_kernel_fnc(int *bag_array, int *
       reaction_size, int *response, int *already_reacted, int *offset,
        int *elem_size, int *indice_base, int *num_elements, int *
       num_base)
47 {
48   int r_size = *reaction_size;
49   int num_elem = *num_elements;
50
51   //índice da thread
52   int thread_indice = (blockDim.x * blockIdx.x) + (threadIdx.x);
53   //verifica thread overflow
```

```
54    if (thread_indice > ((*num_base) - 1))
55      return;
56
57    //índice do elemento base
58    int indice = indice_base[thread_indice];
59    //ajusta índice base para acessar array
60    int idx_base = offset[indice];
61
62    int x_0 = bag_array[idx_base + 0];
63
64    for (int i_0 = indice + 1; i_0 < num_elem; i_0++) {
65      if (already_reacted[i_0] == TRUE)
66        continue;
67      if (elem_size[i_0] != 1)
68        continue;
69      int idx_i_0 = offset[i_0];
70      int x_1 = bag_array[idx_i_0 + 0];
71
72      if (already_reacted[indice] == TRUE)
73        return; //Elemento base já reagiu em outra thread
74
75      int result = cell_2_reaction_gpu_fnc(x_0, x_1);
76
77      if (result == TRUE) {
78        //primeira coluna á sempre o próprio elemento base
79        setElement(response, thread_indice, 0, r_size, indice);
80        setElement(response, thread_indice, 1, r_size, i_0);
81        already_reacted[indice] = TRUE;
82        already_reacted[i_0] = TRUE;
83        return;
84      }
85    }
86
87    for (int i_0 = indice - 1; i_0 >= 0; i_0--) {
88      if (already_reacted[i_0] == TRUE)
89        continue;
90      if (elem_size[i_0] != 1)
91        continue;
92      int idx_i_0 = offset[i_0];
93      int x_1 = bag_array[idx_i_0 + 0];
94
95      if (already_reacted[indice] == TRUE)
96        return; //Elemento base já reagiu em outra thread
97
98      int result = cell_2_reaction_gpu_fnc(x_0, x_1);
99
100     if (result == TRUE) {
```

```
101         //primeira coluna á sempre o próprio elemento base
102         setElement(response, thread_indice, 0, r_size, indice);
103         setElement(response, thread_indice, 1, r_size, i_0);
104         already_reacted[indice] = TRUE;
105         already_reacted[i_0] = TRUE;
106         return;
107       }
108     }
109 }
110
111 __device__ int cell_2_reaction_gpu_fnc(int x1, int x2)
112 {
113   register int Result = 0;
114   Result = ((x1 % x2) == 0);
115   return (Result);
116 } /* end __device__ int cell_2_reaction_gpu_fnc */
```

Trechos de códigos auxiliares que compõem o *framework*

```
1  int gpu_init(int cell)
2  {
3    int gpu, n_gpus;
4    CHECK_ERROR(cudaGetDeviceCount(&n_gpus));
5    if(n_gpus <= 0)
6      return GPU_ERROR;
7    gpu = cell % n_gpus;
8
9    CHECK_ERROR(cudaSetDevice(gpu));
10   CHECK_ERROR(cudaDeviceReset());
11   CHECK_ERROR(cudaDeviceGetAttribute(&GPU_MAX_THREADS_PER_BLOCK,
         cudaDevAttrMaxThreadsPerBlock, gpu));
12
13   return GPU_SUCCESS;
14 }
15 //-------
16
17 void gpu_pre_reaction_call(int num_elements, int num_elements_base,
       int **d_array_data, int **d_reaction_size, int **d_response,
      int **d_already_reacted, int **d_offset, int **d_elem_size, int
      **d_indice_base, int **d_num_elements, int **d_num_base)
18 {
19   CHECK_ERROR(cudaMalloc((void**) d_array_data, RunTimeBagArray->
       size * sizeof(int)));
20   CHECK_ERROR(cudaMalloc((void**) d_response, num_elements_base *
       RunTimeBagArray->reaction_size * sizeof(int)));
21   CHECK_ERROR(cudaMalloc((void**) d_reaction_size, sizeof(int)));
```

```
22    CHECK_ERROR(cudaMalloc((void**) d_already_reacted, num_elements *
          sizeof(int)));
23    CHECK_ERROR(cudaMalloc((void**) d_offset, num_elements * sizeof(
          int)));
24    CHECK_ERROR(cudaMalloc((void**) d_elem_size, num_elements *
          sizeof(int)));
25    CHECK_ERROR(cudaMalloc((void**) d_indice_base, num_elements_base
          * sizeof(int)));
26    CHECK_ERROR(cudaMalloc((void**) d_num_elements, sizeof(int)));
27    CHECK_ERROR(cudaMalloc((void**) d_num_base, sizeof(int)));
28
29    CHECK_ERROR(cudaMemset(*d_response, NOT_REACTED,
          num_elements_base * RunTimeBagArray->reaction_size * sizeof(
          int)));
30    CHECK_ERROR(cudaMemset(*d_already_reacted, FALSE, num_elements *
          sizeof(int)));
31
32    CHECK_ERROR(cudaMemcpy(*d_array_data, RunTimeBagArray->data,
          RunTimeBagArray->size * sizeof(int), cudaMemcpyHostToDevice));
33    CHECK_ERROR(cudaMemcpy(*d_reaction_size, &(RunTimeBagArray->
          reaction_size), sizeof(int), cudaMemcpyHostToDevice));
34    CHECK_ERROR(cudaMemcpy(*d_offset, RunTimeBagArray->offset,
          num_elements * sizeof(int), cudaMemcpyHostToDevice));
35    CHECK_ERROR(cudaMemcpy(*d_elem_size, RunTimeBagArray->elem_size,
          num_elements * sizeof(int), cudaMemcpyHostToDevice));
36    CHECK_ERROR(cudaMemcpy(*d_indice_base, RunTimeBagArray->
          indice_base, num_elements_base * sizeof(int),
          cudaMemcpyHostToDevice));
37    CHECK_ERROR(cudaMemcpy(*d_num_elements, &(RunTimeBagArray->
          num_elements), sizeof(int), cudaMemcpyHostToDevice));
38    CHECK_ERROR(cudaMemcpy(*d_num_base, &(RunTimeBagArray->num_base),
          sizeof(int), cudaMemcpyHostToDevice));
39  }
40  //-------
41
42  int * gpu_post_reaction_call(int num_elements_base, int **
        d_array_data, int **d_reaction_size, int **d_response, int **
        d_already_reacted, int **d_offset, int **d_elem_size, int **
        d_indice_base, int **d_num_elements, int **d_num_base)
43  {
44    int * h_response = (int *)Alloc(num_elements_base *
          RunTimeBagArray >reaction_size * sizeof(int));
45    CHECK_ERROR(cudaMemcpy(h_response, *d_response, num_elements_base
          * RunTimeBagArray->reaction_size * sizeof(int),
          cudaMemcpyDeviceToHost));
46
47    CHECK_ERROR(cudaFree(*d_array_data));
```

```
48    CHECK_ERROR(cudaFree(*d_reaction_size));
49    CHECK_ERROR(cudaFree(*d_response));
50    CHECK_ERROR(cudaFree(*d_already_reacted));
51    CHECK_ERROR(cudaFree(*d_offset));
52    CHECK_ERROR(cudaFree(*d_elem_size));
53    CHECK_ERROR(cudaFree(*d_indice_base));
54    CHECK_ERROR(cudaFree(*d_num_elements));
55    CHECK_ERROR(cudaFree(*d_num_base));
56
57    return h_response;
58  }
59  //-------
60
61  void cell_0 ()
62  {
63    int bag_free;
64    MPI_Status status;
65    int true_buffer = TRUE, false_buffer = FALSE;
66    int cell, tag;
67    int dummy = 0;
68
69    CreateInitialBag();
70    bag_free = TRUE;
71
72    for (;;) {
73      MPI_Recv(NULL, 0, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
          MPI_COMM_WORLD, &status);
74      cell = status.MPI_SOURCE;
75      tag = status.MPI_TAG;
76      switch (tag) {
77
78      case BAG_SEND:
79        ReceiveBag(cell);
80        bag_free = TRUE;
81        break;
82
83      case BAG_REQUEST:
84        if (bag_free) {
85          bag_free = FALSE;
86          MPI_Send(&true_buffer, 1, MPI_INT, cell, BAG_RESPONSE,
              MPI_COMM_WORLD);
87          SendBag(cell);
88        }
89        else
90         MPI_Send(&false_buffer, 1, MPI_INT, cell, BAG_RESPONSE,
              MPI_COMM_WORLD);
91        break;
```

```
92
93      case BAG_PRINT:
94        if (bag_free) {
95          PrintRTBag(RunTimeBag); printf("\n");
96          return;
97        }
98        break;
99      }
100   }
101 } /* end cell_0 */
102 //-------
103
104 void cell_1 ()
105 {
106   int response, processes, i;
107   MPI_Status status;
108
109   MPI_Comm_size(MPI_COMM_WORLD, &processes);
110
111   for (;;) {
112     MPI_Send(NULL, 0, MPI_INT, 2, BR_DOIT, MPI_COMM_WORLD);
113     MPI_Recv(&response, 1, MPI_INT, 2, BR_RESPONSE, MPI_COMM_WORLD,
          &status);
114     if (response == FALSE)
115       break;
116   }
117   for (i = 2;i < processes; ++i)
118     MPI_Send(NULL, 0, MPI_INT, i, DIE, MPI_COMM_WORLD);
119
120   MPI_Send(NULL, 0, MPI_INT, 0, BAG_PRINT, MPI_COMM_WORLD);
121 } /* end cell_1 */
```

A.1.2 Gamma-Base

Código Gerado (run.c)

```
1  # include <stdio.h>
2  # include <sys/types.h>
3  # include <sys/time.h>
4  # include <mpi.h>
5  # include "../include/const.h"
6  # include "../include/types.h"
7  # include "../include/rtbag.h"
8  # include "../include/fnc.h"
9  # include "../include/msg.h"
10 # include "../include/vars.h"
```

```
11
12 extern void cell_0();
13 extern void cell_1();
14
15 void cell_2() {
16   RT_VALUE *GenValueList();
17   register RT_BAG *ptr_0;
18   register int x_0;
19   register RT_BAG *ptr_1;
20   register int x_1;
21
22   int result;
23   register RT_VALUE *link, *linkhead;
24   MPI_Status status;
25   int false_buffer = FALSE;
26   int tag, ierr;
27
28   void ReceiveBag();
29   int RequestBag();
30   for (;;) {
31     result = FALSE;
32     MPI_Recv(NULL, 0, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
          MPI_COMM_WORLD,
33          &status);
34     tag = status.MPI_TAG;
35     if (tag == DIE)
36       return;
37     for (; !RequestBag(0);)
38       ;
39     ReceiveBag(0);
40     for (ptr_0 = RunTimeBag;
41          (ptr_0 != (RT_BAG *) NULL ) && (result == FALSE);
42          ptr_0 = ptr_0->rtb_next) {
43       if (!NODE_LOCKED(ptr_0->rtb_tag)
44           && (NODE_TYPE(ptr_0->rtb_tag) == TY_INT)) {
45         LOCK_NODE(ptr_0->rtb_tag);
46         x_0 = ptr_0->rtb_element->rte_val;
47       } else
48         continue;
49       for (ptr_1 = RunTimeBag;
50            (ptr_1 != (RT_BAG *) NULL ) && (result == FALSE); ptr_1 =
51                ptr_1->rtb_next) {
52         if (!NODE_LOCKED(ptr_1->rtb_tag)
53             && (NODE_TYPE(ptr_1->rtb_tag) == TY_INT)) {
54           LOCK_NODE(ptr_1->rtb_tag);
55           x_1 = ptr_1->rtb_element->rte_val;
56         } else
```

```
57            continue;
58          result = cell_2_reaction_fnc(x_0, x_1);
59          if (result == TRUE) {
60            linkhead = link = GenValueList(2);
61            link->rtv_val = CopyRTElement(ptr_0->rtb_element);
62            link->rtv_name = Alloc(3);
63            strcpy(link->rtv_name, "x1");
64            link = link->rtv_next;
65            link->rtv_val = CopyRTElement(ptr_1->rtb_element);
66            link->rtv_name = Alloc(3);
67            strcpy(link->rtv_name, "x2");
68            link = link->rtv_next;
69            cell_2_action(linkhead);
70          }
71          if (result == FALSE)
72            UNLOCK_NODE(ptr_1->rtb_tag);
73        }
74        if (result == FALSE)
75          UNLOCK_NODE(ptr_0->rtb_tag);
76      }
77      if (result == FALSE)
78        UnlockBagNodes();
79      MPI_Send(NULL, 0, MPI_INT, 0, BAG_SEND, MPI_COMM_WORLD );
80      SendBag(0);
81      MPI_Send(&result, 1, MPI_INT, 1, BR_RESPONSE, MPI_COMM_WORLD );
82    }
83  } /* end cell_2 */
84
85  cell_2_reaction_fnc(int x1, int x2) {
86    register int Result = 0;
87    Result = ((x1 % x2) == 0);
88    return (Result);
89  } /* end cell_2_reaction_fnc */
90
91  cell_2_action(link)
92    RT_VALUE *link; {
93    cell_2_action_0(link);
94    RemoveLockedBagNodes();
95    FreeListValues(link);
96  } /* end cell_2_action */
97
98  cell_2_action_0_e( x2) {
99    register int Result = 0;
100   Result = x2;
101   return (Result);
102 } /* end cell_2_action_0_e */
103
```

```
cell_2_action_0(link)
  RT_VALUE *link; {
  register int x2 = GetValue("x2", link);
  register int result = 0;
  result = cell_2_action_0_e(x2);
  AddRTBagNode(TY_INT, result, &RunTimeBag);
} /* end cell_2_action_0 */

void main(argc, argv)
  int argc;char *argv[]; {
  void (*c[3])();
  int mynumber, ierr;

  c[0] = cell_0;
  c[1] = cell_1;
  c[2] = cell_2;
  ierr = MPI_Init(&argc, &argv);
  if (ierr != MPI_SUCCESS) {
    fprintf(stderr, "MPI initialisation error");
    exit(1);
  }
  MPI_Comm_rank(MPI_COMM_WORLD, &mynumber);
  (c[mynumber])();
  MPI_Finalize();
} /* end main */
```

A.1.3 Gamma-Seq

Código Gerado (run.c)

```
# include <stdio.h>
# include <sys/types.h>
# include <sys/time.h>
# include "../include/const.h"
# include "../include/types.h"
# include "../include/rtbag.h"
# include "../include/fnc.h"
# include "../include/vars.h"

Gamma() {
  for (;;) {
    if (gamma_reaction() == TRUE)
      continue;
    break;
  }
  PrintRTBag();
```

```
17 } /* end Gamma */
18
19 sieve_reaction() {
20   RT_VALUE *GenValueList();
21   register RT_BAG *ptr_0;
22   register int x_0;
23   register RT_BAG *ptr_1;
24   register int x_1;
25   register int result;
26   register RT_VALUE *link, *linkhead;
27
28   for (ptr_0 = RunTimeBag; ptr_0 != (RT_BAG *) NULL ; ptr_0 =
29       ptr_0->rtb_next) {
30     if (!NODE_LOCKED(ptr_0->rtb_tag)
31         && (NODE_TYPE(ptr_0->rtb_tag) == TY_INT)) {
32       LOCK_NODE(ptr_0->rtb_tag);
33       x_0 = ptr_0->rtb_element->rte_val;
34     } else
35       continue;
36     for (ptr_1 = RunTimeBag; ptr_1 != (RT_BAG *) NULL ;
37         ptr_1 = ptr_1->rtb_next) {
38       if (!NODE_LOCKED(ptr_1->rtb_tag)
39           && (NODE_TYPE(ptr_1->rtb_tag) == TY_INT)) {
40         LOCK_NODE(ptr_1->rtb_tag);
41         x_1 = ptr_1->rtb_element->rte_val;
42       } else
43         continue;
44       result = sieve_reaction_fnc(x_0, x_1);
45       if (result == TRUE) {
46         linkhead = link = GenValueList(2);
47         link->rtv_val = ptr_0->rtb_element;
48         link->rtv_name = Alloc(3);
49         strcpy(link->rtv_name, "x1");
50         link = link->rtv_next;
51         link->rtv_val = ptr_1->rtb_element;
52         link->rtv_name = Alloc(3);
53         strcpy(link->rtv_name, "x2");
54         link = link->rtv_next;
55         sieve_action(linkhead);
56         return (TRUE);
57       }
58       UNLOCK_NODE(ptr_1->rtb_tag);
59     }
60     UNLOCK_NODE(ptr_0->rtb_tag);
61   }
62   UnlockBagNodes();
63   return (FALSE);
```

```
} /* end sieve_reaction */

sieve_reaction_fnc( x1, x2) {
  register int Result = 0;
  Result = ((x1 % x2) == 0);
  return (Result);
} /* end sieve_reaction_fnc */

sieve_action(link)
  RT_VALUE *link; {
  sieve_action_0(link);
  RemoveLockedBagNodes();
  FreeListValues(link);
} /* end sieve_action */

sieve_action_0_e( x2) {
  register int Result = 0;
  Result = x2;
  return (Result);
} /* end sieve_action_0_e */

sieve_action_0(link)
  RT_VALUE *link; {
  register int x2 = GetValue("x2", link);
  register int result = 0;
  result = sieve_action_0_e(x2);
  RunTimeBag = AddRTBagNode(TY_INT, result);
} /* end sieve_action_0 */

gamma_reaction() {
  return (sieve_reaction());
} /* end gamma_reaction */

main ()
{
  RunTimeBag = (RT_BAG *) NULL;
  CreateInitialBag ();
  Gamma ();
  exit (0);
}
```