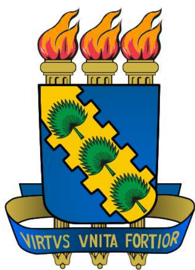
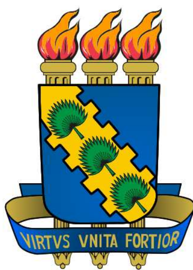


Universidade Federal do Ceará
Departamento de Computação
Mestrado e Doutorado em Ciência da Computação

Fusion: Abstrações Linguísticas sobre Java para Programação Paralela Heterogênea com GPGPUs.

Anderson Boettge Pinheiro

Fortaleza – Ceará
Maio de 2013



Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação
Mestrado e Doutorado em Ciência da Computação

Fusion: Abstrações Linguísticas sobre Java para Programação Paralela Heterogênea com GPGPUs

Autor
Anderson Boettge Pinheiro

Orientador
Prof. Dr. Francisco Heron de Carvalho Junior

Dissertação de mestrado apresentada ao Programa de Pós-graduação em Ciência da Computação da Universidade Federal do Ceará como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Fortaleza – Ceará
Maio de 2013

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca de Ciências e Tecnologia

P718f Pinheiro, Anderson Boethge.
Abstrações linguísticas sobre Java para programação paralela heterogênea sobre GPGPUs. / Anderson Boethge. – 2013.
140f. : il. , color., enc. ; 30 cm.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Departamento de Computação, Programa de Pós Graduação em Ciência da Computação, Fortaleza, 2013.
Área de Concentração: Engenharia de Software.
Orientação: Prof. Dr. Francisco Heron de Carvalho Junior.

1.Programação paralela (Computação). 2. Java (Linguagem de programa de computador). 3. Arquitetura de computador. I. Título.

CDD 005

Agradecimentos

Esta dissertação é fruto de muito trabalho e dedicação e não poderia deixar de deixar aqui meus agradecimentos a todos que estiveram ao meu lado durante esse período, apoiando e compreendendo aqueles momentos em que não estive presente

Ao meu professor e orientador Heron, pela dedicação e permanente apoio. Aos seus ensinamentos, sua forma amiga, exigente e crítica, de fundamental contribuição no meu crescimento enquanto pesquisador.

Aos meus pais Gilnei e Vera, que souberam apoiar uma decisão tão difícil quanto a de um filho sair de casa para buscar seu caminho. A distância tornou tudo mais difícil mas hoje estamos colhendo os frutos daquilo que plantamos a dois anos atrás, e isso graças a vocês que sempre compreenderam minhas decisões, muito obrigado.

A minha namorada, amiga e companheira Francinize, pelo carinho, paciência e compreensão. A gradeço muito por seu incansável apoio, em todos os momentos durante o desenvolvimento desse trabalho.

A toda minha família, por não apenas apoiar a mim mas também aos meus pais que durante esse período tenho certeza tiveram todo apoio de todos vocês, inclusive naqueles momentos mais difíceis em que não pude estar ai com vocês, agradeço profundamente a todos.

Por fim, deixo aqui minha sincera gratidão a todas as pessoas que, direta ou indiretamente, contribuíram para a concretização deste trabalho.

"O saber a gente aprende com os mestres e os livros.
A sabedoria, aprende-se é com a vida e com os humildes."

Cora Coralina

"Ando impressionado com a urgência do fazer.
Saber não é o suficiente: precisamos aplicar.
Estar disposto não é o suficiente: precisamos fazer."

Leonardo da Vinci

Resumo

Unidades de aceleração gráfica, ou GPU (*Graphical Processing Units*), tem se consolidado nos últimos anos para computação de propósito geral, para aceleração de trechos críticos de programas que apresentam requisitos severos de desempenho quanto ao tempo de execução. GPUs constituem um dentre vários tipos de aceleradores computacionais de propósito geral que tem sido incorporados em várias plataformas de computação de alto desempenho, com destaque também para as MIC (*Many Integrated Cores*) e FPGA (*Field Programmable Gateway Arrays*). A despeito da ênfase nas pesquisas de novos algoritmos paralelos capazes de explorar o paralelismo massivo oferecido por dispositivos GPGPU, ainda são incipientes as iniciativas sobre novas abstrações de programação que tornem mais simples a descrição desses algoritmos sobre GPGPUs, sem detrimento à eficiência. Ainda é necessário que o programador possua conhecimento específico sobre as peculiaridades da arquitetura desses dispositivos, assim como técnicas de programação que não são do domínio mesmo de programadores paralelos experientes na atualidade. Nos últimos anos, a NVIDIA, indústria que tem dominado a evolução arquitetural dos dispositivos GPGPU, lançou a arquitetura Kepler, incluindo o suporte às extensões Hyper-Q e Dynamic Parallelism (DP), as quais oferecem novas oportunidades de expressão de padrões de programação paralela sobre esses dispositivos. Esta dissertação tem por objetivo a proposta de novas abstrações de programação paralela sobre uma linguagem orientada a objetos baseada em Java, a fim de expressar computações paralelas heterogêneas do tipo *multicore*/*manycore*, onde o dispositivo GPU é compartilhado por um conjunto de *threads* paralelas que executam no processador hospedeiro, em um nível de abstração mais elevado comparado às alternativas existentes, porém ainda oferecendo ao programador total controle sobre o uso dos recursos do dispositivo. O projeto das abstrações dessa linguagem proposta, doravante chamada Fusion, parte da expressividade oferecida pela arquitetura Kepler.

Abstract

Graphics Processing Units (GPUs) have been consolidated in the recent years for general purpose computing, aimed at accelerating critical sections of programs that exhibit high performance requirements. GPUs constitute one among a set of classes of general-purpose computational accelerators, which have been incorporated in many high performance computing platforms. Other important classes are MICs (Many Integrated Cores) and FPGAs (Field Programmable Gateway Arrays). In spite of the many research works on new parallel algorithms for exploiting the massive parallelism offered by GPU devices, the initiatives on new programming abstractions that aims at simplifying the description of these algorithms on GPUs, without detriment to efficiency, are still incipient. The programmer still needs specific knowledge about the peculiarities of the target GPU architecture, as well as programming techniques that are complex even for parallel programmers with large experience. In the recent years, NVIDIA, one of the main GPU providers, launched the Kepler architecture, including the support for the Hyper-Q and Dynamic Parallelism (DP) extensions, which offer new opportunities for increasing the expressiveness of parallel programming interfaces on describing patterns of parallel computation using these devices. This work aims at proposing new parallel programming abstractions in a object-oriented language based on Java, for expressing heterogeneous *multicore*/*manycore* parallel computations, where the GPU device is shared by a set of parallel threads that runs in the host processor, at a higher level of abstraction compared to the existing alternatives, but still offering to the programmer total control over the device resource usage. The design of the abstractions of the proposed language, so-called Fusion, starts from the expressiveness offered by the Kepler architecture.

Sumário

1 Introdução **1**

1.1 Programação para GPUs 3

1.2 Orientação a Objetos em Programação para GPUs 4

1.3 Computação Heterogênea - Multicore/Manycore 5

1.4 Problema de Pesquisa 6

1.5 Objetivos 8

1.5.1 Objetivo Geral 8

1.5.2 Objetivos Específicos 8

1.6 Metodologia 8

1.7 Organização do Documento 10

2 Computação com Unidades de Processamento Gráfico (GPU) **12**

2.1 Arquitetura GPU 13

2.1.1 Pipeline Gráfico 13

2.1.2 Pipeline Programável 14

2.2 GPGPU ou GPU *Computing* 15

2.3 Evolução 17

2.4 Programação 19

2.5 CUDA 20

2.5.1 Arquitetura 21

2.5.2 Modelo de Programação 21

2.6 Portabilidade 28

2.6.1 Arquitetura Kepler 31

3 Linguagens de Programação para GPGPU **39**

3.1 hiCUDA 39

3.1.1 O Compilador hiCUDA 41

3.1.2 Exemplos de Programa em hiCUDA 43

3.2 CUDA Fortran 45

3.3 JCuda 48

3.3.1 O Compilador JCuda 50

3.4 pyCUDA 51

3.5 Chestnut 52

3.6 JaBEE 54

3.7 Rootbeer . . . 55
3.8 Avaliação de Desempenho . . . 59

4 A Linguagem Fusion 62
4.1 Características e Principais Conceitos . . . 63
4.1.1 Usuários Alvo . . . 63
4.1.2 Paradigma de Programação: Orientação a objetos . . . 64
4.1.3 Transparência sobre Detalhes Arquiteturais da GPU . . . 66
4.1.4 Programação Paralela Heterogênea . . . 67
4.2 Modelo de Programação de Fusion . . . 68
4.2.1 Objeto Acelerador . . . 68
4.2.2 Classe de Objetos Aceleradores . . . 70
4.2.3 Configuração de Grade . . . 74
4.2.4 Unidades e *Streams* . . . 75
4.2.5 Instanciação de Objetos Aceleradores . . . 77
4.2.6 Hierarquias de Memória . . . 79
4.2.7 Comunicação . . . 80
4.2.8 Sincronização . . . 81

5 Implementação e Estudos de Caso 83
5.1 Arquitetura do Compilador . . . 83
5.1.1 Compilador LLVM . . . 84
5.1.2 Projeto VMKit . . . 85
5.1.3 Versão inicial e funções . . . 87
5.2 Primeiro Estudo de Caso: Multiplicação de Matrizes . . . 90
5.3 Segundo Estudo de Caso: Enumeração Completa . . . 98
5.4 Considerações . . . 103

6 Conclusões e Trabalhos Futuros 105
6.1 Cumprimento dos Objetivos . . . 105
6.1.1 Trabalhos Futuros . . . 108

Referências Bibliográficas 115

A Estudo de Caso 1 116
A.1 Código Java . . . 116
A.2 Código CUDA C . . . 119
A.3 Código Fusion . . . 122
A.3.1 Classe MatrixMultiUnits . . . 122
A.3.2 Classe MatrixAccel . . . 125

B Estudo de Caso 2 127
B.1 Código CUDA C . . . 127
B.2 Código Fusion . . . 134
B.2.1 Classe Application . . . 134
B.2.2 Classe TaskEnumaration . . . 137

B.2.3 Classe EnumerationAccel 138

Capítulo 1

Introdução

Nos dias atuais, o grau de interesse pela computação paralela tem se tornado cada vez maior, em função da disseminação e consolidação de processadores de múltiplos núcleos em computadores de propósito geral nas mais diversas categorias, incluindo *smartphones*, *tablets*, *laptops*, *desktops*, servidores e plataformas de computação de alto desempenho (CAD), tais como *clusters* e de processamento paralelo massivo (MPPs - *Massive Parallel Processors*). Além dos tradicionais nichos de aplicação da computação paralela, notadamente em aplicações científicas e de engenharia com requisitos de CAD, novos nichos de aplicação tem buscado na computação paralela a resolução mais rápida de problemas conhecidos e que exigem algum esforço computacional. Tendo em vista esse contexto, onde a computação paralela assume notável grau de importância, muitas frentes de pesquisa tem se dedicado à busca de novas ferramentas e tecnologias que possam auxiliar no desenvolvimento de software com ênfase no paralelismo. Contudo, para exploração eficiente do paralelismo, são necessários, além de um hardware especializado, modelos de programação que consigam explorar os seus recursos adequadamente, de forma a obter o melhor desempenho em função de características de cada aplicação.

Simultaneamente à consolidação dos processadores de múltiplos núcleos (*multicore*), os aceleradores computacionais emergiram na área de CAD, como dispositivos associados aos processadores de uma plataforma computacional para realizar tarefas específicas de alto custo computacional em um tempo de processamento viável, impossível de ser atingido por meio de um processador convencional. Os principais tipos de aceleradores computacionais são FPGAs (*Field-Programmable Gateway Arrays*) [Herbordt et al. 2007], GPUs (*Graphical Processing Units*) [Pharr e Fernando 2005] e MICs (*Many Integrated Cores*) [Duran

e Klemm 2012].

FPGAs consistem de unidades de *hardware* com um arranjo de portas lógicas programáveis, permitindo que os usuários programem seu comportamento de acordo com suas necessidades [Herbordt et al. 2007], sendo de natureza intrinsecamente diferentes de GPUs e MICs.

Uma característica comum aos aceleradores computacionais é o uso massivo do paralelismo. No caso de GPUs e MICs, isso envolve envolve o uso de uma grande quantidade de núcleos simplificados de processamento, o que motiva o uso do termo *manycore* para diferenciá-los da abordagem *multicore* dos processadores convencionais modernos com os quais cooperam nas plataformas de computação onde são empregados.

Cronologicamente, o surgimento de dispositivos GPU antecede o surgimento de dispositivos MIC, sendo sua disseminação e consolidação fortemente motivada pela sua origem como aceleradores gráficos, no amplo mercado de jogos de computador e aplicações de computação gráfica, antes de se tornarem uma alternativa real no mercado de computação de propósito geral, em especial no restrito, embora crescente, mercado de computação de alto desempenho. Por esse motivo, GPUs são ofertadas no mercado em modelos de custo variado, em geral acessível para equipar os computadores de usuários domésticos e de pequenas corporações.

Enquanto isso, MICs foram propostos recentemente no contexto da concorrência industrial como uma alternativa às GPUs, tendo seu marco no lançamento do processador *Intel Xeon Phi* e motivados pelo forte interesse das aplicações em aceleradores computacionais. Em relação às GPUs, MICs buscam oferecer um modelo de programação mais adequado à computação de propósito geral, diminuindo certas barreiras ao uso disseminado de GPUs, relacionadas às dificuldades em lidar com seu modelo de programação, que estão intrinsecamente relacionadas as motivações do trabalho de pesquisa dessa dissertação. No entanto, MICs, como é o caso do *Intel Xeon Phi*, tem sido inicialmente lançados para o restrito mercado de computação de alto desempenho, possuindo custo que se equivale ao de GPUs de topo de linha, também voltadas a esse mercado, tal como a linha Tesla da NVIDIA.

Esse trabalho de pesquisa está particularmente interessado em programação voltada aos dispositivos de aceleração computacional do tipo GPU, tendo em vista a sua maior disseminação e consolidação no mercado ao tempo do início desse projeto, o qual se mantém atualmente.

1.1 Programação para GPUs

As GPUs surgiram no final da década de 1970, como unidades de processamento de propósito especial voltadas à computação gráfica em duas dimensões, com aplicação em jogos de computador e aceleração de programas gráficos de interesse da engenharia. Somente em meados da década de 1990, surgiram os primeiros dispositivos para computação gráfica em três dimensões, porém mantendo sua única finalidade no processamento gráfico. Entretanto, a evolução das GPUs na década de 1990 e início dos anos 2000 permitiu que alguns estágios do chamado pipeline gráfico pudessem ser (re)programados, viabilizando uma maior flexibilidade no uso desses dispositivos que permitiu sua aplicação em computação de propósito geral, dando origem ao termo GPGPU (*General-Purpose Graphics Processing Units*).

Logo, percebendo o potencial da arquitetura GPU para uma vasta área de aplicações, diversas iniciativas de pesquisa começaram o desenvolvimento das primeiras linguagens de programação capazes de fazer uso dos recursos disponibilizados por esses dispositivos, com destaque para a primeira linguagem desenvolvida no âmbito acadêmico, conhecida como BrookGPU [Buck et al. 2004], desenvolvida por pesquisadores da universidade de Utah nos Estados Unidos.

Entretanto, ainda se faz necessário um modelo de programação para GPUs capaz de expressar de forma mais natural as aplicações, ainda assim permitindo explorar as peculiaridades da arquitetura da GPU alvo. A principal solução para programação sobre GPUs proposta em meados dos anos 2000, com relação a portabilidade de programas, foi a arquitetura CUDA (*Computer Unified Device Architecture*) da empresa NVIDIA, tornando essa abordagem mais difundida e utilizada até os dias atuais. A arquitetura CUDA tem por premissa expor o dispositivo GPU como um coprocessador aritmético, expondo os seus processadores conhecidos como multiprocessadores de fluxo (SM, do inglês *Stream Multiprocessor*) para serem utilizados na computação paralela de um determinado conjunto de operações. Entretanto, CUDA ainda é uma linguagem de baixo nível de abstração, exigindo conhecimento especializado por parte do desenvolvedor.

Outras linguagens e ferramentas vem surgindo ao longo dos anos. Em sua grande maioria, incorporam a uma determinada linguagem muito difundida o suporte para ligação à computações descritas na arquitetura CUDA, tais como CUDA Fortran [Fortran 2012] e JCuda [Yan, Grossman e Sarkar 2009], as quais estendem as linguagens Fortran e Java, respectivamente.

Algoritmos e Técnicas de Programação GPGPU Simultaneamente aos esforços nas áreas de linguagens e modelos de programação, devemos ressaltar a grande quantidade de trabalhos de pesquisa, publicados em periódicos científicos de alta reputação, que tem se dedicado a apresentar a implementação de algoritmos conhecidos sobre GPUs, buscando utilizar seus recursos de forma otimizada a fim de minimizar o tempo de execução desses algoritmos a um patamar inalcançável utilizando-se processadores de arquitetura convencional. Embora esse trabalho de pesquisa esteja intrinsecamente relacionado com os esforços na área de linguagens de programação para GPUs, os esforços na área de implementação de algoritmos não são considerados menos importantes, tendo sido utilizados para entender as técnicas de programação comumente usadas sobre essas plataformas.

1.2 Orientação a Objetos em Programação para GPUs

A busca por um nível mais elevado de abstração na programação para dispositivos GPU conduz várias linhas de pesquisas às linguagens orientadas a objetos, cuja importância é atualmente de tal elevado nível para a indústria do software que serve como base para quase todas as técnicas e métodos de sucesso na engenharia de software moderna, o que faz relevante estudar formas de usá-las para programação de computações sobre GPUs.

Linguagens orientadas a objetos tem por premissa a abstração de dados, permitindo a modelagem de um problema real através da abstração de objetos que trocam mensagens através de invocação de seus métodos particulares. Um objeto possui uma propriedade chamada de encapsulamento, na qual abstrai a sua implementação mantendo encapsulados seus atributos e métodos, disponibilizando apenas uma interface para a aplicação ter acesso a suas funções. Ou seja, a aplicação não tem conhecimento sobre a implementação do objeto, mas supõe que ele vai retornar o resultado correto previsto pelos seus métodos. Além disso, o mecanismo de herança, entre classes de objetos que descrevem os atributos e métodos comuns de objetos que representam entidades de mesma natureza, permite a hierarquização de conceitos de forma abstrata e evita a redundância de código.

No entanto, a orientação a objetos é difícil de ser conciliada com arquiteturas GPU, devido as características inerentes a essa arquitetura. Isso justifica o fato de que a maioria dos trabalhos relacionados busquem apenas uma conexão direta com a arquitetura CUDA, pouco preocupada com a proposta de novas abstrações para compatibilizar a orientação a objetos com a programação GPGPU, problema

de interesse nesse trabalho.

1.3 Computação Heterogênea - Multicore/Manycore

Com o advento de GPUs como uma alternativa para acelerar trechos computacionalmente intensivos de programas em aplicações de CAD, as plataformas heterogêneas de computação paralela tornaram-se uma tendência nesse domínio. Por exemplo, o projeto de arquiteturas de *cluster computing* constituídos de centenas de nós de processamento distribuído, onde cada nó é composto por vários processadores *multicore* compartilhando um espaço de memória, cada qual associado a um coprocessador *manycore*, como uma GPU, é uma tendência na lista Top500 [TOP 500 2013], dos 500 mais rápidos computadores paralelos, atualizada semestralmente, já há alguns anos. O mesmo vale para plataformas de arquitetura MPP.

Dentre os computadores listados na lista Top500 de novembro de 2012, 12,4% fazem uso de algum tipo de acelerador computacional, incluindo as GPUs, as quais dominam essa lista respondendo por 10,6% do total (53 máquinas). Porém, em relação ao total de desempenho bruto das máquinas, o impacto das GPUs chega 19,6%, por equiparem máquinas que estão mais ao topo da lista, o que evidencia serem uma tendência na arquitetura de clusters e MPPs. Enquanto isso, já existem 7 máquinas que fazem uso do recém-lançado processador *Intel Xeon Phi*, principal representante da classe MIC.

Em plataformas heterogêneas de computação paralela, o potencial de paralelismo que pode ser explorado pela aplicação encontra-se em vários níveis hierárquicos, que somente podem ser explorados de maneira eficiente com o uso de técnicas e modelos de programação específicos para cada nível, tais como:

- MPI [Dongarra et al. 1996] para troca de mensagens entre processos distribuídos no nós de processamento;
- OpenMP [OpenMP Architecture Review Board 1997] para sincronização entre *threads* sobre os núcleos de processamento de um nó de processamento;
- CUDA, ou OpenCL, para explorar o paralelismo do acelerador computacional, se este for do tipo GPU.

Junte-se a isso a existência de múltiplas hierarquias de memória, que ampliam ainda mais o grau de complexidade da programação eficiente nessas plataformas.

Mesmo em computadores *desktop* ou servidores, vistos individualmente, a computação heterogênea se faz relevante, tendo em vista a proliferação, já amplamente disseminada, da tecnologia de processadores de múltiplos núcleos, que faz do processamento paralelo dentro do processador hospedeiro uma suposição importante que deve ser levada em conta em uma linguagem voltada a conexão entre uma computação nele realizada e a GPU.

Outro fator importante, que tem sido levado em consideração para o uso de GPUs, é a preocupação com o consumo de energia, já que uma característica importante desses dispositivos é o baixo consumo de energia, quando comparadas aos processadores convencionais, alcançando picos de desempenho superiores a um custo energético relativamente menor. As arquiteturas GPU mais recentes possuem preocupação mais enfática em relação aos aspectos de eficiência energética. Um exemplo disso é a arquitetura Kepler [KeplerGK110 2012], da empresa NVIDIA.

Enfim, a computação heterogênea diminui os custos, financeiros e energéticos, para o desenvolvimento de aplicações que exijam um alto poder computacional e que possuam paralelismo massivo, em um contexto onde há limitações na quantidade de núcleos de processamento que podemos incorporar a um processador *multicore*, tanto em relação a gargalos arquiteturais quanto ao custo energético. Mesmo aplicações que não se enquadram nessas características ainda assim podem obter desempenho sobre arquiteturas GPU, desde que bem implementadas [Pilla e Navaux 2010].

Tendo em vista o contexto apresentado nesta seção, esse trabalho de pesquisa está interessado em oferecer contribuições no contexto da computação heterogênea, notadamente na ligação entre processadores *multicore* e coprocessadores *manycore* do tipo GPU, que é de interesse não apenas de aplicações tradicionais de CAD, mas também de quaisquer aplicações que necessitem do uso de GPUs para acelerar sua computação, tendo em vista que quase todos os processadores que equipam os computadores atuais são compostos de múltiplos núcleos.

1.4 Problema de Pesquisa

O avanço, tanto em relação à arquitetura dos dispositivos GPU quanto ao seu modelo de programação, é evidente. No entanto, algo a ser aprimorado é a possibilidade de obter-se um alto nível de abstração na programação sem prejudicar o desempenho das aplicações frente a todo o poder computacional oferecido pelos aceleradores gráficos. No contexto da computação paralela heterogênea, esse problema se torna ainda mais desafiador, tendo em vista a necessidade de conciliar

de forma coerente modelos de programação totalmente distintos.

Neste trabalho em particular, nos restringimos ao caso da ligação entre processadores *multicore* e coprocessadores *manycore* do tipo GPU. Além disso, desejamos usufruir do potencial de linguagens de programação orientadas a objetos, devido a sua capacidade de abstração, compatibilidade das modernas técnicas de engenharia de software, e maior difusão na comunidade.

Com isso, a principal questão a ser respondida por esse trabalho é:

> *Novas abstrações linguísticas podem ser incorporadas a uma linguagem de programação orientada a objetos já difundida a fim de facilitar o uso de um dispositivo GPU, sem gargalos de sincronização, por um programa paralelo que executa sobre um processador multicore ?*

Mesmo com a existência de linguagens de alto poder de abstração, essa ainda é uma questão que gera discussões. Apesar de tornar mais simples a descrição de operações complexas sobre a GPU, conciliar abstrações de alto nível e desempenho não é algo trivial. Em uma interface de programação de alto nível, não são todas as aplicação que conseguem explorar o desempenho potencial da arquitetura GPU. Em geral, são voltadas a facilitar o desenvolvimento de uma certa classe de aplicações, com características específicas.

Ainda que novos projetos desenvolvam ferramentas para facilitar e reduzir a complexidade da programação sobre GPUs, não existe um padrão que garanta a melhor utilização dos recursos, de modo que o programador necessita ter um conhecimento aprofundado sobre os detalhes da arquitetura alvo, bem como sobre as características da aplicação, para obter o melhor desempenho.

Portanto, a principal limitação na utilização de GPUs para propósito geral é a intima relação entre desempenho e as características da arquitetura alvo, impedindo que os modelos computacionais atuais consigam obter o desempenho máximo ofertado por dispositivos GPUs, de especial interesse na área de CAD. Contudo, os avanços tecnológicos tem permitido um nível mais elevado de expressividade dos modelos, removendo limitações que antes impediam alguns padrões de programação paralela serem aplicados nas aplicações de propósito geral. As principais inovações são as tecnologias Hyper-Q e paralelismo dinâmico (DP, do inglês *dynamic Paralellism*), da recente Kepler GK110. A Hyper-Q permite a execução de até 32 fluxos de operações em paralelo algo de importância para CAD, além de, através da tecnologia DP, permitir o lançamento de novos trabalhos diretamente no dispositivo,

evitando comunicações entre hospedeiro e dispositivo que podem comprometer o desempenho das aplicações.

O paralelismo a nível de *threads*, amplamente difundido pela programação paralela em sistemas de computação paralela *multicore*, nos remete a utilização da tecnologia Hyper-Q para conexão de diferentes *threads* de uma aplicação *multithread* com um ou mais dispositivos GPU. De fato, essa e outras extensões tecnológicas são de especial interesse aos objetivos deste trabalho, por tornarem possível a implementação das abstrações propostas como resultado desse trabalho, a fim de responder a sua questão de pesquisa.

1.5 Objetivos

1.5.1 Objetivo Geral

O objetivo geral desse trabalho é o desenvolvimento de abstrações linguísticas sobre uma linguagem de programação orientada a objetos, com o propósito de encapsular a complexidade da ligação com dispositivos GPUs de programas paralelos *multicore*, em um contexto de computação heterogênea envolvendo processadores *multicores* e coprocessadores *manycore*.

1.5.2 Objetivos Específicos

Para alcançar o objetivo geral, identificamos os seguintes objetos específicos para esse trabalho:

i. Caracterizar as principais técnicas de programação utilizadas por programadores das áreas de ciências e engenharias, os principais usuários de plataformas heterogêneas de computação paralela, sobre GPUs;

ii. Identificar e classificar as principais limitações das abordagens sobre linguagens orientadas a objetos de programação para dispositivos GPUs;

iii. Caracterizar o potencial de expressividade para descrição de padrões de computação paralela de novas tecnologias suportadas por novas arquiteturas de dispositivos, tais como Hyper/Q e Paralelismo Dinâmico.

1.6 Metodologia

A proposta de novas linguagens de programação, suportando novas abstrações algumas das quais também poderiam ser incorporadas em linguagens já existentes, exige um conhecimento mais aprofundado sobre como os programadores tem feito

uso dos recursos de programação paralela sobre GPUs para o usufruto das aplicações que desenvolvem. Para tal propósito, é necessário ao pesquisador debruçar-se sobre a vasta literatura que tem sido produzida, desde a segunda metade da década de 2000, a respeito do desenvolvimento de programas usando aceleradores gráficos, especialmente no contexto de aplicações de interesse científico. Por esse motivo, esse trabalho de pesquisa iniciou com um estudo exploratório sobre artigos onde são apresentadas implementações de algoritmos importantes nas áreas de ciências e engenharias, com alta demanda computacional. Para garantir que a abrangência desse estudo seja relevante, foi adotada a taxonomia *Dwarf Mine* [Asanovic et al. 2006], a qual organiza algoritmos de interesse de aplicações científicas de acordo com características relevantes como padrões de comunicação entre processos e de acesso à memória. Entretanto, o resultado desse estudo exploratório não é apresentado nessa dissertação, apesar da sua relevância para construção dos seus resultados.

Iniciou-se então uma busca por ferramentas e linguagens de programação para GPGPU, visando analisar suas principais características e limitações com relação a programação sobre plataformas paralelas heterogêneas. Esse estudo permite a identificação das estratégias de mapeamento e construção das estruturas para os dispositivos GPUs, adotadas por cada um dos trabalhos relacionados. Verificou-se, com isso, as dificuldades em aliar um alto nível de abstração com alto desempenho sobre arquiteturas GPGPU.

Após a etapa exploratória sobre o contexto no qual está inserido esse trabalho, obteve-se o conhecimento necessário para iniciar o desenvolvimento desse trabalho. Seguindo o principal objetivo, era necessário escolher uma linguagem de alto nível que pudesse de alguma forma ser estendida com novas abstrações capazes de realizar flexibilizar o desenvolvimento de aplicações sobre duas arquiteturas distintas, encontradas nas plataformas paralelas heterogêneas. A linguagem escolhida foi a linguagem Java, devido ao seu caráter orientado a objetos e o fato de ser uma linguagem já consolidada e amplamente difundida.

As abstrações pretendidas tem como principal preocupação explorar ao máximo os benefícios da orientação a objetos. Tendo como base uma aplicação *multithread*, as abstrações devem estabelecer uma ponte entre a aplicação e um dispositivo GPU na forma de objetos aceleradores. Abstrações baseadas na orientação a objetos podem ser favoráveis aos usuários, permitindo que dediquem mais tempo para a modelagem da aplicação em si, abstraindo-se de conceitos específicos da arquitetura do hardware da GPU alvo.

Com isso, esse trabalho tem como resultado o protótipo de linguagem de programação que estende a linguagem Java, a qual chamou-se de Fusion. A linguagem Fusion permite a modelagem das aplicações baseadas na orientação a objetos através de abstrações capazes de realizar a ligação aos dispositivos GPUs.

As abstrações propostas para linguagem estão intimamente relacionadas a um tipo específico de objeto, chamado de *objeto acelerador*, responsável por realizar a "ponte" entre o processador hospedeiro (*multicore*) e dispositivo GPU (*manycore*). Um objeto acelerador tem por premissa expressar o poder computacional dos dispositivos GPU e ao mesmo tempo esconder sua complexidade, mantendo encapsulado no objeto todo o paralelismo de partes críticas da aplicação. Nesse contexto, a linguagem Fusion propõe o conceito original de paralelismo entre *kernels*, na forma de *kernels* paralelos sob a abstração de *unidades* de objetos aceleradores.

Assim, cada *thread* é responsável pela instanciação de uma unidade do objeto acelerador, que por sua vez abre um canal de comunicação específico entre processador hospedeiro e dispositivo GPU, pelo qual os métodos do objeto poderão ser lançados para execução sobre a arquitetura GPU. Essa é a principal contribuição da linguagem Fusion, ou seja, permitir que uma aplicação Java *multithread* executando sobre um processador hospedeiro *multicore* possa se comunicar com um dispositivo GPU.

A conexão entre processador hospedeiro e dispositivo GPU é projetada de forma a ser mantida transparente para a aplicação, ou seja, a linguagem Fusion permite que objetos comuns dessa aplicação possam ser substituídos por objetos aceleradores sem muitas mudanças no código, e vice-versa, sem comprometer a estrutura da aplicação. Logicamente que essa substituição só pode acontecer entre objetos equivalentes, que possuam a mesma interface de comunicação e realizem a mesma função.

Finalmente, são desenvolvidos estudos de caso para validação do modelo proposto. Duas aplicações modeladas nas linguagens envolvidas nesse projeto (Java, CUDA C e Fusion) são usadas para demonstrar a utilização das abstrações e a estruturação das aplicações sobre a linguagem Fusion.

1.7 Organização do Documento

Seguido desse capítulo introdutório, essa dissertação possui outros cinco capítulos, descritos a seguir.

O Capítulo 2 apresenta o contexto em que está inserido a área da pesquisa desse trabalho, relatando a evolução e o desenvolvimento dos aceleradores GPUs.

Além disso, será apresentada brevemente a arquitetura CUDA e detalhes sobre a arquitetura Kepler GK110, de especial interesse nesse trabalho.

O Capítulo 3 aborda os principais trabalhos relacionados sobre projetos de interfaces e linguagens de programação para GPGPU, apresentando algumas especificidades de cada um. O capítulo tem por objetivo, além de apresentar as ferramentas atuais, identificar suas limitações e quais as soluções adotadas para abordar a computação GPGPU.

O Capítulo 4 apresenta a linguagem Fusion, com foco nas abstrações desenvolvidas para ela. Características do modelo de programação adotado também poderão ser analisadas, desde sua sintaxe até a maneira como a linguagem lida com a conexão entre processador hospedeiro e dispositivo GPU.

O Capítulo 5 apresenta uma proposta inicial para prototipação do compilador Fusion. Ainda nesse capítulo, são apresentados dois estudos de caso para validação parcial do modelo proposto, bem como uma melhor compreensão do uso de objetos aceleradores pelas aplicações.

Finalmente, o Capítulo 6 discute os objetivos alcançados nessa dissertação e as principais dificuldades encontradas durante a sua realização. Serão também apresentados alguns tópicos identificados para futuros trabalhos.

Capítulo 2

Computação com Unidades de Processamento Gráfico (GPU)

As unidades de processamento gráfica (GPU) surgiram no final da década de 70 com o propósito de aumentar a eficiência e o poder de processamento gráfico em duas dimensões. Os primeiros aceleradores para três dimensões surgiram em 1995. Até o início dos anos 2000, os aceleradores possuíam apenas essa finalidade [NVIDIA 2013].

A partir de meados dos anos 2000, tornou-se significativo e crescente o interesse no uso de GPUs para computação de propósito geral. Sua evolução permitiu a integração de múltiplas unidades de processamento simples, obtendo um desempenho superior ao alcançado pelas arquiteturas convencionais. Posteriormente, foram inseridas novas unidades programáveis, chamados sombreadores (do inglês, *shaders*), as quais possibilitaram a realização de outras funções além de cálculos gráficos. A primeira GPU programável surgiu em 2001, e passaram a ser conhecidos também como processadores gráficos [NVIDIA 2013].

Em 2006, foi introduzida a abstração de programação conhecida como *sombreadores de geométria* (do inglês, *geometry shaders*), que permite ao programador criar seus próprios métodos de processamento [Nguyen 2007] [Blythe 2006]. Isso tornou possível de fato a utilização de aceleradores gráficos para a computação de propósito geral, possibilitando o surgimento das primeiras linguagens específicas para GPGPU (*General-Purpose Graphical Processing Units*). Além desses fatores de natureza técnica, associa-se também o custo/benefício associado a utilização de uma placa gráfica em aplicações onde o desempenho é o ponto

crítico como uma das principais motivações para o interesse por esses dispositivos computacionais.

2.1 Arquitetura GPU

O modelo no qual é baseada a arquitetura das placas GPU modernas é conhecido como SIMD (*Single Instruction Multiple Data*) ou *streaming data-paralell arithmetic architecture*. Sua utilização limita-se a execução de uma única linha de instruções em partes disjuntas de uma estrutura de dados distribuída entre os núcleos de processamento. Nesse modelo, cada unidade de processamento executa a mesma sequência de operações sobre um fluxo de dados independente em relação às outras unidades.

A programação sobre GPUs só foi possível a partir da quarta geração das placas gráficas, através da utilização de APIs (*Application Programming Interface*) especializadas, diretamente sobre a placa de vídeo. No entanto, sua utilização ainda era um desafio para os programadores devido a sua complexidade, exigindo grande conhecimento das APIs e da arquitetura alvo. Além disso, os sombreadores mostraram-se pouco apropriados para a computação de propósito geral. A principal mudança nessa geração foi no *pipeline* gráfico, o qual define os estágios pelos quais os dados são processados pela GPU.

Atualmente, existem linguagens específicas para programação sobre aceleradores gráficos. Por exemplo, a linguagem CUDA C é a mais conhecida, de propriedade da empresa NVIDIA. Essas linguagens permitem a programação de alguns componentes dos dispositivos aceleradores, expondo a GPU como um processador aritmético. Nos primórdios, componentes executavam apenas operações pré-configuradas através das APIs.

Aplicações que fazem uso eficiente de placas GPU possuem grande demanda por recursos computacionais, relacionada a realização de processamentos sobre grandes estruturas de dados que podem ser facilmente realizados em paralelo sobre partições disjuntas dessas estruturas.

2.1.1 Pipeline Gráfico

Um pipeline é uma técnica utilizada com o intuito de acelerar a velocidade de execução de instruções, aumentando também a quantidade de dados processados em paralelo. Os dados são paralelizados em estágios independentes, aumentando a produtividade com a execução de instruções no mesmo instante. Os estágios associados a um pipeline gráfico são:

- **Geração de vértices** - nesta primeira etapa, a aplicação repassa para o *pipeline* um conjunto de vértices que é transformado em um fluxo de dados (*stream*) que permite que sejam processados individualmente;
- **Geração de Primitivas** - o conjunto de vértices é então transformado em uma forma geométrica, onde um conjunto de dois vértices formam um linha, três vértices formam um triângulo, e assim por diante;
- **Geração de Fragmentos** - também conhecida com rasterização, onde determina-se quais *pixels* serão cobertos por cada primitiva, delineando um conjunto de fragmentos;
- **Processamento de Fragmentos** - etapa com maior custo computacional, pois compreende a coloração dos fragmentos juntamente com as informações cromáticas e sobre os materiais de superfície utilizados;
- **Composição** - etapa final, onde a imagem final é construída.

As APIs originais, OpenGL (*Open Graphics Library*) e DirectX3D, são conhecidas por apresentarem um *pipeline* fixo, no qual seu funcionamento é pré-definido, omitindo do programador a sua implementação.

2.1.2 Pipeline Programável

Com o aumento na busca por efeitos gráficos mais complexos, foram necessárias mudanças que tornaram possível ao desenvolvedor da implementação modificar o comportamento de algumas etapas do *pipeline*, uma vez que APIs tradicionais como as citadas anteriormente não davam suporte a efeitos muito complexos, limitando o programador.

Em busca de flexibilizar o desenvolvimento em algumas etapas, permitiu-se que seu comportamento fosse alterado através de programas executados na GPU, conhecidos como *sombreadores* (do inglês, *shaders*). O conjunto de instruções desses programas permite definir operações a serem executadas sobre as etapas do pipeline gráfico. Os três tipos de sombreadores são:

- *Sombreador de Vértice*: permite a inserção de efeitos 3D aos objetos de uma cena, através da manipulação de cor, posição e textura;
- *Sombreador de Geometria*: recebe dados do sombreador de vértice, como primitivas, e responsabiliza-se pela renderização da cena;

- *Sombreador de Pixel*: responsável por efeitos de luz e cor dos *pixels*.

2.2 GPGPU ou GPU *Computing*

O modelo de programação seguido pelas unidades programáveis da GPU é baseado no SPMD (*Single Program Multiple Dada*). Nesse modelo, a GPU processa muitos elementos em paralelo utilizando o mesmo programa. Para isso, os elementos necessariamente devem ser independentes, entre os quais não há comunicação. Ou seja, um programa implementado para uma GPU deve possuir muitos elementos paralelos e um único algoritmo para processá-los. Esse modelo é mais adequado para códigos ditos de linha reta, com muitos elementos sendo processados em etapas exatamente sobre o mesmo código, apropriado para arquiteturas SIMD.

Em programação de propósito geral, um programa utiliza a GPU da mesma maneira que um programa gráfico. Porém, seu mapeamento para a GPU pode ser complexo. Apesar de as operações aritméticas serem as mesmas, e podem facilmente ser seguidas, a terminologia é diferente. Pharr e Fernando 2005 demonstra como esse mapeamento pode ser realizado, mostrando como a terminologia adotada em computação gráfica pode ser adaptada para a computação de propósito geral.

Para a implementação de um programa com ênfase na computação gráfica, um desenvolvedor deve concentrar-se nos seguintes aspectos, relacionados ao *pipeline* gráfico descrito anteriormente:

i. O programador define uma figura geométrica, que será fragmentada na etapa de geração de fragmentos, cobrindo cada pixel;

ii. Cada fragmento será tratado pelo sombreador definido pelo programador;

iii. O código calcula o valor do fragmento através de operações matemáticas, enquanto a memória global obtém dados a partir de uma memória de texturas;

iv. A imagem resultante poderá então ser utilizada como textura em novas passagens pelo *pipeline*.

Nos primórdios, a utilização da GPU deveria seguir as etapas do *pipeline* gráfico para a finalidade de computação de propósito geral. Como mostrado, seguimos o exemplo citado por Owens et al. 2008, onde é realizada uma simulação de fluidos:

i. O programador especifica uma primitiva geométrica que cobrirá um domínio de interesse. Na etapa de geração de fragmentos, será gerado um fragmento para cada pixel coberto por essa primitiva;

ii. Na etapa seguinte, um sombreador específico cobrirá cada fragmento, ou seja, cada ponto executará o mesmo programa (SPMD);

iii. Assim como na computação gráfica, o valor atual de cada ponto será calculado através de um conjunto de operações matemáticas, aplicando-se uma operação *gather* para reunir os valores atuais dos pontos vizinhos que são necessários no cálculo;

iv. O resultado será armazenado na memória global e poderá ser utilizado como entrada para outra passagem no pipeline.

A principal dificuldade encontrava-se na necessidade de utilizar APIs gráficas para a implementação de propósito geral. Com isso, o desenvolvimento do programa deve seguir exatamente as etapas do *pipeline* gráfico, o que não é bem visto pelos programadores que preferem trabalhar diretamente sobre as unidade programáveis. As APIs buscaram minimizar essa dificuldade permitindo o acesso direto às unidades, deixando o desenvolvimento mais natural sem a necessidade de interfaces gráficas, estruturando o desenvolvimento da seguinte maneira:

i. O programador define diretamente seu domínio de interesse e estrutura sua grade de *threads*;

ii. Um sombreador calcula o valor para cada *thread*, implicitamente sem que o programador se preocupe com essa operação;

iii. O programa realiza cálculos computacionais, utilizando uma operação *gather* para ler os valores da memória global e uma operação *scatter* para escrever. Nesse caso, o mesmo *buffer* pode ser utilizado tanto para leitura quanto escrita, favorecendo algoritmos mais flexíveis com menor uso de memória;

iv. O resultado final poderá ser utilizado para novas computações.

Esse modelo permite ao programador especificar o paralelismo de dados diretamente no programa, além de, devido ao acesso direto às unidades programáveis, poder utilizar uma linguagem mais familiar, facilitando a implementação e depuração de seus programas.

2.3 Evolução

Os principais desafios no desenvolvimento de CPUs encontram-se no consumo de energia e acesso a memória. Atualmente, o desenvolvimento está centrado na adição de novos núcleos, porém ainda limitando-se a uma quantidade que varia entre 4 e 8 núcleos, representando a classe dos processadores de múltiplos núcleos (do inglês, *multi-core processors*). Novas arquiteturas surgem a cada dia e novas promessas e expectativas são geradas, como a ideia da fusão de GPU e CPU em um único chip. Por sua vez, GPUs possuem centenas ou milhares de núcleos, especializados no processamento paralelo e massivo, sendo enquadradas na classe dos processadores de muitos núcleos (do inglês , *many-core processors*).

Resumindo as principais características de cada arquitetura, claramente existem diferenças fundamentais oriundas das suas premissas de projeto distintas. Por exemplo, o objetivo de uma CPU concentra-se na execução de uma única *thread*, composta por instruções sequencias, o mais rápido possível, enquanto que uma GPU busca a execução paralela de milhares de *threads*. Para realizar todo controle de operações e instruções de maneira eficiente, a CPU utiliza seus transistores para executar uma sequência de tarefas, enquanto a GPU é especializada em executar milhares de instruções paralelas. De fato, seus transistores estão dedicados a dar maior vazão ao processamento dos dados e não para controle de fluxo de execução.

Com relação ao emprego de memórias *cache*, a técnica que é fundamental para aumentar a vazão de dados entre memória principal e processador em CPUs tradicionais, a GPU possui uma unidade pequena associada a vários controladores de memória independentes, obtendo uma maior largura de banda, tornando possível a execução de milhares de *threads* em paralelo. Diferente das CPUs, o acesso a essas unidades é previsível e pode parte dela ser controlado pelo programador, tornando seu uso mais eficiente e de acordo com o padrão de acesso a memória particular do programa. Com isso, uma GPU é capaz de executar centenas de *threads* por multiprocessador, enquanto que uma CPU executa de 2 a 4 *threads* por núcleo.

Na Tabela 2.1, é possível comparar algumas características de CPUs e GPUs, comparando um processador Intel Xeon E7 e uma placa de vídeo NVIDIA Tesla.

	Intel Xeon E7	NVIDIA Tesla K20X
Poder Computacional	371 GFlops	2.06 TFlops
#Cores	8	2688
Largura de Banda	51.2 GB/s	250 GB/s
Preço	$\geq$ $ 3.600,00	$\geq$ $ 3.200,00

Tabela 2.1: Comparação entre Xeon E7 e Tesla K20X - Fevereiro 2013

Para analisar a evolução das GPUs, a partir da introdução da tecnologia 3D (era pré-GPU), tomamos como base as placas desenvolvidas pela empresa NVIDIA. Na Tabela 2.2, podemos notar o avanço tecnológico. O número de núcleos de GPUs dobra a cada 18 meses, assim como o número de transistores.

Data	Dispositivo	Transistores	Núcleos	Tecnologia
1997	RIVA 128	3 milhões	-	acelerador 3D
1999	GeForce256	25 milhões	-	primeira GPU
2001	GeForce 3	60 milhões	-	primeiro sombreador programável
2002	GeForce FX	125 milhões	-	precisão dupla
2004	GeForce 6800	222 milhões	-	GPGPU
2006	GeForce 8800	681 milhões	128	CUDA
2008	GeForce GTX 280	1.4 bilhões	240	CUDA e OpenCL
2009	FERMI	3 bilhões	512	64-bit end., caching
2011	Kepler GK104	3.5 bilhões	1536	Fabricação 28 nm, SMX
2012	Kepler GK110	7.2 bilhões	2880	Dynamic Parallelism, Hyper Q

Tabela 2.2: Evolução das placas da NVIDIA [NVIDIA 2013]

Nesta dissertação, tomamos por base as placas da empresa NVIDIA, a qual tem notoriamente ditado a evolução tecnológica das GPUs em conjunto com alguns de seus parceiros. Consequentemente, adotamos as terminologias definidas por essa empresa. Durante a evolução, alguns conceitos sobre o hardware foram tomando forma. Os processadores de dados, chamados de processadores de fluxo (SP)[1] passaram a ser reunidos em multiprocessadores chamados de multiprocessadores de

[1] do inglês, *stream processor*.

fluxo (SM)[2]. Os SMs possuem dezenas de SPs que podem ser acessados diretamente através de linguagens de programação específicas, tais como CUDA, nas versões iniciais para GPGPU, e, posteriormente, OpenCL. Cada SM possui registradores específicos e unidades totalmente dedicadas para operações aritméticas.

A arquitetura Fermi [Fermi 2012] foi a primeira arquitetura para GPUs voltada às necessidades de computação de alto desempenho, representando a quarta geração de placas programáveis da NVIDIA. Trouxe várias inovações que permitiam obter um desempenho muito superior em relação às placas gráficas anteriores a sua geração. A arquitetura Fermi define um endereçamento de 64 bits e introduz um novo nível de memória cache, tornando o processo de leitura e acesso aos dados mais rápido.

A arquitetura Kepler é a mais atual arquitetura de GPUs lançada pela NVIDIA, [KeplerGK110 2012], também voltada para computação de alto desempenho, porém com uma maior preocupação com o consumo de energia. A arquitetura Kepler caracteriza-se pela construção baseada em uma tecnologia de transistores de 28 nm de tamanho, contra os 40 nm de sua predecessora Fermi, justificando seu baixo consumo de energia, já que quanto menor o tamanho dos transistores menor é dissipação de calor. Além disso, introduz uma remodelagem nos SM que passaram a ser chamados de SMX. Mais detalhes dessa nova arquitetura serão apresentados na Seção 2.6.1.

2.4 Programação

Como visto na seção anterior, o desenvolvimento de aplicações de propósito geral para GPU exigia um grande conhecimento das APIs por parte do programador, ou seja, poucos desenvolvedores eram capacitados para explorar ao máximo o desempenho de uma GPU para suas aplicações. As operações necessariamente eram executadas através de funções fixas, pré-definidas pela API.

Com o objetivo de flexibilizar o desenvolvimento, novas APIs surgiram. Por exemplo, DirectX 9 permitiu um alto nível de sombreamento e a modificação de algumas de suas funções. Porém, só podia ser programada através de uma linguagem conhecida como HLSL, desenvolvida pela Microsoft. Porém, logo surgiram linguagens mais flexíveis e acessíveis, como a GLSL (*OpenGL Shading Language*) para OpenGL, e a CG (*C for Graphics*) da NVIDIA, que também permitia a implementação para múltiplos alvos, tais como DirectX e OpenGL. Apesar do surgimento dessas novas linguagens, ainda eram muito restritas ao uso

[2]do inglês, *streaming multiprocessor*

das APIs, tornando seu uso uma tarefa árdua, comprometendo a produtividade dos programadores, que deveriam ser especializados nesse tipo de linguagem.

Com isso, muitos pesquisadores voltaram sua atenção para o desenvolvimento de linguagens de mais alto nível, que abstraíssem a GPU como um processador de *streams*. Um programa que segue o modelo de *streams* deve expressar o paralelismo, a comunicação e a transferência de dados utilizando os recursos da GPU. Esse programa deve possuir um conjunto de instruções, um conjunto ordenado de dados e um *kernel*. No *kernel*, o programador define a grade computacional e as funções que serão aplicadas a cada elemento dos dados de entrada, através de uma ou mais *threads* topologicamente organizados segundo a grade computacional.

A primeira linguagem, oriunda do meio acadêmico, foi a BrookGPU [Buck et al. 2004], que abstrai os conceitos gráficos, enfatizando a representação de dados como *streams* e a computação como um *kernel*. Os *kernels* são escritos com um subconjunto restrito de instruções da linguagem C. Em geral, não são utilizados ponteiros. O *kernel* é mapeado para o dispositivo como um código de fragmento de sombreamento e os dados como texturas. Entradas e saídas são realizadas explicitamente com operações de leitura e escrita.

Grandes empresas, percebendo o interesse da comunidade científica nas GPUs, não somente como hardware gráfico, mas também como acelerador para diferentes aplicações, começaram o desenvolvimento de ferramentas para expor a GPU como um processador de propósito geral. As linguagens mais utilizadas atualmente para programação sobre GPUs são desenvolvidas pelas empresas NVIDIA e Khronos Group [Kronos 2012]. A NVIDIA lançou a linguagem conhecida como CUDA C, enquanto a antiga ATI iniciou o desenvolvimento da linguagem Brook++, baseada na linguagem BrookGPU. Atualmente, a AMD (*Advanced Micro Devices*), proprietária da antiga ATI, propõe uma SDK *(Software Development Kit)* para a linguagem OpenCL (*Open Computing Language*), desenvolvida pelo grupo Khronos.

2.5 CUDA

Como visto anteriormente, a arquitetura CUDA tem como principal objetivo encapsular as APIs gráficas e expor a GPU como um processador aritmético, Criado em 2007 pela NVIDIA, é uma extensão da linguagem C, permitindo o desenvolvimento de programas que executam, em parte, nas GPUs. Esses trechos de código são conhecidos como *kernels*.

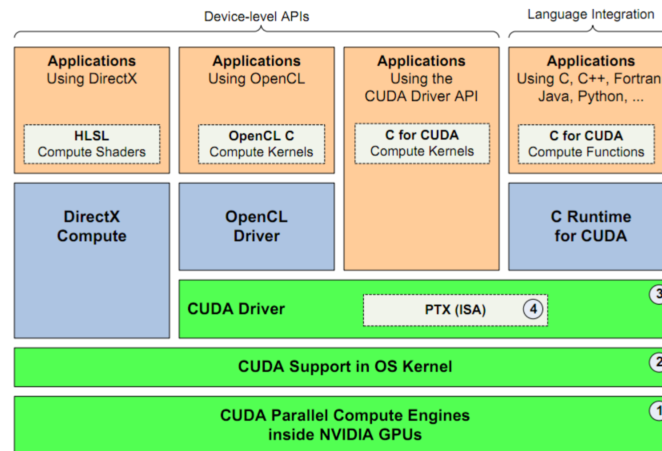


Figura 2.1: Arquitetura Cuda [CUDA1.1 2012]

2.5.1 Arquitetura

A partir da geração G80 das placas GPU NVIDIA [NVIDIA 2013], a empresa começou o desenvolvimento da arquitetura CUDA, que considera hardware e software, composta pelos seguintes componentes:

1. Dispositivo Físico;
2. Suporte para inicialização de hardware;
3. Driver para desenvolvedores;
4. Interfaces de programação em nível de dispositivo, tais como DirectX ou OpenCL, ou mais alto nível, tais como *C Runtime for CUDA*.

Além disso, CUDA fornece integração com outras linguagens, tais como JAVA, Python, Fortran, dentre outros. O ambiente ainda disponibiliza exemplos, bibliotecas padrão otimizadas (BLAS e FFT), compilador (nvcc) e depurador (cudagdb). Na Figura 2.1, é possível observar a arquitetura CUDA mais detalhadamente.

2.5.2 Modelo de Programação

A API CUDA visualiza o dispositivo GPU como um co-processador para a CPU hospedeira, sendo responsável pela execução paralela de blocos básicos de código computacionalmente intensivos e com funções bem caracterizadas através de *threads*, enquanto que a CPU responsabiliza-se pelo controle do fluxo de execução. Dessa forma, o fluxo de execução parte do hospedeiro, inicializando um conjunto de dados

(vetor) que serão copiados para memória global do dispositivo. Por sua vez, o dispositivo realiza os cálculos sobre o conjunto de dados. Ao final da execução, os dados são copiados de volta para o hospedeiro.

O código a ser executado no dispositivo deve ser escrito em um trecho de código específico chamado *kernel*. O *kernel* não pode conter chamadas a outras funções fora do seu escopo, e não permite recursão. Um *kernel* é denotado em uma função especifica identificada pelo marcador *__global__*:

```
__global__ myKernel (arguments){
....
}
```

A chamada para esse trecho de código pode ocorrer em qualquer parte do código fonte seguindo a seguinte notação:

```
myKernel <<dimGrid, dimBlock>> (arguments);
```

As *threads* são agrupadas em blocos e mapeadas para os SMs. Em geral, existem mais *threads* do que um multiprocessador pode executar em paralelo. O número de blocos também pode ser maior que o número de SMs, não sendo necessário recompilação para execução em arquiteturas com diferentes capacidades. Os blocos são organizados em grades e mapeados aos SPs.

O controle de execução pode ser realizado através da utilização de barreiras de sincronização. Como o controle do fluxo é executado sobre as *threads* em um bloco e não sobre os dados, o modelo de programação da arquitetura CUDA (Figura 2.2) é denominado SIMT (*Single Instruction Multiple Thread*) [CUDA4.1 2012]. Cada SP executa suas *threads* intercalando suas instruções, técnica conhecida como IMT (*Interleaved Multithreading*) [Munshi et al. 2005]. A escalabilidade da arquitetura ocorre principalmente pela adição de novos SMs.

A unidade responsável pelo controle de uma linha de instruções de um conjunto de *threads* em um bloco é conhecida como *warp*. Um *warp* pode controlar até 32 *threads*, sendo executadas 4 *threads* em pipeline. Na arquitetura Kepler, cada multiprocessador possui quatro unidades *warp*, diferenciando-se da arquitetura Fermi que possuía apenas duas unidades desse tipo.

Hierarquia de Memória

Um dispositivo possui diferentes níveis de memória (Figura 2.3), os quais seguem a hierarquia de *threads* e dos processadores. Cada SP possui sua própria memória

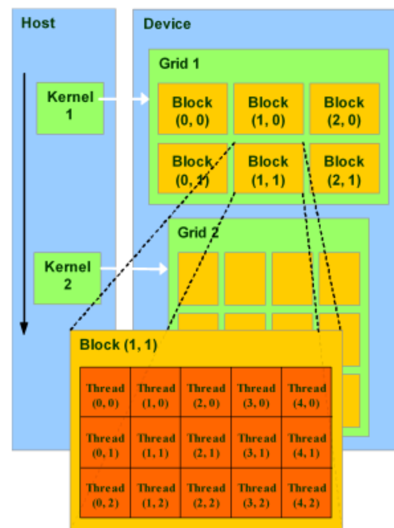


Figura 2.2: Modelo de programação [CUDA4.2 2012]

individual, representada por registradores de 32 bits extremamente rápidos, que podem ser utilizados de maneira concorrente pelas *threads* que executam sobre o SP. Seguindo a hierarquia, cada bloco de *threads* em um SM possui acesso à memória compartilhada que é acessada por todas as *threads* que o compõem. As *caches* são memórias somente para leitura, compartilhadas entre os SPs em um mesmo SM. O principal objetivo dessas memórias é reduzir o número de acesso às memórias externas ao SM, possuindo tempo de acesso baixo mas com tamanho limitado.

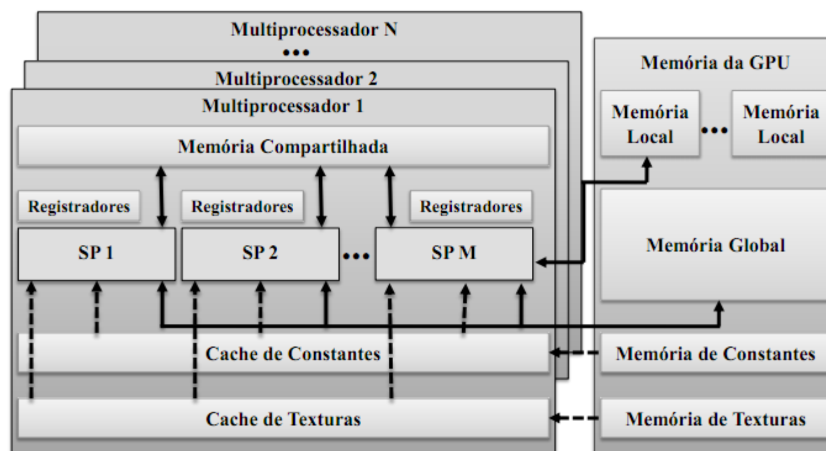


Figura 2.3: Arquitetura da memória GPU

As memórias global, local, de constantes e de texturas são externas ao SM, sendo maiores e com uma latência mais alta. A memória global é compartilhada por todas as *threads* em todos SMs, enquanto que a memória local é individual de cada *thread*,

ambas para leitura e escrita. As memórias de constantes e texturas são memórias otimizadas e acessíveis por todas as *threads*. As caches contidas internamente aos SMs são de acesso a essas memórias. Na Tabela 2.3, é apresentado um resumo das características dos espaços de memória em diferentes níveis de hierarquia.

Memória	Localização - SM	Tamanho	Latência (ciclos)	Escopo	Acesso
Registradores	interna	até 64 KB por SM	≈ 0	thread	leitura/ escrita
Compartilhada	interna	16 KB por SM	$\geqq 4$	bloco	leitura/ escrita
Cache de texturas	interna	depende da global	0-600	bloco	leitura
Cache de constantes	interna	64 KB	0-600	bloco	leitura
Global	externa	até 1024 MB	400-600	grade	leitura/ escrita
Local	externa	depende da global	400-600	thread	leitura/ escrita
Texturas	externa	1.4 bilhões		grade	leitura
Constantes	externa	3 bilhões		grade	leitura

Tabela 2.3: Resumo características da memória GPU

Programação

A linguagem CUDA C estende a linguagem C através de primitivas com o objetivo de permitir o acesso aos recursos da GPU escondendo a sua complexidade.

Os qualificadores de função são primitivas para identificar o local de execução de uma determinada função. São eles:

- `__global__`: define um *kernel*;
- `__device__`: função chamada somente no dispositivo e executada somente sobre o dispositivo;
- `__host__`: define uma função chamada somente no hospedeiro e executada somente sobre o hospedeiro;

Os qualificadores de variáveis são utilizados para sinalizar em qual nível da hierarquia de memória determinada variável será alocada:

- `__device__`: define uma variável a ser alocada na memória global do dispositivo GPU, tornando-se acessível por todas as *threads*;
- `__constant__`: especifica uma variável alocada na memória de constantes;
- `__shared__`: define uma variável alocada no espaço compartilhado, sendo acessível apenas pelas *threads* do mesmo bloco.

As primitivas `__global__` e `__device__` introduzem algumas restrições às funções:

i. não suportam recursão;

ii. não podem conter variáveis declaradas como estáticas; e

iii. não podem conter um número variável de argumentos.

Não é possível recuperar o endereço de uma função declarada com o modificador `__device__`. Por sua vez, funções declaradas com o modificador `__global__` são assíncronas e devem especificar a dimensão da grade e dos blocos. As dimensões são especificadas através de variáveis do tipo `dim3`, que definem vetores de inteiros. Os mecanismos utilizados para manipulação e definição da grade são:

- $gridDim$: contêm a dimensão da grade;
- $blockIdx$: contêm o índice do bloco dentro da grade;
- $blockDim$: contêm a dimensão do bloco;
- $threadIdx$: contêm o índice da *thread* dentro do bloco;
- $warpSize$: contêm o número de *threads* no *warp*.

Toda escrita em uma variável do tipo `__shared__` só poderá ser vista por todas as *threads* após a passagem pela barreira de sincronização, realizada através da primitiva `__syncthreads()`.

No Código 2.1, é possível observar a utilização das primitivas, e como é realizada a transferência de dados entre a CPU hospedeira e o dispositivo GPU. No Código 2.2, um exemplo de definição do *kernel* para ser executado na GPU é apresentado.

Código 2.1: Multiplicação de Matrizes em CUDA: Hospedeiro

```
...
int main (int argc, char *argv[]){
    int *A, *B, *C,
        *A_d, *B_d, *C_d;

    cudaMalloc ((void **)&A_d, total_bytes);
    cudaMalloc ((void **)&B_d, total_bytes);
    cudaMalloc ((void **)&C_d, total_bytes);

    cudaMemcpy(A_d, A, total_bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, total_bytes, cudaMemcpyHostToDevice);

    dim3 gride (LARGURA/SUB_LARGURA, LARGURA/SUB_LARGURA);
    dim3 bloco (SUB_LARGURA, SUB_LARGURA);

    matMulGPU <<<gride, bloco>>> (A_d, B_d, C_d);

    cudaMemcpy(C, C_d, total_bytes, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);

    return 0;
}
```

Nas linhas 3 e 4, do código 2.1, são definidos os dados da aplicação. Na linha 3, são definidas as matrizes armazenadas na memória do hospedeiro. Na linha 4, essas matrizes são armazenadas na memória do dispositivo. Toda variável a ser utilizada no dispositivo deve ser reservada na memória global antes de iniciar sua computação. Essa operação ocorre no bloco de código nas linhas 6-8, através do comando `cudaMalloc((void**) &`*A_d*`,` *total_bytes*`)`, onde `&`*A_d* corresponde ao endereço inicial na memória que vai possuir o tamanho passado como argumento em *total_bytes*.

Os dados são copiados para a memória do dispositivo nas linhas 10 e 11, através do comando `cudaMemcpy(`*A_d*`,` *A*`,` *total_bytes*`, cudaMemcpyHostToDevice)`. Esse comando possui quatro argumentos na seguinte ordem: (*i*) variável de destino, (*ii*) variável de origem, (*iii*) quantidade de dados e (*iv*) tipo de operação de transferência. As operações possíveis para transferência são:

- `cudaMemcpyHostToDevice`: cópia do hospedeiro para o dispositivo;
- `cudaMemcpyDeviceToHost`: cópia do dispositivo para o hospedeiro e;
- `cudaMemcpyDeviceToDevice`: cópia no próprio dispositivo.

É necessário definir a grade computacional que será utilizada sobre o dispositivo, indicando a configuração dos blocos e o número de *threads* em cada bloco. Essas configurações são definidas, como visto, em variáveis internas. Nesse exemplo, são atribuídas nas linhas 13 e 14, para a grade e para os blocos respectivamente.

A chamada do *kernel* ocorre na linha 16, onde observa-se que nesse momento o programador informa a grade computacional que será utilizada por esse *kernel* especificamente através da sintaxe de configuração $<<< grade, bloco >>>$.

Os dados são copiados de volta para o hospedeiro ao final da execução do *kernel* na linha 18 e a memória do dispositivo é reciclada nas linhas 20-22.

Código 2.2: Multiplicação de Matrizes em CUDA: Dispositivo

```
1  __global__ void matMulGPU (int *A, int *B, int *C) {
2      __shared__ int Mds [SUB_LARGURA][SUB_LARGURA];
3      __shared__ int Nds [SUB_LARGURA][SUB_LARGURA];
4
5      int bx = blockIdx.x;
6      int by = blockIdx.y;
7      int tx = threadIdx.x;
8      int ty = threadIdx.y;
9
10     int Row = by * SUB_LARGURA + ty;
11     int Col = bx * SUB_LARGURA + tx;
12
13     float Pvalue = 0;
14     for (int m = 0; m < LARGURA/SUB_LARGURA; ++m){
15         Mds[ty][tx] = A[Row*LARGURA + (m*SUB_LARGURA + tx)];
16         Nds[ty][tx] = B[Col + (m*SUB_LARGURA + ty)*LARGURA];
17
18         __synchthreads();
19
20         for (int k = 0; k < SUB_LARGURA; ++k)
21             Pvalue += Mds[ty][k] * Nds[k][tx];
22
23         __synchthreads();
24     }
25     C[Row*LARGURA + Col] = Pvalue;
26 }
```

O *kernel matMulGPU* é definido como uma função global na linha 1 do Código 2.2. Nas linhas 2 e 3, são definidas variáveis armazenadas na memória compartilhada do dispositivo que serão utilizadas no cálculo da multiplicação de matrizes. Os índices para localização das *threads* são alcançados no bloco de código das linhas 5-8. Logo, são definidas a linha e a coluna específicas da matriz para cada *thread*, nas linhas 10 e 11.

O paralelismo é definido através de dois laços aninhados nas linhas 14 à 24.

No primeiro laço, um bloco de dados é atualizado na memória compartilhada a cada iteração, facilitando o acesso pelo conjunto de *threads*. Essa operação deve ser sincronizada entre todas as *threads* do bloco, o que ocorre na linha 18 através do comando `__synchthreads()`.

O laço mais interno, nas linhas 20 e 21, tem por função realizar o cálculo de cada posição da matriz resultante, onde cada *thread* é responsável por uma linha/coluna. Antes que os dados sejam atualizados novamente, é necessário que todas as *threads* estejam sincronizadas, o que é realizado na linha 23.

Ao final da execução, cada *thread* transfere o valor calculado na variável *Pvalue* para posição que ficou responsável da matriz resultante C.

2.6 Portabilidade

Tratando-se de aplicações de computação de alto desempenho, um dos principais pontos a serem analisados é a portabilidade, que é a capacidade que um código escrito deve possuir de obter desempenho aceitável em dispositivos diferentes, ou seja, que não seja necessário reescrever o código toda vez que o usuário tenha necessidade de utilizar um dispositivo diferente.

A portabilidade da arquitetura CUDA se dá somente entre dispositivos NVIDIA e, dependendo das primitivas utilizadas, com o mesmo *compute capability*. Essa portabilidade é garantida através do seu compilador NVCC (*NVIDIA Compiler Collection*), baseado no LLVM (*Low-Level Virtual Machine*) [Illinois 2012], como ilustrado na Figura 2.4. O compilador invoca diversas ferramentas para diferentes estágios de compilação. Determina-se qual modelo deverá ser utilizado de acordo com as informações coletadas sobre o modelo da GPU alvo no momento da inicialização da aplicação.

Inicialmente, o compilador realiza o tratamento dos marcadores introduzidos pela linguagem CUDA C/C++, a fim de separar os códigos específicos de cada arquitetura, **C** puro, para o host, e **cubin**, para o dispositivo. Um preprocessador do grupo EDG (*Edson Design Group*) separa esse códigos em arquivos diferentes. O código C é então compilado por um compilador nativo enquanto o código cubin é compilado pelo LLVM. O compilador LLVM consiste de um *front-end* e um *back-end*. O *front-end* traduz o código-fonte para código LLVM IR (LLVM *bytecode*). Por sua vez, o *back-end* traduz esse *bytecode* em código *assembly*, chamado de PTX (*Parallel Thread eXecution*). Em uma segunda etapa, ocorre a geração de um código específico para o modelo da GPU a partir do código PTX. Na Figura 2.5 é possível analisar

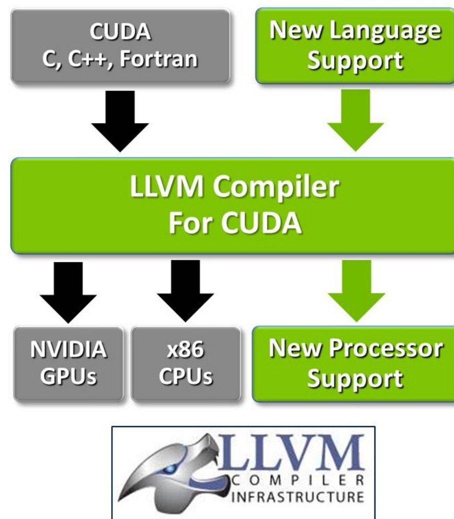


Figura 2.4: Compilador CUDA [CUDA 2013]

o código PTX gerado a partir de um código CUDA, exemplo retirado de Nickolls e Kirk [Patterson e Hennessy 2008] que calcula um produto vetorial.

```
__global__ void saxpy_parallel (int n, float alpha,
floatx, floaty) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i<n) y[i] = alpha x[i] + y[i];
}

int nblocks = ( n + 255) / 256;
saxpy_parallel <<<nblocks , 256>>>(n, 2.0, x, y) ;
```

(a) Código CUDA

```
.entry saxpy ( .param .s32 n, .param .f32 alpha,
.param .u64 x , .param .u64 y ) {
    .reg .u16 %rh <4 >; .reg .u32 %r <6 >; .reg .u64 %rd <8 >;
    .reg .f32 %f <6 >; .reg .pred %p<3 >;
$LBB1__Z9saxpy_GPUifPfS_:
    mov .u16          %rh1, %ctaid.x;
    mov .u16          %rh2 , %ntid.x;
    mul .wide .u16    %r1, %rh1, %rh2;
    cvt .u32 .u16     %r2, %tid.x;
    add .u32          %r3, %r2, %r1;
    ld .param .s32    %r4, [n];
    setp .le .s32     %p1, %r4, %r3;
    @%p1 bra $Lt_0_770;
    cvt .u64 .s32     %rd1, %r3;
    mul .lo .u64      %rd2, %rd1, 4;
    ld .param .u64    %rd3, [y];
    add .u64          %rd4, %rd3, %rd2;
    ld .global .f32   %f1, [%rd4 + 0];
    ld .param .u64    %rd5, [x];
    add .u64          %rd6, %rd5, %rd2;
    ld .global .f32   %f2, [%rd6 + 0 ];
    ld .param .f32    %f3, [alpha];
    mad .f32          %f4, %f2, %f3, %f1;
    st .global .f32   [%rd4 + 0], %f4;
$Lt_0_770 :
    exit;
}
```

(b) Código PTX

Figura 2.5: Exemplo de código PTX gerado pelo compilador CUDA.

Uma alternativa para garantir portabilidade entre dispositivos em geral, válida tanto para dispositivos NVIDIA quanto para dispositivos AMD, seria a utilização da linguagem OpenCL, que também utiliza o compilador LLVM. Isso explica as

mudanças recentes da NVIDIA em seu compilador, permitindo também a inclusão de novas arquiteturas. Atualmente, ambas as empresas desenvolvem ambientes de desenvolvimento (SDK) para as linguagens, as quais constituem alternativas para facilitar o desenvolvimento. CUDA diferencia-se de OpenCL por ser uma arquitetura mais madura, em desenvolvimento há mais tempo, apresentando um desempenho maior devido a maior portabilidade da linguagem OpenCL. Porém, não dispõe de alternativas de compiladores de código aberto, sendo uma arquitetura fechada de propriedade da NVIDIA, a qual reserva-se o direito de fazer uso de conjunto de instruções não documentadas publicamente.

A portabilidade de OpenCL é garantida através do SDK fornecido pelas empresas. Durante a inicialização, o compilador realiza chamadas a diversas rotinas, cujas funções são listadas na Figura 2.6, responsáveis por identificar o sistema operacional e o dispositivo alvo a fim de compilar o código de acordo com o ambiente de execução.

Figura 2.6: Rotinas básicas OpenCL

No caso de dispositivos AMD, a linguagem OpenCL utiliza o compilador *ATI OpenCL Compiler*, o qual possui as mesmas funções do NVCC. Porém, a tradução intermediária não é para PTX e sim para um código chamado *ATI Compute Abstraction Layer* (CAL). Na Figura 2.7, é possível observar as etapas de compilação de ambas as linguagens. Na Tabela 2.4, um resumo das características analisadas nesta seção.

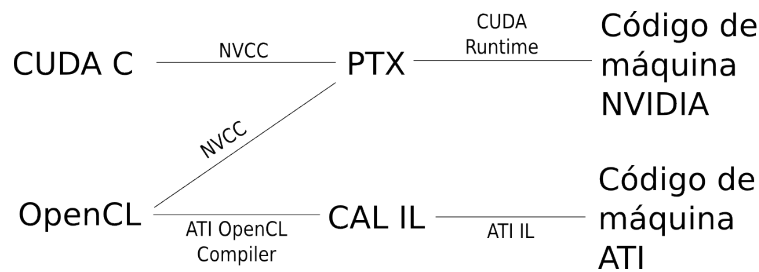


Figura 2.7: Etapas de compilação

CUDA	PTX	OpenCL/CAL
Alto-nível baseado em C	Baixo-Nível	Alto-nível baseado em C
Somente dispositivos NVIDIA	Somente dispositivos NVIDIA	Objetivo independente (NVIDIA, ATI,...)
Utiliza o NVCC para mapeamento para PTX e código de máquina NVIDIA GPU	NVCC é utilizado	Utiliza o ATI OpenCL para mapeamento para CAL IL e código de máquina ATI GPU e o NVCC para PTX e código NVIDIA
Transformações de alto-nível para baixo-nível	Transformações de baixo-nível para alto-nível	Transformações de alto-nível para baixo-nível
Geração de código complexa entre LLVM e CUDA	Geração de código rápida entre LLVM e PTX devido a similaridade	Geração de código complexa entre LLVM e OpenCL

Tabela 2.4: Resumo características na compilação das linguagens

2.6.1 Arquitetura Kepler

A arquitetura Kepler, desenvolvida pela empresa NVIDIA e recentemente lançada, é apresentada na Figura 2.8. Atualmente é suportada parcialmente pelos dispositivos NVIDIA GeForce (GTX série 600 e GTX Titan) e Tesla (K10 e K20), esses últimos voltados a computação de alto desempenho. Suas principais caraterísticas físicas são:

- 15 Streaming Multiprocessors (SMX);

- Memória global de até 8 Gb;
- 3072 núcleos;
- Comunicação CPU-GPU via PCI-e 3 (*Peripheral Component Interconnect-Express*) ;
- Largura de banda: 320 GBytes/s.

Figura 2.8: Diagrama de blocos arquitetura Kepler [KeplerGK110 2012]

Além das características citadas, a arquitetura introduz novas tecnologias que podem ser utilizadas para otimização das aplicações. Essa arquitetura tem por objetivo conciliar desempenho e consumo de energia, diferente da arquitetura Fermi [Nickolls e Dally 2010], a qual busca alto desempenho a qualquer custo. Visto que o baixo consumo tornou-se uma preocupação importante, especialmente com o uso de GPUs como aceleradores computacionais associados aos nós de processamento de clusters, a arquitetura Kepler é promissora, pois a arquitetura consegue maior desempenho devido a tecnologia de 28 nm, permitindo a inserção de mais núcleos de processamento e um consumo menor de energia, além das novas tecnologias que a diferencia da sua predecessora.

A arquitetura Kepler surgiu de uma remodelagem da arquitetura Fermi. Sua primeira versão, suportada pelas GPU de tecnologia GK104, apresenta sua principal diferença com relação a sua predecessora Fermi no fato de que os SMs

foram redesenhados e passaram a se chamar SMX, possuindo agora 4 unidades para gerenciamento de *threads* (*warp*) e passando de 32 para 192 núcleos de processamento, chegando a um total de 1536 núcleos. Na Figura 2.9, podemos observar uma comparação entre um SM da arquitetura Fermi e um SMX da arquitetura Kepler.

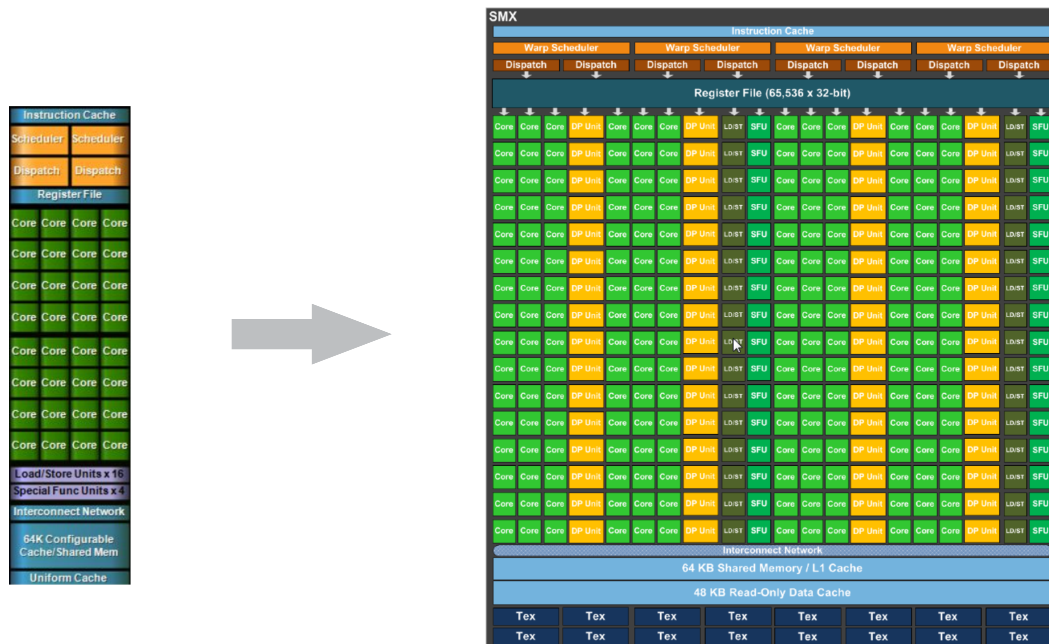


Figura 2.9: *Diferença entre Fermi e Kepler - SMX* [KeplerGK110 2012]

Em 2012, a NVIDIA apresentou as GPUs de arquitetura Kepler com tecnologia GK110 [KeplerGK110 2012], com extensões que permitem uma maior flexibilidade no desenvolvimento das aplicações. Suas principais inovações são o *Dynamic Paralellism*, *Hiper-Q*, *Grid-Management Unit* e *GPU-Direct*, permitindo obter mais desempenho das aplicações sobre as GPUs. As soluções só poderão ser utilizadas na arquitetura CUDA na versão 5, que também está em desenvolvimento, porém já disponibilizada para desenvolvedores em uma versão beta [CUDAZone 2012].

Na Tabela 2.5, a seguir, é possível fazer uma comparação quantitativa entre as arquiteturas Fermi e Kepler.

Paralelismo Dinâmico

O Paralelismo Dinâmico (*Dynamic Paralellism*(DP)) é uma tecnologia que permite realizar a chamada de novos trabalhos diretamente na GPU, sem interferência da CPU, realizando a sincronização dos resultados, gerenciamento e o agendamento das cargas de trabalho. Permite ao programador expor melhor

Tabela 2.5: Comparação entre Fermi e Kepler [KeplerGK110 2012]

	Fermi GF104	Kepler GK104	Kepler GK110
Compute Capability	2.1	3.0	3.5
Threads / Warp	32	32	32
Max Warps / Multiprocessor	48	64	64
Max Threads / Multiprocessor	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	16	16
32-bit Registers / Multiprocessor	32768	65536	65536
Max Registers / Thread	63	63	255
Max Threads / Thread Block	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K	16K	16K
	48K	32K	32K
		48K	48K
Max X Grid Dimension	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	Yes
Dynamic Parallelism	No	No	Yes

o paralelismo e os recursos disponíveis. Uma grande porção da aplicação poderá ser projetada para rodar inteiramente na GPU, liberando a CPU para outras tarefas, reduzindo a transferência de dados entre CPU e GPU, consequentemente aumentando o desempenho das aplicações que podem ser modeladas mais facilmente. Na Figura 2.10, pode-se observar melhor seu funcionamento e uma comparação com relação a arquitetura Fermi.

Essa tecnologia permite que estruturas antes impossíveis de serem tratadas em um único nível de paralelismo possam ser representadas de maneira mais transparente. Com isso, a modelagem das aplicações torna-se mais simples e com possibilidade de otimizações. Na arquitetura Fermi, isso não era possível pelo fato de todos os *kernels* serem lançados no mesmo "instante de tempo" como mostra a Figura 2.10, ou seja, apresentava apenas um nível de paralelismo.

Com mais níveis de paralelismo, essa nova tecnologia introduz a possibilidade de hierarquização dos algoritmos, permitindo que um *kernel pai*, partindo de cálculos pré-definidos, realize a divisão do próximo nível da hierarquia. Na arquitetura Fermi, o programador necessariamente realizava tudo através da CPU, exigindo muitas trocas de dados através da conexão PCI.

Essa flexibilidade pode ser considerada uma das principais mudanças e de maior

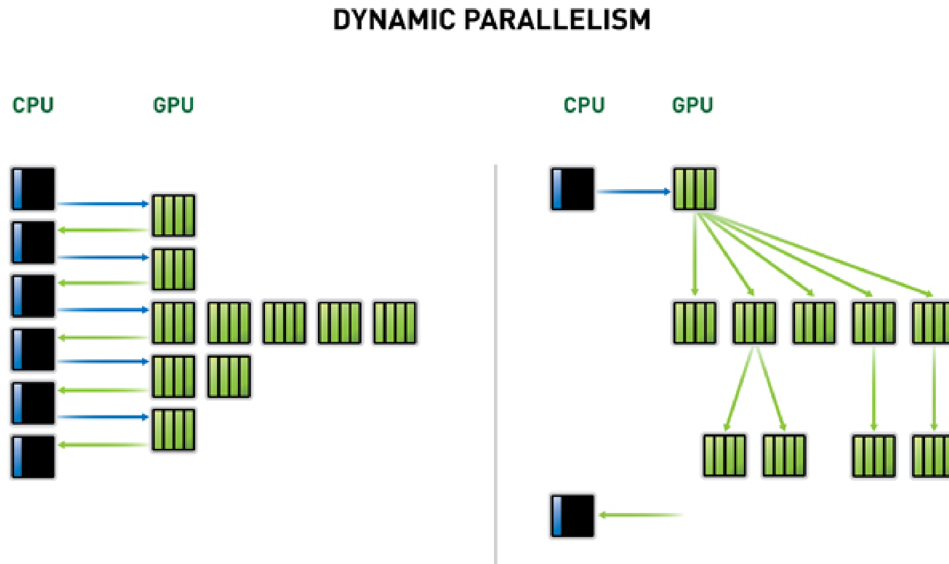


Figura 2.10: *Dynamic Paralellism - Fermi vs Kepler* [KeplerGK110 2012]

importância para programação de propósito geral sobre GPUs. O programador poderá definir seus próprios padrões de execução com suas dependências recursivas, significando uma mudança de paradigma, uma vez que, antes era possível apenas a execução de um único fluxo caracterizando um modelo SIMD. Com tais mudanças, o modelo passa a permitir mais de um fluxo diretamente sobre a GPU, característica de um modelo MIMD (Multiple Instruction Multiple Data). De fato, o modelo MIMD era alcançado através da distribuição de *threads* nos SMs disponíveis, mas não permitia esse nível de paralelismo elevado que propõe a DP, possibilitando uma generalização para um modelo MPMD (Multiple Program Multiple Data).

O paralelismo dinâmico permite novas abordagens, motivando o desenvolvimento de ferramentas que facilitem a programação. Na Figura 2.11, podemos observar como definir os *kernels* que serão executados. O *kernel* $ChildKernel$ será chamado recursivamente pelo *kernel* $ParentKernel$.

```
__global__ ChildKernel(void* data){
    //Operate on data
}

__global__ ParentKernel(void *data){
    ChildKernel<<<16, 1>>>(data);
}

// In Host Code
ParentKernel<<<256, 64>>(data);
```

Figura 2.11: *Dynamic Paralellism* em Cuda: Definição dos *kernels* [NVIDIA 2012]

Hyper-Q

Essa extensão permite que múltiplos cores CPU enviem tarefas para uma única GPU, reduzindo o tempo ocioso do dispositivo. A arquitetura Fermi permitia o lançamento de 16 *kernels* concorrentes simultaneamente, porém, todos na mesma fila de trabalho gerando falsas dependências entre os *streams* (fluxo de execução de um *kernel*). Com o *Hyper-Q*, é possível realizar a conexão de 32 *streams* independentes, sem necessidade de mudanças no código, pois a tecnologia fica encarregada de remover todas as falsas dependências. Na Figura 2.12, é possível observar uma comparação entre as arquiteturas Fermi e Kepler.

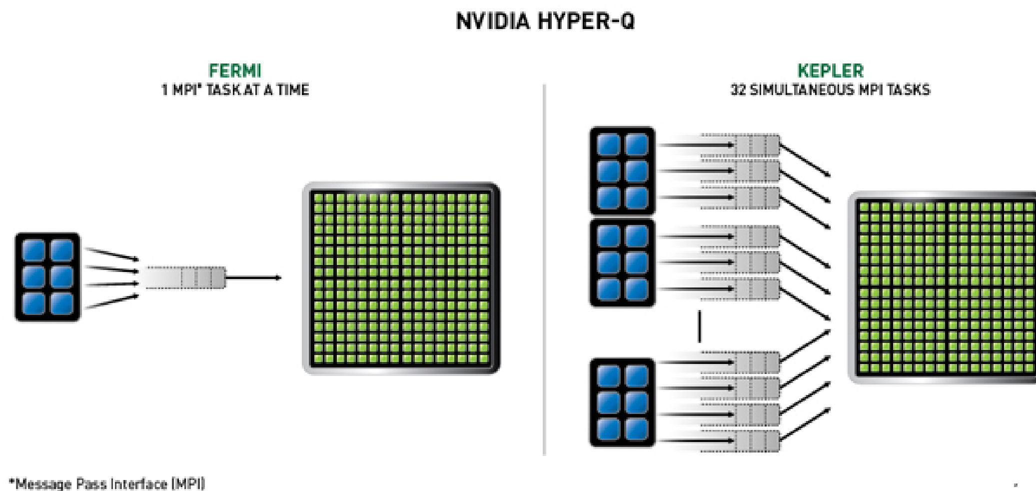


Figura 2.12: *Hiper-Q - Fermi vs Kepler* [KeplerGK110 2012]

Múltiplos fluxos CUDA podem ser mantidos através de conexões separadas, através de processos MPI (*Message Passing Interface*) ou *threads* dentro de um mesmo processo. A Figura 2.13 demonstra a execução de 3 *streams*.

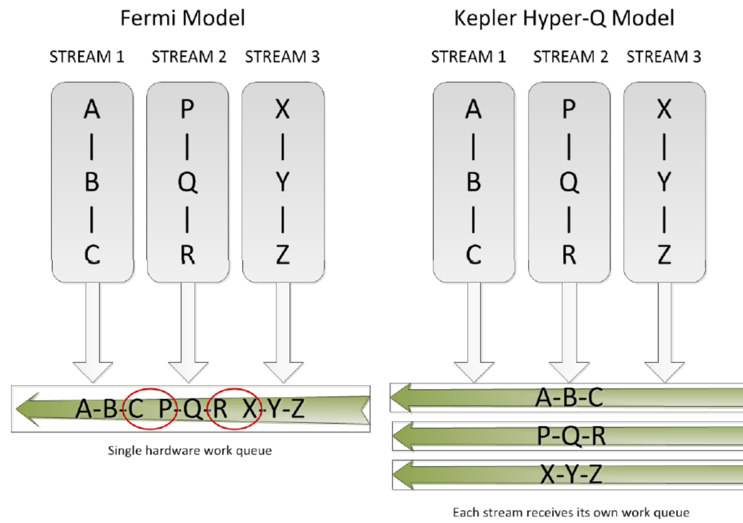


Figura 2.13: *Hiper-Q* fluxo CUDA [KeplerGK110 2012]

No caso da arquitetura Fermi, havia uma serialização entre os três *streams* devido a única fila de execução. Na arquitetura Kepler, cada *stream* é lançado em uma fila independente promovendo a execução paralela das *streams*.

Grid-Mangement Unit

O *Grid-Management Unit* (GMU) ou unidade de gestão de *grids*, é responsável pelo gerenciamento dos *kernels* enviados para execução, ferramenta indispensável para otimizar a utilização do paralelismo dinâmico. A GMU permite pausar o envio de novos *grids* e realiza o controle da fila de execução dos *grids* suspensos até estarem prontos para executar. A Figura 2.14 demonstra seu funcionamento.

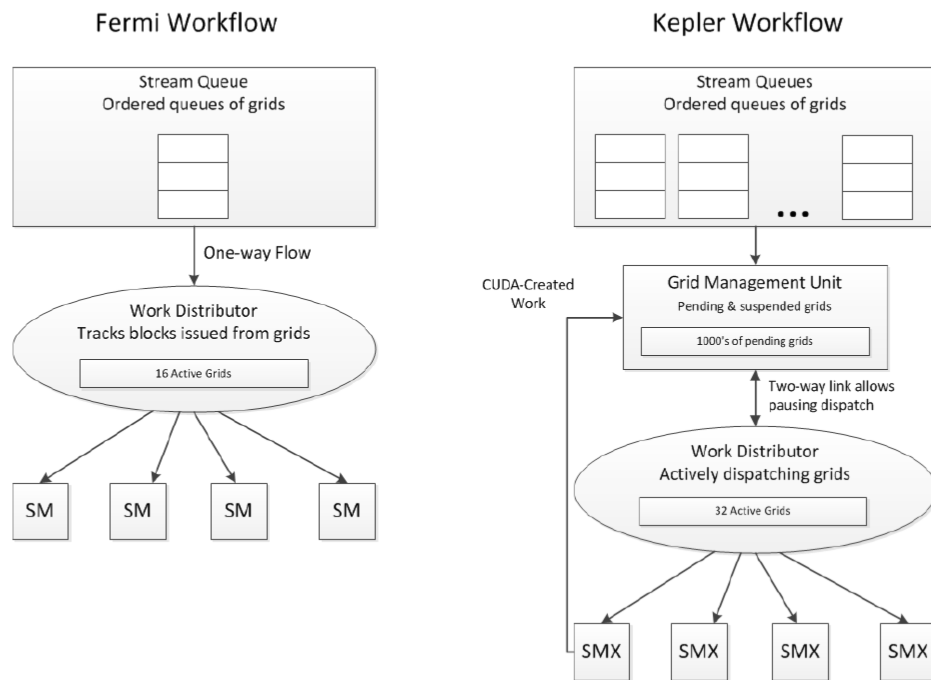


Figura 2.14: *Grid-Management Unit* [KeplerGK110 2012]

Capítulo 3

Linguagens de Programação para GPGPU

Este capítulo tem por objetivo trazer uma análise de alguns trabalhos relacionados ao projeto proposto para esta dissertação. Atualmente existem várias pesquisas que visam oferecer maior nível de abstrações para programação em GPUs. Esse estudo busca identificar algumas das principais ferramentas em desenvolvimento e analisar quais suas principais características, levando em consideração processos de compilação e níveis de abstração.

As ferramentas estudadas incluem APIs para linguagens como C++, Java e Python e outras abordagens como linguagens próprias desenvolvidas especificamente para o desenvolvimento sobre GPU, baseadas na linguagem CUDA C/C++.

3.1 hiCUDA

A linguagem hiCUDA (*high-level CUDA*) [Han e Abdelrahman 2011] é baseada em diretivas de compilação (*pragma*) das linguagens C e C++, provendo abstrações para programação sobre GPU. A linguagem disponibiliza um compilador que converte o código escrito em hiCUDA em um código escrito em CUDA C.

Toda diretiva hiCUDA começa com a notação `#pragma hicuda`, havendo quatro tipos: `kernel`, `loop_partition`, `singular`, e `barrier`.

A diretiva `barrier` tem por objetivo inserir uma barreira de sincronização entre todas as *threads* de um bloco onde foi inserida. A região do *kernel* é identificada entre duas diretivas `kernel`, como mostra o exemplo a seguir:

```
#pragma hicuda kernel kernel-name thread-block-clause thread-clause [nowait]
sequential-code
#pragma hicuda kernel end
```

Todo código escrito dentro desse contexto será traduzido para um *kernel* e executado sobre a GPU. As cláusulas `thread-block` e `thread` representam o número de blocos no *grid* e o número de *threads* por bloco, respectivamente. A cláusula `nowait` indica que a execução dos comandos após a chamado do *kernel* poderão ser executados logo após seu lançamento.

hiCUDA explora o paralelismo através da primitiva `loop_partition`, através da qual é indicado o que cada *thread* irá executar. A sintaxe da primitiva pode ser observada a seguir:

```
#pragma hicuda loop_partition [over_tblock [(distr-type)]] [over_thread]
for loop
```

No uso da primitiva `loop_partition`, é importante destacar os tipos de distribuição possíveis, representada na clausula `over_tblock [(distr-type)]`. Nesse caso, hiCUDA permite dois tipos de paralelismo: por bloco ($BLOCK$) ou ciclico ($CYCLIC$). No primeiro caso, um bloco de *threads* será responsável pela execução de um conjunto de instruções, podendo ocorrer de duas ou mais *threads* ficarem responsáveis por uma iteração do laço. No último caso, cada *thread* é responsável pela execução de uma iteração.

A diretiva `singular` permite que um bloco de código seja executado por uma única *thread*. O código dentro desse bloco é atômico, não podendo ser intercalado com as instruções de outras *threads*:

```
#pragma hicuda singular
sequential-kernel-code
#pragma hicuda singular end.
```

O modelo de dados fornece outras 4 diretivas: global, constant, texture e shared. É através dessas diretivas que o programador pode especificar o uso da hierarquia de memória. A diretiva global pode ser utilizada de três maneiras:

```
1- #pragma hicuda global alloc variable [copyin [variable] | clear]
2- #pragma hicuda global copyout variable
3- #pragma hicuda global free var-sym+ variable:=var-sym [start-idx:end-idx ]?
```

A primitiva `alloc` é responsável por alocar a variável *variable* na memória indicada. As primitivas `copyin` e `copyout` são utilizadas para realizar a transferência

de dados entre a CPU hospedeira e o dispositivo GPU. Os dados são copiados através de `copyin` diretamente para a variável que está sendo alocada, ou copiados de volta para o hospedeiro com a primitiva `copyout`. Nesse caso, o endereço de retorno é abstraído e o compilador deverá fornecer o endereço no hospedeiro da variável que está sendo retornada. A primitiva `free` é responsável por indicar uma região de memória a ser liberada.

Os níveis de memória constantes e texturas são manipulados através das diretivas `constant` e `texture` que possuem duas formas de uso:

```
1- #pragma hicuda constant(texture) copyin variable+
2- #pragma hicuda constant(texture) remove var−sym+
```

Os dados podem ser reservados nas memória de constantes e texturas através da primitiva `copyin` *variable+*. Uma vez que as memórias são somente de leitura, não faz sentido o uso de uma primitiva `copyout`. Nesse caso, é necessário apenas a limpeza desses espaços através da primitiva `remove` $\{var{-}sym\}+$.

Um dos principais pontos para obter desempenho em uma aplicação é o uso da memória compartilhada. Para tal, hiCUDA disponibiliza também a diretiva *shared*, que pode ser utilizada da seguinte maneira:

```
1- #pragma hicuda shared alloc variable [copyin[(nobndcheck)] [variable] ]
2- #pragma hicuda shared copyout [(nobndcheck)] variable
3- #pragma hicuda shared remove var−sym+
```

As diretivas são simples e fáceis de usar, permitindo ao programador explorar todo o paralelismo que as GPUs dispõem sem dificuldades. O compilador é responsável por traduzir todas as primitivas para expressões correspondentes na linguagem CUDA C. Em termos de desempenho, os resultados apresentados na Seção 3.8 demonstram que a tradução não causa perdas de desempenho significativas, levando em consideração a facilidade de uso.

3.1.1 O Compilador hiCUDA

O compilador hiCUDA é de código aberto e está disponível para desenvolvimento colaborativo [Han e Abdelrahman 2012], permitindo atualizações e sugestões enviadas pela comunidade. O compilador possui as seguintes fases:

i. Identificação do kernel - identifica as regiões marcadas pelas diretivas *kernel* e *end_kernel*;

ii. Gerenciador de dados GPU - o compilador gera o código para gerenciar o ciclo de vida dos dados na memória da GPU;

iii. Analise de diretivas - verifica o alcance das diretivas *global* e *constant* dentro de todas as regiões de *kernel*;

iv. Analise do acesso a dados - nessa fase, determina o acesso aos dados de cada região de *kernel*, verificando se todos os dados copiados para memória da GPU cobrem os acessos realizados pelas diretivas identificadas na fase anterior.

v. Geração do *kernel* CUDA C - cada região *kernel* é construída dentro de uma função *kernel* separada. O kernel é construído com as informações coletadas até o momento e os dados informados nas clausulas *tblock* e *thread* da diretiva *kernel*.

vi. Redirecionamento de acesso a *kernel* - todos acessos realizados em uma função *kernel* devem ser redirecionados para suas respectivas regiões na memória da GPU.

vii. Particionamento dos laços - cada região marcada pela diretiva *loop_partition* deverá ser mapeada para a função *kernel*, incluindo os limites e etapas de cada iteração associada a cada *thread* GPU.

viii. Memória compartilhada - a transformação da diretiva *shared* é mais complicada, sendo necessário unir a seção compartilhada em todas as iterações de laços aninhados executados por um mesmo bloco de *threads*, garantindo que todos os acessos simultâneos ocorram sem falhas de dados.

ix. Otimização na alocação de memória na GPU - o compilador utiliza uma técnica de grafos coloridos para construir um registro de alocação, dessa forma é possível reutilizar espaços de memória de variáveis que não se sobrepõe em tempo de execução. Esse é um padrão de alocação de memória em tempo de compilação que pode prover otimizações importantes.

x. Suporte a estruturas dinâmicas - o compilador verifica o uso da diretiva *shape* utilizada para construção de matrizes dinâmicos.

Na Figura 3.1, observa-se o fluxo de compilação de um programa hiCUDA.

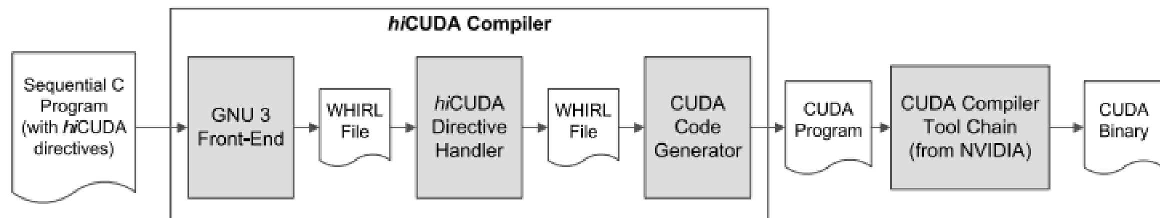


Figura 3.1: Fluxo de execução hiCUDA [Han e Abdelrahman 2011]

3.1.2 Exemplos de Programa em hiCUDA

Para demonstrar a utilização da linguagem hiCUDA e os benefícios que ela proporciona para o desenvolvimento das aplicações, será apresentado um algoritmo para multiplicação de matrizes [Han 2009]. No Código 3.1, é possível observar o algoritmo escrito originalmente em C. No Código 3.2, o mesmo algoritmo agora escrito na linguagem hiCUDA.

Código 3.1: Multiplicação de Matrizes em C [Han e Abdelrahman 2011]

```
1  float A[64][128];
2  float B[128][32];
3  float C[64][32];
4
5  randomInitArr((float*)A, 64*128);
6  randomInitArr((float*)B, 128*32);
7
8  for (i=0; i<64; ++i){
9      for (j=0;j<32;++j){
10         float sum = 0;
11         for (k=0; k<128; ++k){
12             sum+= A[i][k]*B[k][j];
13         }
14         C[i][j]=sum;
15     }
16 }
17 printMatrix((float*)C, 64, 32);
```

Código 3.2: Multiplicação de Matrizes em hiCUDA [Han e Abdelrahman 2011]

```
1  float A[64][128];
2  float B[128][32];
3  float C[64][32];
4
5  randomInitArr((float*)A, 64*128);
6  randomInitArr((float*)B, 128*32);
7
8  #pragma hicuda global alloc A[*][*] copyin
9  #pragma hicuda global alloc B[*][*] copyin
10 #pragma hicuda global alloc C[*][*]
```

```
11
12 #pragma hicuda kernel matrixMul tblock(4,2) thread(16,16)
13     for (i=0; i<64; ++i){
14 #pragma hicuda loop_partition over_tbblock over_thread
15         for (j=0;j<32;++j){
16             float sum = 0;
17             for (kk=0; kk<128; kk+=32){
18 #pragma hicuda shared alloc A[i][kk:kk+31] copyin
19 #pragma hicuda shared alloc B[kk:kk+31][j] copyin
20 #pragma hicuda barrier
21                 for(k=0; k<32; k++){
22                     sum+= A[i][k]*B[k][j];
23                 }
24 #pragma hicuda barrier
25 #pragma hicuda shared remove A B
26         }
27         C[i][j]=sum;
28     }
29 }
30 #pragma hicuda global copyout C[*][*]
31 #pragma hicuda global free A B C
32 printMatrix((float*)C, 64, 32);
```

O mesmo programa escrito em CUDA C pode ser analisado no Código 3.3. A maior quantidade de linhas de código, bem como a sua complexidade, são visíveis, demonstrando que a utilização de níveis mais altos de abstração podem facilitar a programação para GPUs.

Código 3.3: Multiplicação de Matrizes em CUDA C [Han e Abdelrahman 2011]

```
1 float A[64][128];
2 float B[128][32];
3 float C[64][32];
4
5 randomInitArr((float*)A, 64*128);
6 randomInitArr((float*)B, 128*32);
7
8 size = 64 * 32 * sizeof(float);
9 cudaMalloc((void**)&d_A,size);
10 cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
11 size = 128 * 32 * sizeof(float);
12 cudaMalloc((void**)&d_B,size);
13 cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
14
15 size = 64 * 32 * sizeof(float);
16 cudaMalloc((void**)&d_C,size);
17
18 dim3 dimBlock (16,16);
19 dim3 dimGrid (32/dimBlock.x, 64/dimBlock.y);
20
21 matrixMul<<<dimGrid, dimBlock>>>(
22     d_A, d_B, d_C, 128, 32);
```

```
23
24 cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
25
26 cudaFree(d_A);
27 cudaFree(d_B);
28 cudaFree(d_C);
29
30 __global__ void matrixMul (float *A, float *B, float *C, int wA, int wB){
31     int bx = blockIdx.x, by = blockIdx.y;
32     int tx = threadIdx.x, ty = threadIdx.y;
33
34     int aBegin = wA * 16 * by + wa * ty + tx;
35     int aEnd = aBegin + wA;
36     int aStep = 32;
37     int bBegin = 16 * bx + wB * ty + tx;
38     int bStep = 32 * wB;
39
40     __shared__ float As[16][32];
41     __shared__ float Bs[32][16];
42
43     float Csub = 0;
44
45     for (int a=aBegin, b=bBegin; a<aEnd; a+=aStep, b +=bStep){
46         As[ty][tx]=A[a]; As[ty][tx+16]=A[a+16];
47         Bs[ty][tx]=B[b]; Bs[ty+16][tx]=B[b+16*wB];
48
49         __syncthreads();
50
51         for (int k=0; k<32; ++k){
52             Csub+= As[ty][k]*Bs[k][tx];
53         }
54         __syncthreads();
55     }
56     C[wB*16*by+16*bx+wB*ty+tx]=Csub;
57 }
```

3.2 CUDA Fortran

Os principais pacotes utilizados para HPC (*High Performance Computing*) são originalmente escritos em Fortran. Pensando nisso, a empresa NVIDIA em acordo com o Grupo Portland (IGP) desenvolveram o compilador CUDA Fortran. CUDA Fortran [Fortran 2012] permite que os programadores continuem melhorando esses pacotes utilizando a linguagem Fortran nativa.

CUDA Fortran é um conjunto de extensões construído sobre a arquitetura CUDA. Tais extensões permitem a declaração de variáveis na memória da GPU, alocação dinâmica, transferências entre CPU e GPU e invocação de subrotinas GPU diretamente a partir da CPU. Permite ao programador utilizar declarações Fortran

90 para atribuir e alocar dados na GPU, ao invés de usar funções específicas CUDA, como demonstrado no trecho do Código 3.4.

Código 3.4: Alocação de memória com CUDA Fortran

```
real, device :: A(M,N)
real, device, allocatable :: B(:,:)
...
allocate(B(size(A,1), size(A,2)), stat=istat)
```

Em CUDA Fortran, os programas são escritos através de subrotinas e funções que recebem atributos para identificar seu local de execução. Nesse caso, as atribuições são definidas através do especificador *attributes*(*tipo*). Os tipos possíveis para sub-programas são:

- `host` - identifica uma sub-rotina que será chamada somente no processador hospedeiro;
- `global` - subrotinas que executaram no dispositivo, ou seja, um kernel chamado a partir do processador hospedeiro;
- `device` - define uma sub-rotina para execução somente no dispositivo, chamadas a partir do próprio dispositivo.

É possível observar as semelhanças com a linguagem CUDA C, algo interessante permitindo que o programador passe a reconhecer à sintaxe da linguagem CUDA C através da utilização de uma sintaxe Fortran, facilitando a migração entre as linguagens. Outra semelhança é a chamada de um *kernel* para execução, como mostra o Exemplo 3.5. Nesse exemplo, o *kernel* denominado `ksaxpy` é chamado através do comando `call ksaxpyy≪n/64, 64≫(n, a, x, y)`.

Código 3.5: Exemplo de definição e chamada de *kernel* [Fortran 2012]

```
attributes (global) subroutine ksaxpy (n,a,x,y)
    real, dimension(*) :: x,y
    real, value :: a
    integer, value :: n,i
    i=(blockidx % x-1) * blockdim % x + threadidx % x
    if (i<=n) y(i) = a * x(i) + y(i)
end subroutine

subroutine solve(n,a,x,y)
    real, divece, dimension(*) :: x,y
    real :: a
    integer :: n
    call ksaxpy<<<n/64, 64>>>(n,a,x,y)
end subroutine
```

A identificação das *threads* é realizada igualmente a linguagem CUDA C, com os índices de acesso em um bloco de *threads* (*blockidx* e *blockidy*) e os índices de acesso a *thread* (*threadidx* e *threadidy*). No exemplo da Figura 3.5, cada *thread* executara uma iteração do laço.

A hierarquia de memória também é expressa através de atributos específicos, que indicam onde cada variavel esta alocada. Os qualificadores de váriaveis podem ser:

- `device`: variável alocada no dispositivo para memória global;
- `constant`: indica uma variável que deverá ser alocada na memória de constantes;
- `shared`: dados alocados na memória compartilhada;
- `texture`: variável mapeada para memória de texturas.

A cópia de dados entre CPU e GPU também segue a sintaxe de vetores do Fortran 90:

```
real ::  A(M,N) ! A instantiated in host memory
real,device ::  Adev(M,N) ! Adev instantiated in GPU memory
...
Adev = A ! Copy data from A (host) to Adev (GPU)
...
A = Adev !  Copy data from Adev (GPU) to A (host)
```

Essas atribuições permitem que os programadores expressem todo seu programa utilizando Fortran, não exigindo muitas chamadas de biblioteca em tempo de execução. Nos Códigos 3.6 e 3.7, é possível observar um exemplo básico da utilização de CUDA Fortran para multiplicação de matrizes. O exemplo demonstra como especificar um *kernel* e executá-lo. Outras operações podem ser visualizadas, tais como a troca de dados entre CPU e GPU, sincronização entre *threads* e uso da memória compartilhada.

Código 3.6: Lançando o *kernel* [Group 2012]

```
1 subroutine mmul( A, B, C )
2
3   use cudafor
4   real, dimension(:,:) :: A, B, C
5   integer :: N, M, L
6   real, device, allocatable, dimension(:,:) :: Adev,Bdev,Cdev
```

```
  type(dim3) :: dimGrid, dimBlock

  N = size(A,1) ; M = size(A,2) ; L = size(B,2)
  allocate( Adev(N,M), Bdev(M,L), Cdev(N,L) )
  Adev = A(1:N,1:M)
  Bdev = B(1:M,1:L)
  dimGrid = dim3( N/16, L/16, 1 )
  dimBlock = dim3( 16, 16, 1 )
  call mmul_kernel<<<dimGrid,dimBlock>>>( Adev,Bdev,Cdev,N,M,L )
  C(1:N,1:M) = Cdev
  deallocate( Adev, Bdev, Cdev )

end subroutine
```

Código 3.7: Definição do *kernel* [Group 2012]

```
attributes(global) subroutine MMUL_KERNEL( A,B,C,N,M,L)

  real,device :: A(N,M),B(M,L),C(N,L)
  integer,value :: N,M,L
  integer :: i,j,kb,k,tx,ty
  real,shared :: Ab(16,16), Bb(16,16)
  real :: Cij

  tx = threadidx%x ; ty = threadidx%y
  i = (blockidx%x-1) * 16 + tx
  j = (blockidx%y-1) * 16 + ty
  Cij = 0.0
  do kb = 1, M, 16
     Ab(tx,ty) = A(i,kb+ty-1)
     Bb(tx,ty) = B(kb+tx-1,j)

     call syncthreads()

     do k = 1, 16
        Cij = Cij + Ab(tx,k) * Bb(k,ty)
     enddo

     call syncthreads()
  enddo
  C(i,j) = Cij
end subroutine
```

3.3 JCuda

JCuda é uma API para linguagem JAVA que suporta a programação em CUDA. Nesse contexto, o mais natural para realizar a tradução de uma linguagem de mais baixo nível para Java seria a utilização de JNI (Java Native Interface) [Liang 1999], que permite a integração de código escrito em Java com outras linguagens, tais como C e C++.

Contudo, esse processo de tradução é tedioso e propenso a erros devido aos vários passos necessários. Primeiramente, seriam necessários a escrita de um código Java, um código utilizando JNI para C, o qual será executado no processador hospedeiro e um código CUDA C para execução no dispositivo GPU [Yan, Grossman e Sarkar 2009]. Todo esse trabalho levou ao desenvolvimento de um compilador capaz de gerar os códigos automaticamente, assim como todo mapeamento e transferências de dados.

Em resumo, a ferramenta permite ao programador inicializar um dispositivo GPU, bem como definir e carregar seus *kernels* para execução. A API disponibiliza diversos métodos para o gerenciamento dos dispositivos, alocação e transferências de memória, além de métodos para funções matemáticas pré-definidas. No Código 3.8, é possível observar sua utilização e sintaxe.

Código 3.8: Exemplo de código JCuda [Yan, Grossman e Sarkar 2009]

```
1  double[ ][ ] l_a = new double[NUM1][NUM2];
2  double[ ][ ][ ] l_aout = new double[NUM1][NUM2][NUM3];
3  double[ ][ ] l_aex = new double[NUM1][NUM2];
4
5  initArray(l_a); initArray(l_aex); //initialize value in array
6
7  int [ ] ThreadsPerBlock = {16, 16, 1};
8  int [ ] BlocksPerGrid = new int[3]; BlocksPerGrid[3] = 1;
9  BlocksPerGrid[0] = (NUM1 + ThreadsPerBlock[0] - 1) / ThreadsPerBlock[0];
10 BlocksPerGrid[1] = (NUM2 + ThreadsPerBlock[1] - 1) / ThreadsPerBlock[1];
11
12 /* invoke device on this block/thread grid */
13 cudafoo.foo1 <<<< BlocksPerGrid, ThreadsPerBlock >>>> (l_a, l_aout, l_aex);
14 printArray(l_a); printArray(l_aout); printArray(l_aex);
15 ... ... ...
16 static lib cudafoo (?cfoo?, ?/opt/cudafoo/lib?) {
17 acc void foo1 (IN double[ ][ ]a, OUT int[ ][ ][ ] aout, INOUT float[ ][ ] aex);
18 acc void foo2 (IN short[ ][ ]a, INOUT double[ ][ ][ ] aex, IN int total);
19 }
```

A interface para acessar o código CUDA C externo é apresentada nas linhas 16-19. Nesse ponto, é demonstrado o modificador `acc`, que indica a chamada para uma função *kernel*. Os argumentos da função são declarados com `IN`, `OUT` ou `INOUT`. Esses argumentos indicam o momento da transferência dos dados, que pode ocorrer antes ou depois do lançamento do kernel, bem como ambos os casos. Essa foi a forma encontrada para indicar ao compilador JCuda o momento das transferências de dados, ficando a cargo deste tais tarefas.

3.3.1 O Compilador JCuda

O compilador JCuda é baseado no compilador Poliglota [Nystrom, Clarkson e Myers 2002], um compilador de entrada para linguagem Java. Os códigos gerados pelo compilador JCuda a partir do Código 3.8, podem ser observados nos Códigos 3.9 e 3.10. No Código 3.9, a classe Java estática gerada a partir da função lib (linha 16) do código anterior, que introduz as declarações com os nomes das funções nativas para as funções introduzidas no código JCuda para foo1 (linha 17) e foo2 (linha 18). Observa-se que, nesse momento, são adicionados três parâmetros, utilizados para definição do *grid*, dos blocos e o tamanho da memória compartilhada.

Código 3.9: Código Java estático [Yan, Grossman e Sarkar 2009]

```
1  private static class cudafoo {
2      native static void HelloL_00024cudafoo_foo1(double[][] a, int[][][] out,
3          float[][] aex, int[] dimGrid, int[] dimBlock, int sizeShared);
4      static void foo1(double[][] a, int[][][] aout, float[][] aex, int[] dimGrid,
5           int[] dimBlock, int sizeShared) {
6          HelloL_00024cudafoo_foo1(a, aout, aex, dimGrid, dimBlock, sizeShared);
7      }
8      native static void HelloL_00024cudafoo_foo2(short[][] a, double[][][] aex,
9          int total, int[] dimGrid, int[] dimBlock, int sizeShared);
10     static void foo2(short[][] a, double[][][] aex, int total, int[] dimGrid,
11         int[] dimBlock, int sizeShared) {
12         HelloL_00024cudafoo_foo2(a, aex, total, dimGrid, dimBlock, sizeShared);
13     }
14     static { java.lang.System.loadLibrary( "HelloL_00024cudafoo_stub" ); }
15 }
```

No Código 3.10, é apresentado o código de ligação entre Java e CUDA C gerado a partir da função foo1 pela ferramenta JNI, onde são adicionadas as transferências de dados de acordo com os modificadores IN, OUT e INOUT, passados como parâmetro.

Código 3.10: Código C gerado [Yan, Grossman e Sarkar 2009]

```
1  extern global void foo1(double * d_a, signed int * d_aout, float * d_aex);
2  JNIEXPORT void JNICALL
3  Java HelloL_00024cudafoo_HelloL_100024cudafoo_1foo1(JNIEnv *env, jclass cls,
4      jobjectArray a, jobjectArray aout, jobjectArray aex, jintArray dimGrid,
5      jintArray dimBlock, int sizeShared) {
6      /* copy array a to the device */
7      int dim_a[3] = {2};
8      double * d_a = (double*) copyArrayJVMToDevice(env, a, dim_a, sizeof(double));
9      /* Allocate array aout on the device */
10     int dim_aout[4] = {3};
11     signed int * d_aout = (signed int*) allocArrayOnDevice(env, aout, dim_aout,
12         sizeof(signed int));
13     /* copy array aex to the device */
14     int dim_aex[3] = {2};
```

```
15      float * d_aex = (float*) copyArrayJVMToDevice(env, aex, dim_aex, sizeof(float));
16      /* Initialize the dimension of grid and block in CUDA call */
17      dim3 d_dimGrid; getCUDADim3(env, dimGrid, &d_dimGrid);
18      dim3 d_dimBlock; getCUDADim3(env, dimBlock, &d_dimBlock);
19      foo1 <<< d_dimGrid, d_dimBlock, sizeShared >>> ((double *)d_a,
20          (signed int *)d_aout, (float *)d_aex);
21      /* Free device memory d_a */
22      freeDeviceMem(d_a);
23      /*copy array d_aout->aout from device to JVM, and free device memory d_aout*/
24      copyArrayDeviceToJVM(env, d_aout, aout, dim_aout, sizeof(signed int));
25      freeDeviceMem(d_aout);
26      /* copy array d_aex->aex from device to JVM, and free device memory d_aex */
27      copyArrayDeviceToJVM(env, d_aex, aex, dim_aex, sizeof(float));
28      freeDeviceMem(d_aex);
29      return;
30  }
```

3.4 pyCUDA

A API pyCUDA, desenvolvida por Andreas Klöckner [Klöckner 2012], visa dar suporte para arquitetura CUDA sobre a linguagem Python. Com essa ferramenta, é possível escrever um *kernel* em CUDA C puro e chamá-lo dentro de um código Python. A API disponibiliza o coletor de lixo da linguagem e a biblioteca Numpy da linguagem Python.

Em pyCUDA, um *kernel* é definido como mostra o Código 3.11. Nesse caso, é utilizado uma função chamada SourceModule que irá receber o código escrito em CUDA C que define o *kernel*. O código apresentado será compilado automaticamente pelo compilador NVCC [NVCC 2012] da arquitetura CUDA.

Código 3.11: Definição de um kernel em pyCUDA [Klöckner 2012]

```
1 mod = SourceModule("""
2     __global__ void doublify(float *a){
3         int idx = threadIdx.x + threadIdx.y*4;
4         a[idx] *= 2;
5     }
6     """)
```

O código será obtido através de uma variável (`mod`) que armazena o módulo de software que será obtido como uma função (`func`), demonstrado no Código 3.12.

Código 3.12: Obtendo o *kernel* como uma função [Klöckner 2012]

```
1 func = mod.get_function("doublify")
2 func(a_gpu, block=(4,4,1))
```

Para uma certa variável arbitrária x, a alocação da memória e transferências de dados são realizadas explicitamente através das funções `mem_alloc(`x`.nbytes)` e

`memcpy_htod(`x_gpu`, `x`)`, funções internas ao controlador CUDA.

Em Python, utiliza-se frequentemente a biblioteca Numpy para geração de dados. Os dados gerados possuem precisão dupla, sendo necessária a transformação para precisão simples, necessária para maioria das placas gráficas. Essa transformação é realizada através da função x`.astype(numpy.float32)` da própria linguagem Python.

O último passo consiste na inicialização do *kernel* com os dados da memória alocada e o tamanho do bloco definido. No Código 3.13, é possível observar esse último passo além das transferências de dados entre o processador hospedeiro e o dispositivo GPU discutidas anteriormente.

Código 3.13: Inicialização do kernel [Klöckner 2012]

```
1 x_doubled = numpy.empty_like(x)
2 cuda.memcpy_dtoh(x_doubled, x_gpu)
3 print x_doubled
4 print x
```

3.5 Chestnut

A linguagem Chestnut simplifica a sintaxe do modelo de programação C, sendo muito familiar para programadores C e Fortran, sendo parte de um grande sistema de programação para GPU que inclui um modelo de programação multi-nível e compiladores em cada nível de tradução [Stromme, Carlson e Newhall 2012].

O código Chestnut consiste de um código sequencial com laços paralelos sobre os dados. Ou seja, uma sequencia de instruções com partes paralelas executadas sobre a GPU. O paralelismo é expresso dentro de declarações paralelas, chamadas de contexto paralelo. Um contexto paralelo é definido pela expressão `foreach loop`, que contém um conjunto de instruções para acessar e modificar elementos de matrizes paralelas. Um exemplo na linguagem Chestnut é apresentado no Código 3.14.

Código 3.14: Multiplicação de matrizes [Stromme, Carlson e Newhall 2012]

```
1 RealArray2d a[10, 5] = read("a.data");
2 RealArray2d b[5, 8] = read("b.data");
3 RealArray2d output[10, 8];
4 foreach e in output
5     e = 0;
6     for (Int i=0; i < a.height; i++) {
7         e = e + a[e.x, i] * b[i, e.y];
8     }
9 end
```

```
10 write(output, "output.data");
```

É possível observar que o programador não precisa se preocupar em alocar memória e realizar transferências entre CPU e GPU. Os dados declarados no contexto paralelo são automaticamente alocados na memória da GPU, sendo copiados para memória da CPU somente se acessados fora do contexto paralelo.

Trocas explícitas de dados entre CPU e GPU podem ser realizadas através de arquivos. Os dados de um vetor ou matriz podem ser lidos ou escritos em um arquivo de dados. Essa abstração nos acessos e transferências de dados sobre a memória do dispositivo simplificam a programação, mas não favorecem o desempenho. Outros conceitos que a linguagem abstrai são os blocos e *threads*, bem como o mapeamento da execução paralela nos dados da GPU. Em geral, o modelo de programação não é exposto ao programador.

A extrema simplicidade do modelo esconde inclusive estruturas de sincronização como semáforos e a comunicação explícita entre *threads*. Todas essas características acabam por limitar os tipos de paralelismo que podem ser expressos com a linguagem. Na Figura 3.2, é possível observar a arquitetura do sistema Chestnut.

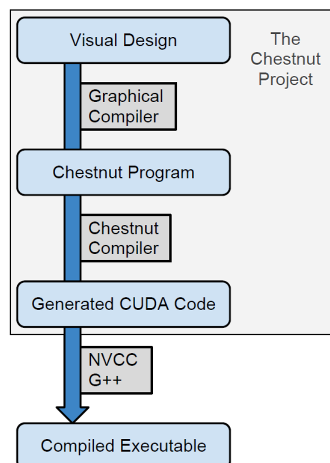


Figura 3.2: Sistema Chestnut [Stromme, Carlson e Newhall 2012]

O sistema conta com uma interface gráfica opcional para programadores iniciantes. O compilador gráfico traduz a representação gráfica para código fonte Chestnut. Por sua vez o compilador Chestnut irá traduzir o código Chestnut para código fonte CUDA C++ e *Thrust* [Bell e Hoberock]. Na sequência, o sistema faz uso do NVCC g++ para compilar o código CUDA C++/Thrust para um executável. O ambiente disponibilizá outras opções como a exportação dos resultados da compilação em cada nível.

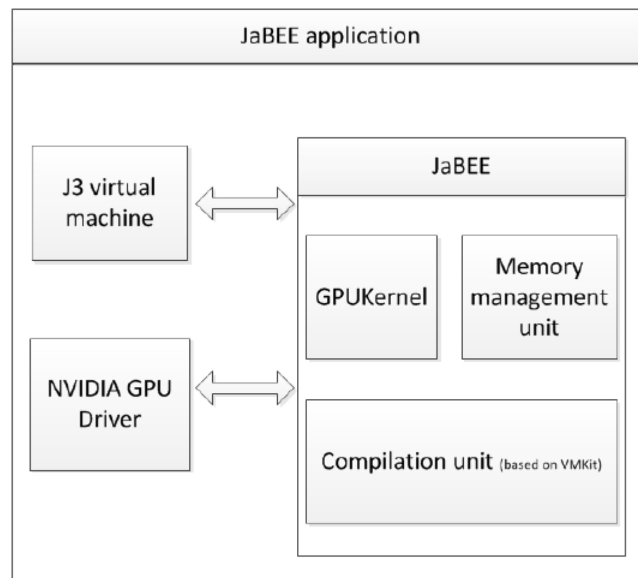


Figura 3.3: Arquitetura JaBEE [Zaremba, Lin e Grover 2012]

3.6 JaBEE

O Java Bytecode Environment Execution (JaBEE) tem por objetivo dar suporte a construções orientadas à objeto, despacho dinâmico, encapsulamento e criação de objetos em GPUs [Zaremba, Lin e Grover 2012]. O projeto visa facilitar a programação de GPU utilizando a linguagem Java, através da compilação dos programas e do gerenciamento transparente da memória.

A arquitetura do ambiente é composta por três componentes, e pode ser vista na Figura 3.3:

i. Uma classe Java que serve como base chamada `GPUKernel`, que provê uma interface Java para o código que deverá executar na GPU.

ii. Um compilador online que compila seletivamente *bytecode* Java para o código GPU.

iii. Um sistema de gerenciamento de memória, para realizar transferências de dados entre CPU e GPU.

No Código 3.15, é apresentado um exemplo de programa escrito em JaBEE.

Código 3.15: Exemplo de programa em JaBEE [Zaremba, Lin e Grover 2012]

```
1 public class Julia extends GPUKernel {
2
3 static Complex c= new Complex ( -0.8 , 0.156) ;
```

```
4 static int DIM =1000;
5 byte tab []= new byte [ DIM *DIM ];
6
7 byte julia (int x, int y) {
8     double jx =( double )( DIM /2-x)/( DIM /2) ;
9     double jy =( double )( DIM /2-y)/( DIM /2) ;
10    Complex a= new Complex (jx , jy);
11    for (int i =0;i <255; i ++) {
12        if (a.mul (a). add (c). magnitude2 () > 1000)
13            return i;
14    }
15    return 255;
16 }
17
18 public void run () {
19    int i= BlockId .x+ BlockId .y* GridSize .x;
20    tab [i]= julia ( BlockId .x, BlockId .y);
21 }
22
23 public static void main ( String [] args ) {
24    Julia m= new Julia ();
25    m. start ( new Dim(DIM , DIM ), new Dim ());
26 }
27 }
```

A execução incia-se por uma chamada a partir de um código Java para executar um `GPUKernel`. O código é compilado em duas etapas. Inicialmente, o *bytecode* Java é compilado em um código intermediário LLVM IR. Esse código é então ligado a uma biblioteca que implementa funções matemáticas básicas e métodos específicos para GPU. Logo, o código está livre de referencias externas e pode ser transformado em código PTX. Esse fluxo de compilação pode ser observado no diagrama da Figura 3.4.

3.7 Rootbeer

Rootbeer [Pratt-Szeliga, Fawcett e Welch 2012] é um projeto no qual os autores também tem por objetivo facilitar a programação para GPU utilizando a linguagem Java. Suas principais metas são simplificar a escrita de código em Java e a (de)serialização com geração e lançamento automático de *kernels*. É uma ferramenta livre e de código aberto sob a licença GNU General Public License version 3 (GPLv3).

A ferramenta é considerada pelos autores como sendo a mais completa ferramenta para computação em GPU partindo de um código escrito na linguagem Java, permitindo a utilização da maioria dos recursos da linguagem Java exceto invocação de métodos, reflexão e métodos nativos. Segundo os autores, a ferramenta passou

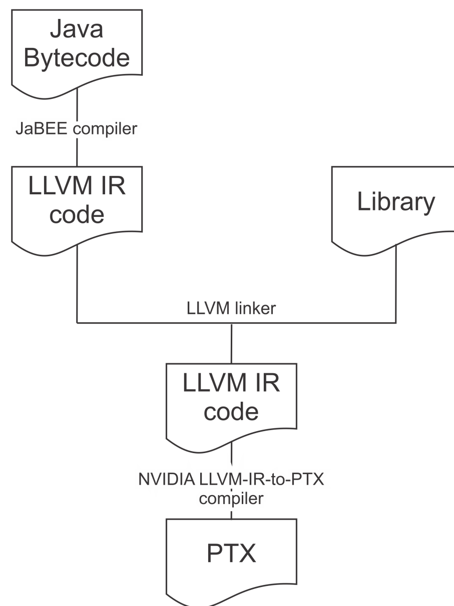


Figura 3.4: Fluxo de compilação [Zaremba, Lin e Grover 2012]

por um longo processo de testes, sendo considerada estável.

Partindo do principio de que o paralelismo é obtido de uma forma clássica, através de vários laços aninhados com chamadas de funções interligadas. Os autores verificaram que, em geral, o laço mais interno não possui alto paralelismo para obter desempenho significativo sobre GPUs. Nessa situação, como já observado anteriormente, se faz necessária a reestruturação do código, ou seja, processo com varias etapas manuais onde o programador verifica várias opções de configurações até obter um melhor desempenho, levando a uma tarefa complexa e sujeita a erros.

Tratando-se de uma ferramenta para linguagem Java, como algumas já citadas anteriormente, uma das principais etapas para reestruturação de um algoritmo escrito em Java consiste no mapeamento cuidadoso de grafos complexos compostos por objetos Java para arrays de tipos básicos, tarefa que passa a ser automática com a utilização de Rootbeer.

Outro ponto muito importante é a escrita do *kernel*. Em ferramentas como jCUDA deve ser em outra linguagem, em geral CUDA C. Os autores preocuparam-se em permitir que o programador simplesmente escreva seu código em Java puro, sem modificar sua sintaxe.

Semelhante a proposta da linguagem JaBEE, os autores propuseram uma interface chamada *Kernel*, apresentada no Código 3.16. Essa interface possui um método chamado `gpuMethod`, porta de entrada do programador para a GPU. Nesse

método o desenvolvedor obtêm dados para a GPU definindo campos privados na classe que implementa a interface *Kernel*. Logo, o compilador identifica todos os campos e objetos alcançáveis pelo método e copia todos para à GPU. No final da computação todos os dados são copiados de volta para o CPU, automaticamente.

Código 3.16: Interface Kernel [Pratt-Szeliga, Fawcett e Welch 2012]

```
public interface Kernel {
    void gpuMethod();
}
```

Para entender melhor a proposta do compilador Rootbeer, será analisado um exemplo apresentado pelos autores. No Código 3.17, é possível observar uma classe chamada `ArraySum` que implementa a interface *Kernel*. Essa classe possui três variáveis privadas, `source`, `ret` e `index` que serão utilizadas para transferências de objetos entre CPU e GPU.

Código 3.17: Implementação da interface Kernel [Pratt-Szeliga, Fawcett e Welch 2012]

```
public class ArraySum implements Kernel {
    private int[] source;
    private int[] ret;
    private int index;
    public ArraySum (int[] src, int[] dst, int i){
        source = src; ret = dst; index = i;
    }
    public void gpuMethod(){ int sum = 0;
        for(int i = 0; i < array.length; ++i){
            sum += array[i];
        }
        ret[index] = sum;
    }
}
```

Toda computação implementada no método `gpuMethod` será tratado pelo *cross-compiler*, ou seja, será interpretado para a arquitetura GPU. As informações que serão utilizadas pela classe `ArraySum` são configuradas em outra classe chamada `ArraySumApp`, responsável pelo lançamento do *kernel*, apresentada no Código 3.18.

Código 3.18: Lançamento do Kernel [Pratt-Szeliga, Fawcett e Welch 2012]

```
public class ArraySumApp {
    public int[] void sumArrays(List<int[]> arrays){
        List<Kernel> jobs = new ArrayList<Kernel>();
        int[] ret = new int[arrays.size()];
        for(int i = 0; i < arrays.size(); ++i){
            jobs.add(new ArraySum(arrays.get(i), ret, i));
        }
        Rootbeer rootbeer = new Rootbeer();
```

```
9          rootbeer.runAll(jobs);
10         return ret;
11     }
12 }
```

No código, podemos observar a criação de uma lista de *kernels* que serão utilizados. Cada *kernel* recebe uma lista de inteiros para somar. Os *kernels* são armazenados em um vetor que será inicializado na linha 9. É importante observar que o compilador é identificado através de um objeto chamado `rootbeer`, inicializado na linha 8. O código é transformado em um arquivo de extensão *jar* correspondente. Logo, esse código passará por um módulo específico denominado *Rootbeer Static Transformer* (RST), sendo transformado em um arquivo de extensão *jar* final, pronto para ser utilizado.

O módulo RST é responsável pela tradução do código contido no arquivo de extensão *jar* inicial em código CUDA C, que será posteriormente traduzido para GPU. Para isso, o módulo utiliza o *framework* Soot [Vallee-Rai 2000], que transformará os arquivos de extensão *class* em uma representação intermediária chamada Jimple [Vallee-Rai 2000, Vallee-Rai e Hendren 1998]. O código CUDA C é gerado no final da fase de compilação. Esse código será então compilado pelo compilador `nvcc` da linguagem CUDA C e encapsulado no arquivo de extensão *jar* final, ficando acessível ao compilado Rootbeer para execução.

O *bytecode* Java é gerado após a fase de geração do código CUDA C. Nessa fase, serão serializados todos os tipos acessíveis a partir da classe *Kernel*. Ao final das fases descritas, é adicionada uma nova interface para suporte a interface *Kernel*, chamada *CompiledKernel*. Essa interface facilitará ao compilador Rootbeer obter o código *cubin*, gerado na primeira fase, e serializar os objetos instanciados na classe Kernel. O código representado em Jimple é então transformado novamente em bytecode Java. o arquivo de extensão *class* resultante é empacotado juntamente com outros arquivos de extensão *class* necessários para o comilador Rootbeer em um arquivo de extensão *jar*. Esse arquivo resultante pode ser usado normalmente como uma aplicação Java.

Rootbeer é uma ferramenta que desperta nosso interesse por apresentar uma tradução transparente ao usuário. O compilador demonstra que é possível utilizar uma linguagem de alto nível para implementação em uma arquitetura com tantas complexidades como a arquitetura GPU. Alguns dados de uma avaliação de desempenho simples de Rootbeer são apresentados na próxima seção.

3.8 Avaliação de Desempenho

As linguagens e ferramentas apresentadas possuem características diferentes. As abstrações propostas por algumas das ferramentas são de fácil gerenciamento. O maior problema está na tradução das linguagem para uma linguagem específica para GPU, no caso a linguagem CUDA C/C++.

A linguagem hiCUDA demonstra-se como uma ferramenta muito simples e que fornece resultados de execução bem semelhantes a linguagem CUDA C. A tradução do código hiCUDA para CUDA C não afeta significativamente seu desempenho, tornando a ferramenta muito atrativa para comunidade.

CUDA Fortran por sua vez permite que o programador descreva seu programa para execução na GPU utilizando a sintaxe Fortran. O compilador é o mesmo utilizado pela linguagem Fortran. O que permite a utilização de CUDA sobre Fortran são os arquivos com a extensão *cuf*. Esses arquivos são tratados pelo compilador como arquivos fonte Fortran que contém extensões CUDA e são pré-processados.

As APIs pyCUDA e JCuda constituem uma boa solução para programadores Python e Java, pois permite a integralização da linguagem CUDA de maneira simples e de fácil gerenciamento. Porém, as APIs não permitem a criação de objetos o que vai contra o modelo orientado à objetos o qual as linguagens seguem.

Como vimos, a linguagem Chestnut é parte de um sistema para desenvolvimento de aplicações sobre GPU. No entanto, seu desempenho é muito inferior a linguagem CUDA C++ pura. Na Tabela 3.1, é possível observar os tempos de execução para três aplicações: GOL (*Game of Live*), MatrixMult (multiplicação de matrizes) e Heat (algoritmo de dispersão) em comparação a linguagem CUDA C++.

Benchmark	Sequencial	CUDA	Chestnut	*Speedup*
GOL	371.0s	0.8s	1.7s	217
MatrixMult	398.6s	0.2s	1.1s	347
Heat	349.5s	1.7s	5.4s	65

Tabela 3.1: Tempos de execução nas versões sequencial, CUDA e Chestnut [Stromme, Carlson e Newhall 2012]

Os tempos demonstram que a linguagem possui um desempenho inferior ao código escrito na versão em CUDA C puro. Essa é uma consequência da tradução do

código Chestnut para código CUDA C++, que é realizado nas etapas de compilação. O principal objetivo da linguagem é a facilidade de programação, ou seja, a linguagem cumpre seu papel mas deixa a desejar em termos de desempenho. Apesar dos tempos de execução inferiores, ainda assim podemos considerar o desempenho da linguagem Chestnut em relação a versão sequencial. Isso se dá devido a utilização do suporte para memória compartilhada e a sincronização entre *threads* que Chestnut adota da linguagem CUDA.

A linguagem JaBEE possui uma abordagem mais interessante em termos de orientação a objetos. A linguagem permite formas de manipulação de objetos não permitidas em outras ferramentas. Assim como Chestnut, busca a transparência no gerenciamento da memória. Seu desenvolvimento é recente e atualmente existe um protótipo em desenvolvimento. Seu desempenho é inferior a CUDA C puro. A Tabela 3.2 apresenta o *speedup* para a aplicação apresentada na seção anterior. Segundo os autores, esses valores devem ser melhorados com otimizações futuras.

	J3	Oracle Java	JaBEE	CUDA
Program P	1	1.26	4.15	9.97
Program O	0.30	1.27	1.04	3.59

Tabela 3.2: *Speedup* de avaliação da linguagem JaBEE [Zaremba, Lin e Grover 2012]

Os programas Rootbeer apresentam desempenho promissor, levando em consideração as fases necessárias para transformação do código escrito em Java para um código que possa ser executado sobre a GPU. Apresentamos dois testes que nos chamaram a atenção. Os tempos de execução do primeiro teste são apresentados na Tabela 3.3, o qual consiste em um algoritmo para multiplicação de matrizes densas.

System	**Time (ms)**
JavaOnly	3.531,551
Java with Rootbeer	52,662
Event	**Time (ms)**
Rootbeer Serialization	557
GPU Execution	51,687
Rootbeer Deserialization	418

Tabela 3.3: Tempos de execução Java vs Rootbeer - Multiplicação de matrizes [Pratt-Szeliga, Fawcett e Welch 2012]

Observa-se que o tempo de execução puro contribui com mais de 95% do tempo total. Já na Tabela 3.4, os tempos não refletem a eficiência do compilador. Isso ocorre porque a complexidade da computação é muito baixa, sendo a transferência de dados o gargalo na aplicação, justificando os tempos de serialização e deserealização altos. Os dados demonstram que deve-se evitar ao máximo as transferências de dados entre o processador hospedeiro e o dispositivo GPU.

System	**Time (ms)**
JavaOnly	129
Java with Rootbeer	502
Event	**Time (ms)**
Rootbeer Serialization	167
GPU Execution	125
Rootbeer Deserialization	210

Tabela 3.4: Tempos de execução Java vs Rootbeer - Sobel Filter [Pratt-Szeliga, Fawcett e Welch 2012]

Capítulo 4

A Linguagem Fusion

A programação de propósito geral utilizando aceleradores gráficos tornou-se nos últimos anos uma ampla área de estudos. As linguagens tem evoluído muito, mas ainda há muito para ser desenvolvido. Linguagens que ofereçam abstrações de alto nível favorecem o desenvolvimento das aplicações, tornando mais simples a descrição de operações complexas sobre a arquitetura de um computador alvo. Conciliar abstrações de alto nível e desempenho em linguagens de programação é um problema de especial interesse na Computação de Alto Desempenho (CAD), especialmente visando plataformas de computação paralela.

Assim, com a proliferação do uso de GPUs, a importância do desenvolvimento de ferramentas que permitam a utilização de GPUs para propósito geral de maneira prática e simplificada cresce a cada dia, motivando a evolução nos modelos de programação, permitindo o surgimento de novas linguagens de programação que facilitem o desenvolvimento voltado a esses dispositivos. Antes, utilizavam-se rotinas pré-definidas que evoluíram para bibliotecas complexas com um grande poder de abstração, agilizando esse desenvolvimento. Porém, as abstrações presentes nas linguagens de programação atuais não são adequadas para o desenvolvimento sobre os aceleradores gráficos, o que tem sua origem na íntima relação dessas linguagens com a arquitetura Von Neumann.

Este capítulo descreve a principal contribuição deste trabalho, uma linguagem de alto nível que tem por objetivo facilitar a programação de aceleradores gráficos de arquitetura Kepler GK110, a qual denominamos Fusion, o qual explora a expressividade adicional do modelo de programação oferecida por esses dispositivos. Fusion tem sido inicialmente prototipada como uma extensão da linguagem Java.

A principal característica de Fusion é servir como ponte para ligar uma

computação paralela realizada por um conjunto de *threads* homogêneas que executam sobre um processador hospedeiro com múltiplos núcleos (*multi-core*) a um dispositivo GPU, de forma que cada *thread* possa acelerar uma computação local de forma independente em relação às outras, sem gargalos de sincronização, o que nos levou ao conceito de *kernel paralelo*, representado por um conjunto de kernels disparados simultaneamente, cada qual encapsulado em um *thread* do processador hospedeiro. A noção de *kernel* paralelo, o qual constitui uma contribuição inovadora deste trabalho de pesquisa, é possível em dispositivos que suportam o Hyper-Q.

Outra característica importante de Fusion é a adaptação ao paradigma de orientação a objetos, onde a tecnologia de paralelismo dinâmico também se faz essencial.

4.1 Características e Principais Conceitos

A linguagem Fusion foi desenvolvida segundo um conjunto de princípios, os quais motivaram a idealização das abstrações que introduz. Esta seção apresenta e motiva os princípios que tem norteado as decisões do projeto da linguagem Fusion.

4.1.1 Usuários Alvo

Atualmente, existem muitos desenvolvedores interessados no poder computacional oferecido pelos aceleradores gráficos. Esses usuários distribuem-se tanto no meio acadêmico quanto no meio industrial. No meio acadêmico, usuários interessados em aplicações científicas demonstram preocupação com a complexidade das técnicas do modelo de programação dos aceleradores gráficos para obter o melhor aproveitamento dos recursos de processamento, já que frequentemente não se tratam de especialistas em programação, mas especialistas no domínio de interesse das suas aplicações. Entretanto, nesse meio não existem preocupações muito severas com relação ao tempo de implementação e aos recursos disponíveis.

No meio industrial, por outro lado, objetiva-se uma maior produtividade no desenvolvimento, a fim de economizar tempo e recursos, sem impacto sobre a qualidade (desempenho e funcionalidades) dos produtos gerados, sendo por isso importante também a curva de aprendizado das técnicas e modelo de programação dos aceleradores gráficos por parte dos programadores especializados.

A fim de atender aos usuários dos dois meios descritos acima, a linguagem Fusion deve ser capaz de separar as preocupações com otimizações das computações para a GPU das preocupações com a interface com a funcionalidade implementada, evitando o retrabalho. Nesse sentido, são reconhecidos dois tipos de usuários

para linguagem, sob a perspectiva da aplicação final. O *usuário desenvolvedor*, responsável por implementar partes reusáveis de software otimizadas para execução nos aceleradores gráficos, que farão parte das aplicações, e o *usuário especialista*, que fará uso dessas partes reusáveis através de uma interface mais abstrata, voltada ao domínio da aplicação.

Portanto, as preocupações com os detalhes arquiteturais e técnicas de programação peculiares dos aceleradores gráficos ficam a cargo do usuário desenvolvedor. Todo paralelismo fica encapsulado em partes independentes e reusáveis de software, sendo papel do usuário especialista apenas utilizá-las sem maiores preocupações de como encontra-se implementado. Esse é um dos importantes princípios da engenharia de software moderna.

4.1.2 Paradigma de Programação: Orientação a objetos

A linguagem CUDA, apresentada na Seção 2.5, é atualmente a mais utilizada para implementação de programas sobre GPUs. Por tratar-se de uma extensão, essa linguagem possui uma sintaxe semelhante a linguagem C, que estende, sendo considerada uma linguagem de baixo nível de abstração. Por isso, torna-se complexa para desenvolvedores que não possuem conhecimento sobre detalhes da arquitetura das GPU, bem como de suas técnicas e modelos de programação.

O paradigma da orientação a objetos, doravante chamado de OO, tem por premissa a abstração de dados, sendo muito difundido dentre as linguagens de programação de mais alto nível, por permitir a modelagem de um problema real através da abstração de objetos. Os objetos surgiram como abstração linguística inicialmente na linguagem Simula 67 [Dahl 1968, Dahl 2002]. Porém, a primeira linguagem totalmente orientada ao emprego de objetos como unidades básicas de composição de software, por isso dita puramente orientada a objetos, foi Smalltalk [Deutsch e Goldberg 1991]. Nos dias atuais, C++, Java e C# são os representantes mais disseminados dentre as linguagens que suportam esse paradigma, embora não sejam puramente orientadas a objetos, por oferecerem abstrações que permitem trabalhar com outras técnicas conhecidas de estruturação de programas, como tipos abstratos de dados e componentes.

O paradigma OO pode tornar-se complexo para ser usado para representar computações sobre GPUs, pois um objeto precisa manter definidos seu estado interno e comportamento. O estado interno de um objeto é representado pelas valorações de seus atributos. Os métodos desse objeto oferecem um meio para a observação

externa do seu estado e representam seu comportamento, definindo como o seu estado pode ser modificado de um estado seguro para outro estado seguro, os quais satisfazem uma propriedade dita *invariante*[1].

OO tornou-se muito utilizado por um princípio chamado *encapsulamento*, que afirma que não é necessário conhecer como um objeto é implementado, ou seja, quem são os atributos que representam seu estado, para fazer suposições sobre sua implementação. Assim, é possível alterar um objeto sem comprometer a aplicação, desde que a propriedade invariante seja mantida pelos métodos.

Portanto, o estado de um objeto só pode ser alterado se acessado através de uma interface específica e bem delimitada, característica forte em uma linguagem orientada a objetos. No contexto das GPUs, o estado de um objeto não pode ser mantido de forma segura em sua memória global, uma vez que sua principal função é a realização de cálculos e não o armazenamento de dados. Porém, a memória global é persistente, o que permite a um objeto manter seus atributos no dispositivo, uma vez que existem garantias de que ao relançarmos um *kernel* os dados do lançamento anterior estarão presentes na memória global da GPU. Porém, essa técnica exige cuidados. É de responsabilidade do programador o gerenciamento da alocação e movimento de dados entre a CPU e a GPU, uma vez que podem existir dados intermediários que necessariamente devem ser atualizados para uma computação correta.

As características citadas justificam à arquitetura CUDA não seguir esse paradigma de programação, visando apenas a utilização da GPU como coprocessador aritmético. Porém, uma abordagem baseada na orientação a objetos pode trazer benefícios para programação sobre GPUs, simplificando e reduzindo o tempo e esforço de desenvolvimento, o que interessa especialmente aos usuários do meio comercial, bem como tornando os códigos de aplicações que fazem uso do potencial de GPUs mais abstratos, apresentando dependências mínimas de suposições de baixo nível sobre a arquitetura CUDA, o que interessa especialmente aos usuários do meio acadêmico. Por esse motivo, Fusion introduz o conceito de *objeto acelerador*, uma extensão do conceito de objeto tradicional para lidar com partes de um software que deseja-se acelerar sua execução em uma GPU.

[1]Uma invariante pode ser definida como um predicado lógico que especifica restrições sobre os valores das atributos do objeto, segundo algum formalismo lógico.

Abordagens Orientadas a Objetos para GPGPU

No estudo realizado no Capítulo 3, observa-se que todas as abordagens buscam a ligação de uma linguagem alvo a arquitetura CUDA. No contexto de orientação a objetos, as ferramentas para as linguagens que seguem esse paradigma, tais como JCuda e pyCUDA, permitem a integração da linguagem com a arquitetura CUDA. Objetos definidos nessas ferramentas possuem métodos escritos em CUDA C, exigindo conhecimento da arquitetura CUDA. Além disso, as transferências de dados devem ser realizadas explicitamente.

Já as ferramentas JaBEE e Rootbeer permitem a definição de um objeto através de uma biblioteca que realiza a ligação com CUDA, permitindo a criação de um objeto, em mais alto nível, com seu comportamento acelerado pela GPU de forma transparente para o programador. Outro ponto forte, assim como a linguagem Chestnut, é permitir a definição de variáveis sem a necessidade da alocação explícita. Essas características são desejáveis e favorecem o desenvolvimento.

A Linguagem Java

A linguagem escolhida como base para Fusion é a linguagem Java. Alguns fatores que levaram a essa escolha são o seu caráter orientado a objetos, a sua simplicidade e a sua ampla disseminação tanto no meio acadêmico quanto no industrial.

Outro ponto de interesse na linguagem Java é a possibilidade de construção de um compilador utilizando a ferramenta VMKit [Geoffray et al. 2010], que permite a personalização de uma maquina virtual adequada para a linguagem proposta, com base na JVM (*Java Virtual Machine*). Na Seção 5.1.2, a ferramenta VMKit será apresentada e sua utilização será justificada.

Dependência de CUDA

A sintaxe proposta para Fusion segue a sintaxe de Java, possuindo construções semelhantes, sendo introduzidas apenas novas estruturas para o desenvolvimento voltado para GPUs. Porém, para realizar a ligação entre o nó de processamento hospedeiro (*host*), possivelmente composto de múltiplos núcleos, e o dispositivo GPU (*device*), parte-se inicialmente da linguagem CUDA C. Portanto, a linguagem Fusion possui restrições quanto ao seu uso, ocorrendo somente sobre dispositivos ofertados pela empresa NVIDIA. Trabalhos futuros podem desenvolver uma comunicação específica sem a utilização da linguagem CUDA C como ponte de comunicação com o dispositivo, acrescentando portabilidade a linguagem.

4.1.3 Transparência sobre Detalhes Arquiteturais da GPU

Como já discutido, o nível de abstração e transparência a ser oferecido por Fusion é um dos principais desafios nesse trabalho, pois não se pode proporcionar desempenho sem que o programador, conhecedor da aplicação, otimize seu código de acordo com a arquitetura alvo e a natureza da aplicação.

Por exemplo, o armazenamento dos dados em regiões específicas de memória, como visto, garante uma melhor eficiência na execução do algoritmo. Portanto, o programador de um objeto em Fusion poderá de forma explícita indicar a localização dos dados na hierarquia de memória. Para isso, é necessário permitir acesso às memórias global, compartilhada, de texturas e de constantes. As três últimas são de fundamental importância e devem ser indicadas explicitamente. Basicamente, o programador tem a sua disposição mecanismos semelhantes à linguagem CUDA C para a programação de objetos aceleradores.

4.1.4 Programação Paralela Heterogênea

Uma importante tendência na área de CAD é a computação paralela heterogênea, na qual plataformas de computação paralela são equipadas com aceleradores computacionais de muitos núcleos (*many-core*), tais como GPUs e MICs (*Many Integrated Cores*)[2], em seus nós de processamento de memória distribuída, os quais individualmente são compostos por múltiplos núcleos (*multi-core*). Dentre os computadores listados na lista Top500 de novembro de 2012, 12,4% fazem uso de algum tipo de acelerador computacional, incluindo as GPUs, as quais dominam essa lista respondendo por 10,6% do total (53 máquinas). Porém, em relação ao total de desempenho bruto das máquinas, o impacto das GPUs chega 19,6%, por equiparem máquinas que estão mais ao topo da lista, o que evidencia serem uma tendência na arquitetura de clusters e MPPs (*Massive Parallel Processors*).

As múltiplas hierarquias de níveis de paralelismo nessas plataformas podem ser exploradas eficientemente usando modelos de programação adequados para cada nível. Por exemplo, recomendaria-se:

- MPI [Dongarra et al. 1996] para troca de mensagens entre processos distribuídos no nós de processamento;

- OpenMP [OpenMP Architecture Review Board 1997] para sincronização entre *threads* sobre os núcleos de processamento de um nó de processamento;

[2]O principal representante dessa classe é o recém-lançado Intel Xeo Phi, o qual já equipa 7 máquinas do Top500 em Novembro de 2012.

- CUDA, ou OpenCL, para explorar o paralelismo do acelerador computacional, se este for do tipo GPU.

Mesmo em computadores *desktop* ou servidores, vistos individualmente, a computação heterogênea se faz relevante, tendo em vista a proliferação, já amplamente disseminada, da tecnologia de processadores de múltiplos núcleos, que faz do processamento paralelo dentro do processador hospedeiro uma suposição importante que deve ser levada em conta em uma linguagem voltada a conexão entre uma computação nele realizada e a GPU. De fato, essa é a preocupação que leva à principal contribuição da linguagem Fusion, ou seja, ser uma ponte entre um processador hospedeiro de múltiplos núcleos (*multi-core*) e um dispositivo GPU, através da noção de kernel paralelo tornado possível pela tecnologia Hyper-Q da arquitetura Kepler.

4.2 Modelo de Programação de Fusion

Muitas aplicações exigem cálculos complexos com várias fontes de paralelismo, muitas vezes concorrentes, que podem ser representadas por objetos individuais na aplicação. Tais objetos podem realizar uma parte específica dessa aplicação ou podem representar um técnica especifica de computação.

Fusion parte do pressuposto de que computações cujo desempenho é crítico, por influenciar severamente o desempenho global de uma aplicação com requisitos de computação de alto desempenho, estão encapsuladas em objetos. Nesse caso, fica a cargo do programador determinar quais desses objetos podem ou devem ser acelerados por meio de um dispositivo acelerador, como uma GPU.

A idéia por trás do modelo de programação proposto por Fusion é o encapsulamento do paralelismo de determinadas regiões críticas da aplicação por meio da abstração de *objeto acelerador*, o qual é alocado sobre o dispositivo GPU e deve ser capaz de explorar o seu potencial massivo de paralelismo.

4.2.1 Objeto Acelerador

A arquitetura CUDA baseia seu paralelismo no nível de *threads*. Não diferente disso, o paralelismo de Fusion é mantido no mesmo nível, porém encapsulado em um **objeto acelerador**. Um objeto acelerador é uma abstração capaz de intermediar a comunicação entre um programa paralelo *multi-thread* que executa sobre um processador de múltiplos núcleos (*multi-core*), dito hospedeiro, e uma GPU, dito dispositivo, a fim de acelerar o desempenho de um programa. Trata-se

de um contexto híbrido, onde podem ser realizadas computações tanto no hospedeiro quanto no dispositivo.

Um objeto acelerador, em execução, é representado por um conjunto de *unidades*, cada qual alocada a uma *stream* independente em um dispositivo, o qual se supõe ser de arquitetura Kepler e suportar Hyper-Q[3]. De fato, unidades representam uma abstração para o conceito de *stream*, introduzido pela arquitetura Fermi para representar fluxos de instruções simultâneos que podem estar executando de forma intercalada na GPU, permitindo que múltiplos *kernels* possam ser disparados enquanto outros estão executando na GPU. Através do Hyper-Q, a arquitetura Kepler introduziu o suporte a *streams* independentes, que permitem que vários *kernels* possam estar executando paralelamente no dispositivo GPU.

A Figura 4.1 ilustra a ligação entre um programa paralelo que executa no processador hospedeiro, composto de múltiplas *threads*, a uma GPU, através de um objeto acelerador. Cada unidade permite o disparo independente de *kernels* por uma *thread*, dita hospedeira, através da *stream* a qual encontra-se associada. Os métodos de um objeto acelerador estão distribuídos em suas unidades, podendo ser *kernels* que serão executados pelo dispositivo GPU ou procedimentos comuns que serão executados pelo processador hospedeiro. No primeiro caso, são chamados métodos kernel (do inglês, *kernel methods*). No último caso, são chamados métodos host (do inglês, *host methods*). Existem ainda métodos que representam funções específicas que são chamadas diretamente no dispositivo, por métodos kernel, chamados de métodos device (do inglês, *device methods*). Em alguns casos, como esse, optamos por adotar os nomes nas língua inglesa, por simplicidade.

A linguagem Fusion tem como princípio permitir que o programador desenvolva seus métodos kernel de forma semelhante ao desenvolvimento de *kernels* na arquitetura CUDA, modelando de forma explícita o trabalho de cada *thread* no dispositivo GPU.

Alguns métodos de um objeto acelerador são ditos paralelos, possuindo implementação em várias unidades. Nesse caso, a chamada a um *método paralelo* deve ser realizada sobre todas as unidades que a implementam para que se complete, pelas *threads* hospedeiras que estão associadas às unidades. Métodos que não são paralelos são chamados métodos singulares.

[3]A dependência sobre a tecnologia Hyper-Q não é uma característica intrínseca de Fusion. A abstração de unidades é viável sobre qualquer arquitetura de dispositivo onde seja possível a execução simultânea de múltiplas linhas de instruções, representando *kernels* distintos, possivelmente disparados simultaneamente.

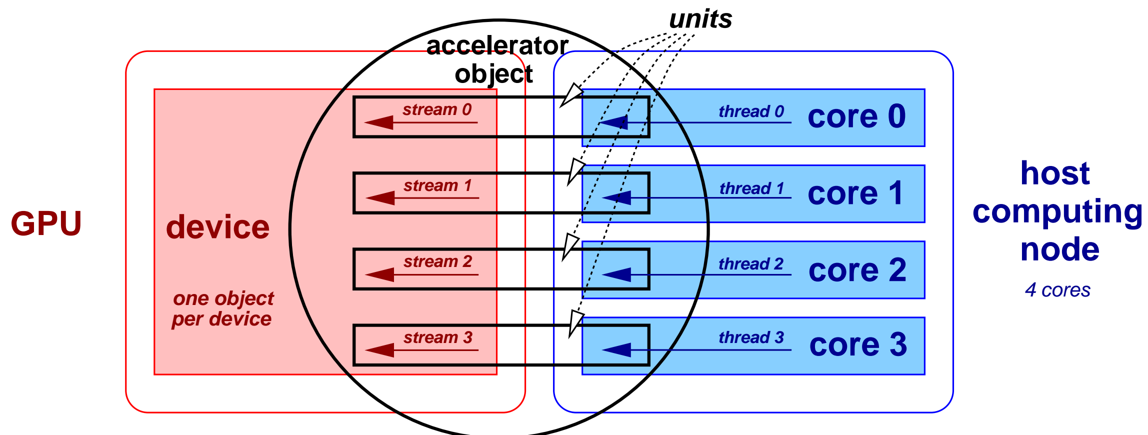


Figura 4.1: Ligação de um Programa Paralelo no Processador Hospedeiro a uma GPU usando um Objeto Acelerador

Um método kernel pode realizar chamadas a outros métodos kernel da mesma unidade, possibilitando a hierarquização do algoritmo diretamente sobre o dispositivo. Isso é possível através da tecnologia de paralelismo dinâmico (*dynamic parallelism*).

Além de permitir a ligação livre de gargalos de sincronização entre um programa paralelo *multi-thread* no processador hospedeiro e a GPU, essa representação de um objeto acelerador como um conjunto de unidades permite descrever computações do tipo MPMD (*Multiple Program Multiple Data*) sobre a GPU, uma vez que um método paralelo pode ter implementações distintas em cada unidade, representando papéis distintos que desempenham em uma computação paralela. De fato, poderíamos falar em MKMD (*Multiple Kernel Multiple Data*), quando o método paralelo representa *kernels* a serem executadas em paralelo na GPU. Uma discussão sobre unidades e *streams* é ainda apresentada na Seção 4.2.4

Objetos aceleradores podem ser usados em qualquer parte do código em um programa Fusion. Além disso, o modelo facilita a substituição de objetos aceleradores por objetos comuns, que não usam o dispositivo mas que implementem a mesma interface, e vice-versa.

Quanto ao estado do objeto, pode estar distribuído na memória global do dispositivo e na memória do hospedeiro, o que define quais dados são alcançáveis pelo conjunto de métodos kernel. A observação externa desse estado deve ser realizada por meio de métodos host, seguindo preocupações de sincronização para garantir a consistência dos dados.

4.2.2 Classe de Objetos Aceleradores

Uma classe de objetos aceleradores possui a mesma definição de classes na orientação a objetos, representando um conjunto de objetos aceleradores com características similares, nesse caso, um conjunto comum de unidades, métodos e atributos, diferenciando-se apenas por seus estados de execução. Um objeto acelerador é portanto uma instância de uma *classe aceleradora*.

O Código 4.1 apresenta um exemplo de classe aceleradora, a partir da qual são instanciados objetos que realizam certas operações aritméticas simples sobre um valor inteiro inicialmente fornecido. Esse código será usado ao logo dos próximos parágrafos para ilustrar aspectos sintáticos relevantes da linguagem Fusion.

Uma classe aceleradora declara um conjunto de unidades, bem como seus métodos e atributos, usando convenções sintáticas semelhantes as de Java. As unidades são identificadas através da palavra reservada `unit`. Caso seja paralela, precede-se a palavra reservada `unit` pelo qualificador `parallel`. Métodos, host ou kernel, podem ser declarados tanto no escopo de uma unidade quanto fora dos escopo de alguma delas. Nesse último caso, diz-se que estão declaradas no escopo da classe. Para distinguir métodos *host* de métodos *kernel*, utilizam-se os modificadores `host` e `kernel`, respectivamente. Caso não seja usado um dos dois modificadores, assume-se a declaração de método *host*.

Código 4.1: Exemplo - Classe Aceleradora

```
1  accelerator class FooAccelClass
2  {
3      private global int value;
4
5      FooAccelClass (int initial_value) { this.value = initial_value; }
6
7      /* A kernel method that is parallel across all the units.
8         It does not declare a default implementation.
9         It is synchronized for updating value concurrently without interference. */
10     public synchronized parallel kernel void perform () thread<<<nthreads>>>
11                                                          block<<<nblocks>>>;
12     /* A host method with a default implementation, both for Adders and Multipliers.
13        The use of the "host" modifier is optional */
14     public host int getValue() { return value; }
15
16     parallel unit Adder
17     {
18         shared int increment;
19
20         Adder (int increment) { this.increment = increment; }
21
22         // parallel method implementation for Adder
```

```
23          void perform () { value = value + increment; }
24
25          public synchronized parallel kernel undo()thread<<<nthreads>>>
26                                                block<<<nblocks>>>
27          { value = value - increment; }
28      }
29
30      parallel unit Multiplier
31      {
32          shared int factor;
33
34          Multiplier (int factor) { this.factor = factor; }
35
36          // parallel method implementation for Multiplier
37          void perform () { value = value * factor; }
38      }
39  }
```

Além do construtor do objeto acelerador, como em Java, cada unidade poderá também conter o seu próprio método construtor, cuja sintaxe de declaração segue as convenções da declaração de construtores de Java, ou seja, sem valor de retorno e com o nome da própria unidade. A principal utilização de um construtor de unidade é a inicialização do seu estado, como se faz com objetos Java.

Um método declarado no escopo de uma classe é um método que deve ter implementações específicas para cada uma das unidades declaradas na classe. Como forma de simplificar a programação no caso onde várias unidades compartilham uma mesma implementação, o método declarado no escopo da classe pode definir uma implementação, dita *default*, que pode ser sobrescrita pela versão específica de cada unidade. Caso contrário, não existindo implementação *default*, apenas a assinatura do método é especificada e sobrecarregada em cada unidade.

Um método paralelo, seja ele declarado no escopo da classe ou da unidade, deve ser sempre declarado com o modificador `parallel`. Caso contrário, trata-se de um método singular, mesmo que se trate de uma unidade paralela. Nesse caso, o método é singular em relação a cada instância da unidade paralela, durante a execução, não havendo necessidade de que seja invocado sobre todas as unidades, como acontece com métodos paralelos. Obviamente, métodos paralelos não fazem sentido em unidades singulares. Nesse caso, o modificador `parallel` não tem qualquer efeito, causando apenas uma advertência do compilador.

No exemplo do Código 4.1, há duas unidades paralelas, chamadas de `Adder` e `Multiplier`. Há dois métodos *kernel* paralelos: `perform`, entre todas as unidades da classe; e `undo`, apenas entre unidades `Adder`. A primeira, que é declarada no escopo

da classe, não apresenta uma implementação *default*, possuindo implementações diferentes para cada unidade. Por sua vez, o método *host* `getValue` define uma implementação *default*, válida para todas as unidades. Para unidades `Adder`, a operação `perform` adiciona um incremento a variável `value`, a qual é compartilhada entre as unidades e encontra-se na memória global do dispositivo, enquanto para unidades `Multiplier`, essa operação multiplica um fator a `value`. Ambos, fator e incremento, são fornecidos nos construtores de suas unidades, assim como o valor inicial de `value` é fornecido no construtor da classe. O método kernel paralelo `undo` desfaz um incremento realizado por uma invocação a `perform` na unidade `Adder`. É importante ressaltar o fato de que os métodos `perform` e `undo` são sincronizados (modificador `synchronized`), de modo que a atualização da variável `value` nas invocações concorrentes dessas operações não sofrem interferência. As chamadas a `perform`, por exemplo, são executadas em alguma ordem sequencial tão logo sejam realizadas em todas as unidades `Adder` e `Multiplier`.

Seguindo regras de escopo comuns de Java, o estado local de uma unidade, representado por seus atributos, só pode ser alterado pelos métodos da própria unidade. Logo, a implementação *default* de um método somente pode acessar atributos declarados no escopo da classe.

A declaração de um método kernel configura a sua topologia de grade, de forma a mapeá-lo para um *kernel* no dispositivo. Assim, a classe aceleradora pode conter configurações de grade diferentes em cada método, definidas pelo desenvolvedor do objeto, que assume-se possuir total conhecimento sobre as características do dispositivo alvo. As configurações de grades são detalhadas na próxima seção.

Por sua vez, métodos *device* devem ser definidos por meio do modificador `device`, podendo ser declarados no escopo de uma unidade ou da classe. Os métodos device são funções específicas e de uso restrito. São chamadas e executadas diretamente no dispositivo e não podem ser invocadas no processador hospedeiro. Ao contrário de métodos kernel, não requerem a configuração de uma grade.

Os modificadores de visibilidade de métodos em Java (`private`, `protected` e `public`) também aplicam-se normalmente a métodos *kernel* e *host*. Assim, um método *kernel* pode ser público, quando pode ser chamado diretamente pela *thread* do processador hospedeiro que possui a unidade em que está definida, ou privada, quando somente pode ser invocado por um método, *host* ou *kernel*, do próprio objeto. Como já discutido, a chamada de um método kernel a partir de um outro método kernel só é possível com o suporte ao paralelismo dinâmico. No Código 4.1,

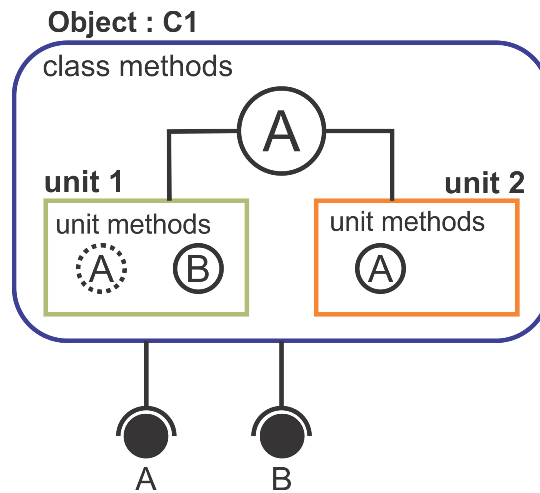


Figura 4.2: Composição de um objeto paralelo

todos os métodos declarados são públicos, e não há chamada de um método kernel a outro.

Na Figura 4.2, podemos observar a composição de um objeto acelerador instanciado a partir de uma classe aceleradora chamada C1. Nesse caso, temos um objeto composto por duas unidades paralelas. Cada unidade paralela será instanciada em um conjunto homogêneo de unidades durante a execução. A classe C1 define um método kernel em seu escopo, chamado **A**. Esse método é implementado pelas unidades 1 e 2. A unidade 1, em seu escopo, faz uma reescrita sobre o método **A**, representada pelo círculo de linha tracejada, e define um novo método kernel chamado de *B*. A unidade 2 utiliza a implementação padrão do método **A**, representada pelo circulo de linha contínua dentro do escopo da unidade. Portanto, o objeto em questão disponibiliza dois métodos acelerados que podem ser chamados em qualquer parte do código fonte da aplicação. O método *B* consiste de um método de unidade, enquanto o método *A* é considerado um método de classe.

4.2.3 Configuração de Grade

A topologia da grade de um método *kernel* é construída indicando-se o número de blocos que ela deve possuir e o número de *threads* que cada bloco terá a sua disposição. Pode ser constituída de duas ou três dimensões, o que depende da versão do dispositivo disponível.

As informações sobre a topologia da grade de um método *kernel* vem logo após a assinatura desse método, através das cláusulas `threads` e `blocks`.

A quantidade de blocos de uma grade é informada através do comando `blocks`

$\lll b_x, b_y, b_z \ggg$, onde b_x representa número de blocos no eixo X, analogamente para os eixos Y (b_y) e Z (b_z). Dessa forma, se uma grade possui duas dimensões, somente os eixos X e Y serão preenchidos. Por sua vez, o número de *threads* em um bloco é configurado através do comando `threads` $\lll t_x, t_y, t_z \ggg$, onde t_x, t_y e t_z denotam as quantidades de *threads* em cada dimensão.

Para verificar o número de *threads* ou blocos de uma determinada dimensão, basta indicar o eixo. Por exemplo, para verificar o número de blocos no eixo X, utiliza-se o comando `numBlocks(x)`, enquanto que utiliza-se o comando `numThreads(x)` para verificar o número de *threads* nesse mesmo eixo.

Analogamente à linguagem CUDA C, existem comandos que permitem a uma *thread* de um *kernel* verificar sua posição na topologia da grade. O comando `threadIdx(`*axis*`)`, onde *axis* representa o eixo, podendo ser `x`, `y`, ou `z`, retorna o número da *thread* dentro de um bloco. Por sua vez, `blockIdx(`*axis*`)` indica o bloco ao qual a *thread* pertence. Finalmente, os comandos `gridSize(`*axis*`)` e `blockSize(`*axis*`)` retornam os tamanhos da grade e do bloco no eixo indicado, no qual a *thread* está inserida.

A Figura 4.3 ilustra uma grade configurada com `blocks` $\lll 3, 2, 3 \ggg$ e `threads` $\lll 4, 4, 4 \ggg$, perfazendo um total de 1152 *threads*.

Figura 4.3: Mapeamento de uma unidade paralela para GPU

4.2.4 Unidades e *Streams*

Unidades constituem uma importante abstração introduzida por Fusion, motivada pela execução de múltiplos *kernels* concorrentes em uma GPU. Como

discutido anteriormente, todo trabalho ou *kernel* é enviado para o dispositivo GPU através de um *stream*, representando um fluxo de operações a serem executadas pelo dispositivo, implícita ou explicitamente. Esse *stream* representa uma ligação entre o hospedeiro e o dispositivo. Uma unidade é, portanto, uma abstração para o conceito de *streams* da arquitetura CUDA.

Por padrão, apenas um *stream* é criado implicitamente em CUDA, sobre o qual os *kernels* da aplicação são enfileirados para execução. Porém, é possível a criação de vários *streams* através da tecnologia Hyper-Q, que possibilita a execução de vários *streams* concorrentemente sobre o mesmo dispositivo em filas de execução separadas. Antes dessa nova tecnologia, os *streams* disputavam em apenas uma fila de execução, gerando a serialização de *kernels* mesmo em *streams* diferentes.

Na arquitetura CUDA, o número de *streams* é determinado antes do lançamento dos *kernels*. Após sua criação, um *stream* pode ser referenciado no construtor de um *kernel*, de forma que será associado a um *stream* específico ao ser disparado.

Em Fusion, cada unidade, por padrão, é associada a um único *stream*, garantindo que seus métodos *kernel* possam manter um fluxo de operações, determinado pelo programador do objeto. Dessa forma, diferentes unidades podem ser executadas em paralelo, utilizando *streams* independentes de um mesmo dispositivo e executar suas operações em paralelo, aumentando o desempenho da aplicação. Na Figura 4.1, podemos observar como é realizado o mapeamento das unidades de um objeto acelerador no dispositivo GPU.

Código 4.2: Exemplo - Instanciação de Objeto Acelerador

```
1  abstract class HostThread extends Runnable
2  {
3      protected FooAccelClass gpu_obj;
4      protected int size, rank;
5      protected int[] val;
6
7      HostThread (FooAccelClass gpu_obj, int size, int rank, int[] val)
8      {
9          super();
10         this.gpu_obj = gpu_obj;
11         this.size = size;
12         this.rank = rank;
13         this.val = val;
14     }
15 }
16
17 class AdderHostThread extends HostThread
18 {
19     @Override
```

```
20      protected void run()
21      {
22          FooAccelClass::Adder gpu_obj_add = unit Adder<size>(10) of gpu_obj;
23          gpu_obj_add.perform();
24          val[rank] = gpu_obj_add.getValue();
25          gpu_obj_add.undo();
26      }
27  }
28
29  class MultiplierHostThread extends HostThread
30  {
31      @Override
32      protected void run()
33      {
34          FooAccelClass::Multiplier gpu_obj_mul = unit Multiplier<size>(5) of gpu_obj;
35          gpu_obj_mul.perform();
36          val[rank] = gpu_obj_mul.getValue();
37      }
38  }
39
40  class Main
41  {
42     public static void main(String[] args)
43     {
44        int cores = Runtime.getRuntime().availableProcessors();
45        int[] val = new int[cores];
46        private final ForkJoinPool forkJoinPool = new ForkJoinPool(cores);
47        int device = 0;
48        int value = 5;
49        FooAccelClass gpu_obj = new FooAccelClass<device>(value);
50        for (int i=0;i<cores;i++)
51        {
52           if (i % 2 == 0)
53              forkJoinPool.submit(new AdderHostThread(gpu_obj, cores/2, i, val));
54           else
55              forkJoinPool.submit(new MutiplierHostThread(gpu_obj, cores/2, i, val));
56        }
57        forkJoinPool.shutdown();
58        forkJoinPool.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);
59     }
60  }
```

4.2.5 Instanciação de Objetos Aceleradores

Um objeto acelerador é instanciado de maneira semelhante a um objeto comum Java. Portanto, um objeto é criado através do comando `new` a partir de uma classe específica fornecida.

Considere o exemplo no Código 4.2, onde um objeto `gpu_obj`, da classe `FooAccelClass` do Código 4.1 é instanciado na linha 49, no dispositivo identificado pelo inteiro 0, e utilizado por um conjunto de *threads* hospedeiras instanciadas

no laço entre as linhas 50 e 56 a partir das subclasses AdderHostThread e MutliplierHostThread da classe HostThread.

A diferença em relação a instanciação de um objeto Java comum está na presença da especificação opcional do dispositivo onde o objeto será instanciado, representado pela variável device. Supõe-se que os dispositivos são numerados de 0 a $n-1$, onde n é o número de dispositivos conectados ao computador hospedeiro. Caso não seja fornecido o valor inteiro que identifica o dispositivo, o objeto paralelo será instanciado em um dispositivo automaticamente determinado em tempo de execução. Os argumentos ao construtor do objeto são próprios da classe.

Uma vez que a utilização das unidades de um objeto acelerador ocorre através de um conjunto de *threads* no processador hospedeiro, cada uma é responsável por instanciar uma unidade, que pode ser referente a uma unidade singular ou a uma unidade paralela da classe aceleradora, de modo que uma unidade paralela será instanciada em um time homogêneo de unidades, com relação aos seus atributos e métodos, cada qual inicializada por uma *thread* distinta. Para isso, a declaração unit deve ser executada pela *thread* hospedeira.

Por exemplo, no Código 4.2, as *threads* AdderHostThread inicializarão a unidade Adder de gpu_obj, enquanto que as *threads* MultiplierHostThread inicializarão a sua unidade Multiplier. Essas inicializações são realizadas respectivamente nas linhas 22 e 34, através da declaração unit, informando obrigatoriamente o número de unidades que constituem a unidade paralela em execução entre os delimitadores angulares. Todas as unidades da unidade paralela devem declarar o mesmo valor. Após essas declarações, os métodos das unidades Adder e Multiplier, respectivamente, podem ser invocados sobre as variáveis gpu_obj_add e gpu_obj_mul, como qualquer método de um objeto Java.

O número de *threads* dedicadas para o objeto acelerador, bem como o número de unidades de cada unidade paralela, é definido pelo usuário especialista, sendo de sua total responsabilidade a inicialização adequada das unidades do objeto acelerador. No caso de uma unidade singular, o número de unidades é obviamente desnecessário, causando um erro de compilação, uma vez que uma única unidade será instanciada.

A invocação de um método paralelo de um objeto acelerador previamente instanciado só poderá completar-se quando todas as unidades envolvidas, ou seja, que possuem implementação desse método, estiverem devidamente instanciadas, de modo que podem ter seus métodos invocados. Enquanto isso, permanecerá bloqueada. É para essa finalidade, isto é, a detecção de quando todas as unidades

de uma unidade paralela foram instanciadas, que o número de unidades da unidade paralela torna-se essencial. Portanto, quando todas as suas unidades forem instanciadas, o objeto acelerador estará apto para execução através das invocações dos métodos da unidade inicializada pela *thread* hospedeira. Essa é uma informação que somente o usuário especialista pode definir, tendo em vista que somente ele tem conhecimento do número de *threads* utilizadas por sua aplicação.

Ainda no exemplo Código 4.2, os métodos `perform` e `undo` são invocados logo após a inicialização das unidades no método `run`, que define o comportamento das *threads* hospedeiras. Note que o método `undo` somente é invocado pelas `threads AdderHostThread`. De fato, esse método não é acessível pela variável `gpu_obj` das *threads* `MultiplierHostThread`.

Cuidados especiais devem ser garantidos sobre o sistema de tipos herdado de Java para garantir a segurança das operações sobre a variável do objeto acelerador. Após a declaração `unit`, note que o tipo da variável que referencia o objeto acelerador (e. g. `gpu_obj_add`) passa a ser definido pela classe aceleradora e pela unidade (e. g. `FooAccelClass:Adder` ou `FooAccelClass:Mutiplier`), que é propositalmente incompatível com o tipo representado somente pela classe aceleradora (e. g. `FooAccelClass`). Dessa forma, caso haja necessidade de passar um objeto acelerador entre objetos e métodos na *thread* hospedeira, o tipo da variável que receberá o objeto acelerador, caso já tenha sido inicializado para alguma unidade, deve ser do tipo $ClassName{:}UnitName$, ao invés de $ClassName$, pois forçaria um erro de tipos estaticamente detectável, onde $ClassName$ representa o nome da classe aceleradora e $UnitName$ o nome de uma de suas unidades.

4.2.6 Hierarquias de Memória

A hierarquia de memória em Fusion segue o modelo da arquitetura CUDA. O desenvolvedor do objeto poderá utilizar normalmente os níveis de memória usuais: *global*, *compartilhada*, *de constantes* e *de texturas*, através dos modificadores `global`, `shared`, `constant` e `texture` na declaração de variáveis, respectivamente. A alocação de variáveis é realizada de maneira transparente. Com as informações da declaração, o compilador se encarrega de criar os devidos comandos para PTX necessários para realizar a alocação da variável na memória do dispositivo.

É importante ressaltar que somente variáveis de tipos primitivos e arrays desses tipos podem ter seus valores alocados sobre o dispositivo, ou seja, podem ser declaradas com os modificadores acima citados. Não é permitido que objetos sejam

alocados no dispositivo.

Variáveis declaradas no escopo da classe são sempre alocadas na memória global do dispositivo, com o modificador `global`. Caso contrário, são alocadas na memória do processador hospedeiro. Por sua vez, no escopo de unidades ou em um método *kernel*, variáveis nos níveis global, compartilhado, texturas e constantes podem ser definidas, sendo que o não uso de um modificador causa a alocação da variável na memória global do dispositivo. A seguir, um exemplo de declaração de variável na memória compartilhada:

```
shared int [][] matrix = new int [size][size];
```

Nesse exemplo, uma matriz de inteiros com dimensão $size \times size$ será alocada na memória compartilhada do dispositivo GPU.

A memória do dispositivo GPU é persistente, e não está sob o alcance do coletor de lixo da máquina Java. Decidimos, nessa primeira versão de Fusion, tornar explícito o gerenciamento da memória do dispositivo, o que implica incluir operações para liberação de valores de variáveis. De fato, a liberação dos espaços utilizados pelo objeto acelerador sempre que uma variável se torna inacessível é conveniente, tendo em vista a limitação de memória dos dispositivos GPU e a possibilidade de vários objetos aceleradores a compartilharem. Esse é o motivo pelo qual optamos por não tratar nessa dissertação a respeito do problema de gerenciamento dinâmico da memória de dispositivos GPU, entendendo ser um problema complexo que merece um tratamento de um projeto específico.

Assim, Fusion suporta a operação `clear(`*var*`,` *value*`)`, onde *var* indica uma variável específica cujo valor será liberado da memória e *value* o valor que essa variável vai assumir após a liberação (e.g. `null`).

4.2.7 Comunicação

Dentre as restrições do modelo de programação sobre GPUs, a comunicação entre *threads* de um mesmo *kernel* e entre *kernels* é uma das mais marcantes. De fato, o modelo intimida o programador a pensar em termos de sincronização e trocas de dados explícitas entre as *threads*. Quando isso é necessário, deve ser tratado com muito cuidado, pois o mal uso de sincronização entre as *threads* pode influenciar severamente no desempenho dos programas sobre esses dispositivos.

Comunicação Vertical, entre Métodos Kernel de uma Mesma Unidade

Em um objeto paralelo da linguagem Fusion, a comunicação entre dois kernels distintos de um mesma unidade é chamada de *comunicação vertical*, e pode ser realizada de duas maneiras:

- **Memória do Hospedeiro**: dados na memória do hospedeiro acessados por um método *kernel* são copiados do hospedeiro para o dispositivo antes da execução de seu corpo e do dispositivo para o hospedeiro após o seu retorno, quando suas alterações sobre os dados tornam-se visíveis para outros métodos, kernel e host;

- **Memória Global do Dispositivo**: os dados na memória global do dispositivo, de variáveis declaradas no escopo de unidades, mantêm-se persistentes entre chamadas de métodos kernel, bastando o acesso correto por outros métodos kernel, sendo desnecessário transferências entre o hospedeiro e o dispositivo.

Comunicação Horizontal, na Invocação de um Método Kernel Paralelo

A comunicação entre *kernels* paralelos, executando em *streams* diferentes, na invocação de um método *kernel* paralelo é chamada de *comunicação horizontal*. Tal comunicação pode ser realizada através da memória global do dispositivo, por intermédio de variáveis declaradas com o modificador `global` no escopo da classe.

A comunicação através da memória global deve ser realizada com cuidado, pois dois métodos kernel não podem manter a sincronização através de paralelismo dinâmico, como *kernels* pai e filho, no caso de métodos recursivos ou que realizam chamadas a outros métodos, em uma mesma unidade paralela. Nesse caso, a sincronização pode ser realizada através de comandos específicos de sincronização.

4.2.8 Sincronização

Caso seja necessária a execução síncrona de dois métodos kernel de unidades diferentes, é necessário realizar a comunicação através do hospedeiro. Tarefa simples mas que pode comprometer o desempenho da aplicação, devido as transferências de dados entre hospedeiro e dispositivo, discutidas anteriormente. Dispomos basicamente de quatro opções operações de sincronização:

- synchronized(waitMethod(id_method)) recebe um método de uma unidade paralela como parâmetro (id_method), enquanto esse método estiver

executando o hospedeiro fica aguardando, deve ser usado somente entre hospedeiro e dispositivo;

- synchronized(waitEvent()) recebe um método de uma unidade paralela como parâmetro, enquanto esse método estiver executando o hospedeiro ou a *thread* lançadora fica aguardando;
- synchronized(device(id_device)) o hospedeiro ou a *thread* lançadora espera até que todos os métodos lançados em um determinado dispositivo (id_device) terminem sua execução, independente do tipo da unidade;
- synchronized(threads()) todas as *threads* de um bloco são sincronizadas.

Algumas observações são relevantes com relação a sincronia de métodos *kernel* lançados a partir de outros métodos *kernel* diretamente no dispositivo utilizando a tecnologia de paralelismo dinâmico. Uma *thread* de um método *kernel* só poderá realizar a sincronização com outros métodos *kernel* lançados no mesmo bloco de *threads* em que ela se encontra, utilizando o synchronized(waitMethod(id_method)).

Métodos kernel lançados em um mesmo bloco de *threads* são implicitamente sincronizados com todas as *threads* do bloco até o final de sua execução. A sincronização com métodos *kernel* fora do bloco de *threads* corrente é indefinida devido a questões arquiteturais. Segundo os engenheiros David Kirk e Wen-mei W. Hwu, da NVIDIA, não é garantido que um *stream* seja única entre os diferentes blocos. Portando, seu uso em um bloco que não o alocou pode resultar em um comportamento indefinido [Kirk e Hwu 2010].

Capítulo 5

Implementação e Estudos de Caso

O assunto tratado neste capítulo engloba dois aspectos importantes para avaliar a viabilidade da linguagem Fusion no uso prático: a arquitetura de um compilador e a implementação de estudos de caso com programas reais de interesse de aplicações de computação de alto desempenho.

O compilador proposto para a linguagem Fusion é baseado no compilador LLVM (*Low Level Virtual Machine* [Illinois 2012]). Mais especificamente, utiliza uma máquina virtual Java (JVM) personalizada através de uma ferramenta que permite a construção de ambientes de execução virtual customizados.

O compilador Fusion tem sido idealizado como uma ferramenta capaz de gerar código específico para dispositivos GPUs a partir do código Fusion. Porém, o compilador encontra-se ainda em prototipagem. Em sua primeira versão, busca-se uma tradução direta das classes aceleradoras do código Fusion para implementação correspondente em CUDA C, onde os métodos kernel são diretamente traduzidos para *kernels* CUDA.

Com os estudos de caso, busca-se demonstrar a utilização das abstrações propostas pela linguagem Fusion. São apresentados dois estudos de caso. O primeiro consiste de um algoritmo simples para multiplicação de matrizes. O segundo estudo de caso consiste de uma aplicação onde os autores propõem uma nova técnica para o problema do Caixeiro Viajante Assimétrico sobre GPGPU [Pessoa et al. 2011].

5.1 Arquitetura do Compilador

O primeiro protótipo do compilador Fusion será baseado no compilador LLVM, que busca otimizar em tempo de compilação, ligação e execução programas escritos em diferentes linguagens de programação. Sua arquitetura é independente da

linguagem, e atualmente permite a ligação com várias linguagens de programação, tais como Java, Python, Fortran, etc. Um fato importante é que o compilador da arquitetura CUDA é baseado no compilador LLVM, o que constitui mais um motivo para reforçar a sua utilização.

5.1.1 Compilador LLVM

O LLVM é um compilador modular e de código aberto. Ele permite que códigos escritos em diferentes linguagens de programação sejam traduzidos, através de um compilador de entrada (*front-end*) adequado para a linguagem alvo, para uma representação intermediária chamada de LLVM IR, a qual é otimizada através de compiladores ou módulos intermediários. Um compilador de saída (*back-end*) é responsável por transformar o código LLVM IR na linguagem de máquina de diferentes arquiteturas.

No contexto deste trabalho, o código alvo é o código PTX (*Parallel Thread eXecution*). O PTX é um código semelhante a um código *assembly*, no qual é descrita a execução de cada *thread*, utilizando uma estrutura hierárquica para abstrair o *hardware* GPU. Um conjunto de instruções do PTX é estendido por instruções específicas GPU, tais como instruções de sincronização e suporte para endereçamento de memória [PTX3.1 2012].

O LLVM IR é semelhante ao PTX, também de baixo nível, embora com seus conjuntos de instruções próprios. Porém, existem diferenças significativas. O LLVM utiliza um formato específico diferente do PTX, chamado de SSA (*Static Single Assignment*). O SSA provê ao LLVM segurança de tipos, operações de baixo nível e a expressividade para representar virtualmente qualquer linguagem de alto nível.

Na Seção 2.6, foi possível ter uma breve introdução ao compilador CUDA, chamado de *nvcc*. Basicamente, existem duas fases de compilação. Primeiramente, o compilador *nvcc* traduz o código CUDA C para código PTX e finalmente para código de máquina específico para cada arquitetura GPU. O compilador *nvcc* utiliza o compilador LLVM para geração de código PTX devido às similaridades entre o código LLVM IR e o código PTX. As otimizações realizadas no código intermediário e no PTX não são reveladas pela NVIDIA.

Nosso interesse no LLVM é voltado para um projeto associado, chamado de VMKit. O VMKit provê uma ferramenta para construção de máquinas virtuais personalizadas. Mais detalhes desse projeto serão descritos na próxima seção.

O compilador LLVM é um sistema que pode ser estendido com novas funções,

tipos e instruções. Adicionar novas instruções pode se tornar algo complexo, pois é necessário estender todos os passos de otimização que o usuário pretende utilizar com a sua extensão. No entanto, se o componente a ser inserido pode ser expresso na forma de uma chamada de função, essa nova extensão pode ser inserida através de *funções intrínsecas* [Illinois 2013].

Uma função intrínseca é o principal mecanismo para extensão do compilador LLVM, pois é transparente para os passos de otimização e muito mais simples de ser implementada do que uma nova instrução. Funções intrínsecas devem sempre começar pelo prefixo llvm., reservado para esse tipo de função. Uma função intrínseca é uma função externa que realiza chamadas ou invoca instruções LLVM. Para adicionar novas funções intrínsecas ao compilador LLVM, é necessário seguir alguns passos [Illinois 2013]:

- Documentar a função intrínseca, definindo se o código é um gerador específico e quais as suas restrições;
- Adicionar uma nova entrada para a função intrínseca, descrevendo suas principais características de acesso a memória e otimizações;
- Suporte a chamadas, uma vez que a função intrínseca pode ser chamada um número constante de vezes, mas é necessário adicionar suporte nas funções $canConstantFoldCallTO$ e $ConstantFoldCall$;
- Adicionar casos de teste para o compilador.

5.1.2 Projeto VMKit

O VMKit é definido como um substrato para construção de MREs (*Managed Runtime Environments*). A utilização de MREs vem crescendo cada vez mais para aplicações em geral. Exemplos de MREs são a JVM (*Java Virtual Machine*) e a CLR (*Common Language Runtime*) [Hamilton 2003], que tem aplicação desde servidores *web* até sistemas embarcados. Um MRE executa uma representação intermediária de uma aplicação e caracteriza-se por ser compacto [Geoffray et al. 2010].

Apesar de ter por premissa suportar várias linguagens, o desenvolvimento de um MRE é algo complexo que exige definições de projeto específicas para cada linguagem alvo, como o coletor de lixo adotado, por exemplo.

Atualmente, existe um MRE baseado em JVM desenvolvido com VMKit, chamado de J3. Esse projeto é de nosso interesse, pois podemos utilizá-lo como

base para nosso compilador. Assim, todo código escrito em Fusion pode ser transformado em código intermediário LLVM e então para PTX, que será utilizado pela plataforma CUDA para executar sobre os dispositivos disponíveis. De fato, a maioria das extensões sobre Java serão realizadas através de funções intrínsecas sobre o compilador JIT do J3.

O VMKit torna-se interessante por não impor nenhum modelo de objetos, ou seja, funciona como o núcleo de um MRE fornecendo funcionalidades básicas, tais como *threads*, gerenciamento dinâmico de memória e um compilador JIT (*Just-In-Time*) independente da linguagem intermediária.

As desvantagens para escolha do VMKit são as dependências de outros projetos em desenvolvimento, como o projeto LLVM, para fornecer um compilador JIT, o projeto MMTK, para o gerenciamento dinâmico de memória, e o POSIX Threads (PThreads), para o gerenciamento de *threads*. Outro ponto importante é o fato do J3 ainda não possuir alguns mecanismos de otimização, tornando-o de 2 a 3 vezes mais lento que MREs de uso industrial [Geoffray et al. 2010].

Enfim, a arquitetura proposta para o compilador Fusion é baseada na máquina virtual J3. Máquinas virtuais diferentes não podem ser utilizadas, tais como a Oracle JVM ou a IBM JVM, de uso industrial. O compilador deverá receber um código escrito na linguagem Fusion, que é uma extensão de Java, e retornar um código LLVM IR, o qual é transformado em código PTX e repassado para o compilador CUDA, que deverá construir o código de maquina específico para o modelo de dispositivo disponível.

Em sua primeira versão, o compilador Fusion/J3 estenderá a máquina virtual J3 para gerar dois códigos. Um código Java puro com chamadas para as funções definidas no segundo código, gerado para a arquitetura CUDA. O código CUDA C será responsável por realizar a ligação da aplicação ao dispositivo GPU. O fluxo de compilação para essa primeira versão é apresentado na Figura 5.1. O código CUDA C tem como base as classes aceleradoras que compõem a aplicação, tendo em vista que toda implementação específica para o dispositivo é desenvolvida nessas classes.

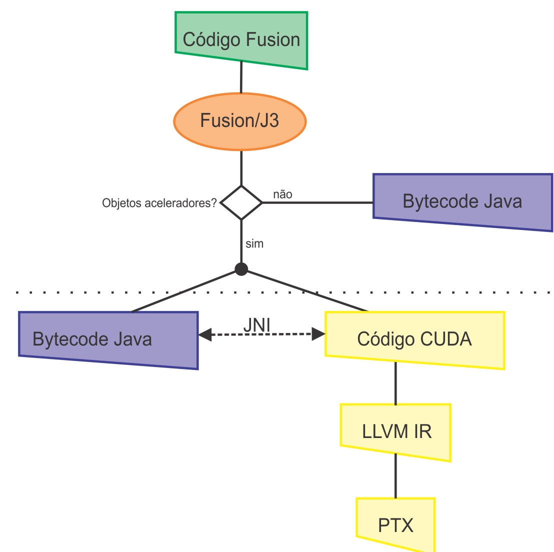


Figura 5.1: Fluxo de operações para o compilador Fusion/J3

Futuramente, propõe-se estender o compilador Fusion/J3 com a utilização do compilador de saída NVPTX. O NVPTX possui um conjunto de instruções pré-definidas que constrói o código PTX. Dessa forma, evita-se o passo de transformação para o código escrito em CUDA C, gerando diretamente um código PTX correspondente ao código Fusion.

5.1.3 Versão inicial e funções

Em sua versão inicial, a principal função do compilador Fusion/J3 será o reconhecimento das extensões sintáticas de Fusion sobre Java. Essas extensões estão intimamente ligadas à arquitetura CUDA, tendo em vista que objetos aceleradores Fusion são a ligação da aplicação aos dispositivos GPU.

Como vimos, o compilador é responsável por gerar dois novos códigos com base no código Fusion. A comunicação entre o código Java e o código CUDA C ocorre através de chamadas JNI. No código Java, um método kernel possui apenas a chamada para função externa que possui a implementação do método em questão, no código CUDA C. Na Figura 5.2, é possível ter uma visão simplificada do modelo inicial, onde é simulado a criação dos arquivos gerados pelo compilador Fusion/J3.

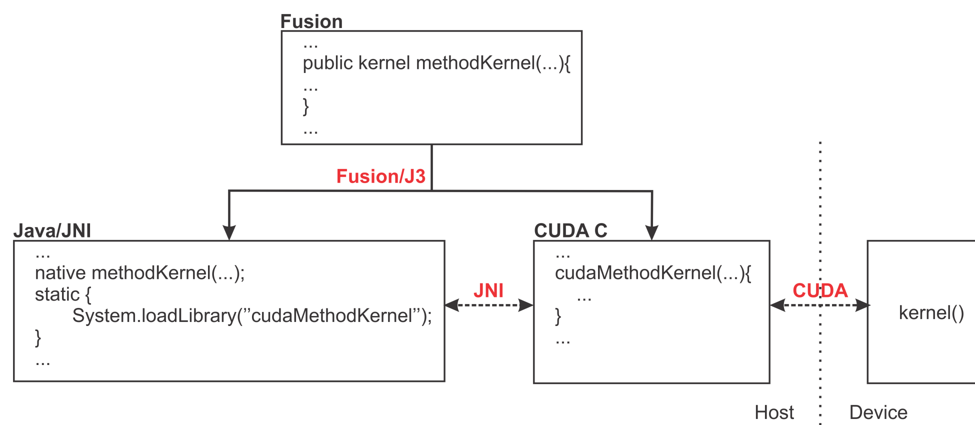


Figura 5.2: Ligação do código Fusion ao dispositivo GPU

A implementação de cada método kernel definida no código Fusion é traduzida em um código CUDA C, de forma que cada classe aceleradora deverá possuir um código correspondente nessa linguagem. O código Java resultante conterá somente a chamada para as funções externas no código CUDA C que deverão disparar os *kernels* no dispositivo GPU através do compilador CUDA.

Essa arquitetura deverá resultar em um acréscimo no tempo de compilação, devido às fases necessárias para geração dos códigos resultantes, e em tempo de execução, por serem realizadas chamadas do código Java para o código CUDA C para só então o *kernel* ser disparado no dispositivo. Além disso, há o tempo de comunicação entre as chamadas.

As principais responsabilidades do compilador Fusion/J3 são:

- Gerenciamento dos objetos aceleradores criados para aplicação;
- Construção e controle das unidades dos objetos aceleradores;
- Criação e gerenciamento das variáveis à serem alocadas no dispositivo;
- Geração do código Java puro com chamadas JNI para funções em CUDA C;
- Geração da versão CUDA C do código Fusion, com base nas classes aceleradoras.

Como visto, é necessário realizar diversas operações explícitas na linguagem CUDA C que são abstraídas por Fusion. O primeiro ponto a ser analisado corresponde a abstração proposta pelas unidades em Fusion. Uma unidade deverá

ser associada a uma *stream* no código CUDA C correspondente, que é construída da seguinte maneira, onde `stream` corresponde a variável que representa a *stream*:

```
cudaStream_t stream;
cudaStreamCreate(&stream);
```

Em Fusion, quando inicializamos uma nova unidade paralela *id_unit* de um objeto acelerador *accel_obj*, usando a declaração "`unit` *id_unit* <*size*> `of` *accel_obj*", onde *size* denota a cardinalidade da unidade paralela, o compilador deverá realizar a criação de uma *stream* e associá-la à unidade que está sendo criada, armazenando essa informação em uma tabela de símbolos. O parâmetro *size* permite que o compilador conheça o número de unidades do mesmo tipo que deverão ser inicializadas, o que permite que controle quando o disparo dos métodos dessas unidades será permitido, ou seja, quando todas as *size* unidades tiverem sido inicializadas.

Os métodos kernel são construídos como *kernels* CUDA, o qual corresponde a uma função global executada por todas as *threads* disparadas no dispositivo GPU sobre seus processadores de *stream*. O *kernel* recebe todos os argumentos necessários para computação da solução, ou parte dela.

Em Fusion, diferenciando-se da arquitetura CUDA, todas as variáveis alcançáveis por um método *kernel* deverão ser associadas a ele, mesmo que esse método não as recebam como argumentos. Uma pré-compilação deverá ser realizada para identificar todas as variáveis alcançáveis, assim quando um *kernel* é construído em seu correspondente CUDA C será possível passar todos os argumentos necessários para sua correta avaliação.

Outro ponto importante é que o gerenciamento de variáveis em CUDA C é realizado explicitamente, desde a alocação dos espaços de memória no dispositivo até a transferência dos dados para essas regiões, como no exemplo abaixo:

```
int variavel_d;
int variavel_h;
...
cudaMalloc((void **) &variavel_d, tam*sizeof(int));
cudaMemcpy(variavel_d, variavel_h, tam * sizeof(int),
cudaMemcpyHostToDevice);
```

A linguagem Fusion busca reduzir a complexidade e o trabalho manual associado

a esse gerenciamento. A alocação dos dados no dispositivo é construída somente para variáveis declaradas com o qualificador de memória global. Uma das responsabilidades do compilador é forçar que somente variáveis de tipos primitivos e vetores desses tipos podem ser instanciadas com o qualificador global, e não objetos em geral.

As cópias de dados entre o dispositivo e o processador hospedeiro são realizadas utilizando os operadores de atribuição tradicionais de Java, entre variáveis localizadas na memória global do dispositivo e na memória do processador hospedeiro. O uso do modificador `async` antes de uma atribuição define a transferência como assíncrona, tal como em "`async` x = y;", onde x e y são variáveis localizadas em níveis diferentes da hierarquia de memória.

Alternativamente, o usuário desenvolvedor tem disponível a função `memCpy(`*dest*, *init_d*, *orig*, *init_h*, *size*`);`, onde *dest* indica o destino dos dados, *init_d* indica a posição inicial no caso de um vetor, *orig* indica a origem dos dados, *init_h* indica a posição inicial na origem e *size* indica a quantidade de dados que serão transmitidos. Os argumentos *init_d*, *init_h* e *size* são úteis em transmissões parciais sobre o espaço de dados, nas quais a variável de destino recebe apenas parte dos dados da origem.

```
global int var_device = new [10];
int var_host = new [10];
...
memCpy(var_device, var_host,10);
```

As posições iniciais de cada variável foram omitidas nesse caso, onde a variável no dispositivo (`var_device`) recebe 10 posições a partir de sua posição inicial os dados da variável no hospedeiro (`var_host`) partindo também de sua posição inicial. Existe ainda a opção de cópia assíncrona obtida com a função `memCpyA(`*args*`)` que segue as mesmas definições da função `memCpy`.

5.2 Primeiro Estudo de Caso: Multiplicação de Matrizes

Nesta seção, é utilizada uma aplicação simples de multiplicação de matrizes *multi-thread*, a partir do qual deseja-se acelerar a computação realizada pelas *threads*, agora dita hospedeiras, as quais calculam individualmente resultados parciais da multiplicação de matrizes, através de uma GPU. Os Algoritmos 1 e 2 serão utilizados como base para esse estudo de caso.

Dadas duas matrizes A e B, deseja-se calcular o produto entre matrizes C = A×B

Algoritmo 1: Multiplicação de matrizes *multi-thread* - Disparo e inicialização das threads

Data: Dimensões das matrizes A e B
Result: Matriz resultante
1 Recebe dimensões de A e B, onde $numLinhasA$ é o número de linhas de A;
2 Preenche A e B;
3 Identifica o número de núcleos em $numNucleos$;
4 $nLinhas = numLinhasA/numNucleos$;
5 \\ *Número de linhas que cada thread computará*
6 $numThread = numNucleos$;
7 **for** $i = 0 \rightarrow nThread$ **do**
8 | Inicializa e dispara Thread i;
9 **end**

Algoritmo 2: Multiplicação de matrizes multithread - Processamento das threads

Data: Indentificador
Result: Solução Parcial
1 Recebe identificador (id);
2 $limInferior = id/nLinhas$;
3 \\ *Linha inicial para thread corrente*
4 $limSuperior = limInferior * nlinhas$;
5 \\ *Linha final para thread corrente*
6 **for** $h = limInferior \rightarrow limSuperior$ **do**
7 | **for** $i = 0 \rightarrow numColunasB$ **do**
8 | | **for** $j = 0 \rightarrow numColunasA$ **do**
9 | | | $C[h][i] += A[h][j] * B[j][i]$;
10 | | **end**
11 | **end**
12 **end**

usando um time de *threads* paralelas. Para isso, o algoritmo divide a matriz A em blocos de linhas de mesmo tamanho, cada qual associada a uma *thread* da aplicação, que, por sua vez, são em número equivalente à quantidade de núcleos do processador. Assim, uma matriz de N linhas será dividida em $N/4$ blocos de linhas em um processador de quatro núcleos. Dessa forma, cada núcleo é responsável por uma *thread* da aplicação e, portanto, pelo cálculo do bloco de linhas correspondente da matrix resultante, C.

Versão Java Na versão do código escrito na linguagem Java, a aplicação começa recebendo como parâmetros as dimensões das matrizes A e B. Então, as matrizes A, B e C são alocadas. As matrizes de entrada, A e B, são preenchidas aleatoriamente com valores de 1 a 100 no método buildMatrices(). O próximo passo consiste na chamada para função que deverá realizar o cálculo da matriz resultante (calculate), cujo código pode ser lido no Código 5.1.

Código 5.1: Java: Método calculate

```
 1 public void calculate()
 2 {
 3     threadPool = new MatrixMultiplerThread[nCore];
 4     for(int i=0; i<threadPool.length; i++)
 5     {
 6         threadPool[i] = new MatrixMultiplierThread(i);
 7         threadPool[i].start();
 8         try
 9         {
10             threadPool[i].join();
11         }
12         catch (InterruptedException e)
13         {
14             //thread was interrupted
15         }
16     }
17
18     //print result matrix
19     printResult();
20 }
```

O método calculate cria e inicializa um vetor de *threads* (linha 3), chamado de threadPool, com o número de *threads* igual ao número de núcleos de processamento disponíveis. Para cada *thread*, é repassado um índice que servirá para identificar a *thread* e calcular as posições que cada uma atuará sobre a matriz A (linha 6). O Código 5.2 apresenta o método run da classe MatrixMultiplierThread, subclasse da classe Thread, que representa o cálculo realizado sobre cada *thread* que armazena os resultados na matriz C.

Código 5.2: Java: Método *run* das *threads* da aplicação

```
1  public void run()
2  {
3      for (int h=index*upperBound; h<(index*upperBound)+upperBound; h++)
4      {
5          for(int i=0; i<columnsB; i++)
6          {
7              for(int j=0; j<columnsA; j++)
8                  matrixC[h][i] += matrixA[h][j] * matrixB[j][i];
9          }
10     }
11 }
```

Versão CUDA C A versão escrita em CUDA C também segue o critério de divisão da matriz A em blocos de linhas. Cada bloco de linhas da matriz resultante C agora será calculado por um *kernel* CUDA. Dessa maneira, além do paralelismo obtido na divisão da matriz A em blocos ganha-se com o paralelismo no calculo de cada posição. Cada *kernel* é composto por um bloco de *threads*, das quais cada uma será encarregada de realizar o cálculo de uma posição da matriz resultante C.

Nesta versão, após os passos de criação e preenchimento das matrizes A e B no hospedeiro, é necessário alocar os espaços de dados na memória do dispositivo e transferir as matrizes para esses endereços, essa tarefa pode ser vista no Código 5.3 no bloco das linhas 5 a 12.

Código 5.3: CUDA C: - Alocação e transferências de memória

```
1  int *matrixA = (int*) malloc (sizeof(int)*rowsA*columnsA);
2  int *matrixB = (int*) malloc (sizeof(int)*rowsB*columnsB);
3  int *matrixC = (int*) malloc (sizeof(int)*rowsA*columnsB);
4  ...
5  int matrixA_d;
6  cudaMalloc((void**) &matrixA_d, sizeof(int)*rowsA*columnsA);
7  int matrixB_d [rowsB][columnsB];
8  cudaMalloc((int*) &matrixB_d, sizeof(int)*rowsB*columnsB);
9  int matrixC_d [rowsA][columnsB];
10 cudaMalloc((int*) &matrixC_d, sizeof(int)*rowsA*columnsB);
11
12 cudaMemcpy(matrixB_d, matrixB, sizeof(int)*rowsB*columnsB, cudaMemcpyHostToDevice);
13
14 dim3 num_blocks (1);
15 dim3 block_size (nRows, columnsB);
```

Observa-se que a matriz A não é transferida nesse momento, para execução paralela dos *kernels* sobre a GPU utilizamos *streams* diferentes, cada *streams* receberá um *kernel* com o bloco respectivo da matriz A. A matriz B também poderia ser transferida da mesma forma. Porém, seria necessário sua transposição

favorecendo a leitura dos dados também por blocos. Como nosso objetivo é demonstrar as abstrações propostas para Fusion, não vamos realizar essa transposição sobre B nesse exemplo.

A grade computacional é configurada nas linhas 14 e 15. A grade do *kernel* possui apenas um bloco com suas *threads* distribuídas em duas dimensões de tamanho nRows $\times$ columnsB.

O próximo passo consiste na criação dos *streams* que receberão os *kernels* paralelos. Esse procedimento pode ser visto no Código 5.4. Observa-se que o trecho entre as linhas 6 e 10 realiza a cópia dos dados da matriz A para o correspondente *stream* que receberá o *kernel* atuante sobre aquele espaço de dados específico.

Código 5.4: CUDA C: Criação dos *streams*

```
1  numStreams = nCore;
2  cudaStream_t vectorOfStreams[numStreams];
3
4  for(int stream_id=0; stream_id<numStreams; stream_id++)
5      cudaStreamCreate(&vectorOfStreams[stream_id]);
6
7  for(int stream_id=0; stream_id<numStreams; stream_id++)
8      cudaMemcpyAsync(&matrixA_d[stream_id*upperBound],&matrixA_h[stream_id*upperBound],
9                      nRows*nColumnsA*sizeof(int),cudaMemcpyHostToDevice,
10                     vectorOfStreams[stream_id]);
11
12 for(int stream_id=0; stream_id<numStreams; stream_id++)
13     calculate<<<num_blocks,block_size,0,vectorOfStreams[stream_id]>>>
14             (&matrixA_d[stream_id*upperBound],matrixB,
15              matrixC[stream_id*upperBound], nRows, columnsA, columnsB);
```

O lançamento dos *kernels* paralelos ocorre entre as linhas 12 e 15 do Código 5.4, recebendo as configurações da grade computacional e o identificador da *stream* na qual será lançado. Os argumentos para o *kernel* correspondem ao endereço inicial do bloco de dados da matriz A_d, o endereço da matriz B_d, o endereço da matriz C_d, o número de linhas que cada kernel atuará e o número de colunas de A e B.

O código que implementa o *kernel* responsável por calcular as soluções parciais pode ser visto no Código 5.5. No trecho de código entre as linhas 13 e 17, a *thread* com indice (0,0) é responsável por copiar o bloco da matriz A para memória compartilhada no dispositivo.

Código 5.5: CUDA C: *Kernel* calculate

```
1 __global__ calculate (int *A_d, int *B_d, int *C_d,
2                       int nRows, int nColA, int nColB,
3                       int stream)
4 {
```

```
5     int idx = threadIdx.x;
6     int idy = threadIdx.y;
7     int aBegin = stream*nRows*nColA;
8     int cBegin = stream*nRows*nColB;
9     int i, j;
10    __shared int As [nRows*nColA];
11    int subC = 0;
12
13    if (idx+idy=0)
14    {
15        for(i=0,j=aBegin ; i<nRows*nColA; j++ , i++)
16            As[i] = A_d[j];
17    }
18
19    __syncthreads();
20
21    for(int i=0; i<nColA; i++)
22        subC += As[nColA*idy+i] * B_d[i*nColB+idx];
23
24    C_d[cBegin+idy*nColB+idx] = subC;
25 }
```

Versão Fusion Para versão Fusion, são criadas duas classes. A primeira classe corresponde a aplicação, chamada de MatrixMultiUnits é responsável por receber as informações referentes às matrizes A e B e reservar espaço para matriz C. A segunda classe, chamada de MatrixAccel consiste de uma classe aceleradora, ou seja, é através de objetos instanciados a partir dessa classe que a aplicação conseguirá acelerar sua computação usando a GPU.

A classe MatrixMultiUnits possui exatamente os mesmos métodos que a primeira classe apresentada, MatrixMultiThreads na versão escrita em Java puro. Após a fase de inicialização das matrizes, o método calculate() da classe, apresentado no código 5.6, é invocado, responsável por inicializar e disparar as *threads* para aplicação. Sua principal diferença encontra-se no trabalho das *threads*.

Código 5.6: Fusion: Método calculate.

```
1  private void calculate()
2  {
3      //Number of threads equals of the rows
4      threadPool = new myThread[nCore];
5
6      //Instantiates the acelerator object for linking to the GPU
7      accelMulti = new MatrixAccel(nRows);
8      accelMulti.setMatrix(matrixA, matrixB);
9
10     for(int i=0; i<nCore; i++)
11     {
12         threadPool[i] = new MyThread(i, accelMulti);
```

```
13            threadPool[i].start();
14            try
15                threadPool[i].join();
16            catch (InterruptedException e) { //thread was interrupted }
17
18        }
19
20        //Get result on accelerator object
21        matrixC = accelMulti.getResult();
22
23        //Clears the object variable throttle
24        accelMulti.clearAll();
25    }
```

No Código 5.6, é possível perceber o objeto acelerador accelMulti, instanciado na linha 7 a partir da classe aceleradora MatrixMulti. As *threads* hospedeiras são disparadas na linha 13 pela aplicação, as quais serão responsável por instanciar as unidades paralelas desse objeto, o que pode ser visto na linha 14 do Código 5.7.

Código 5.7: Fusion: Método run() das *threads* hospedeiras.

```
1  private static class MyThread extends Thread
2  {
3      int index;
4      MatrixAccell accelMulti;
5
6      MyThread(int index, MatrixAccel accelMulti)
7      {
8          this.index = index;
9          this.accelMulti = accelMulti;
10     }
11
12     public void run()
13     {
14         MatrixAccell:Multiply accelMulti_unit = unit Multiply <nCore> of accelMulti;
15         accelMulti_unit.multiplyBlockLine();
16     }
17 }
```

Em execução, cada *thread* hospedeira realiza uma chamada ao método multiplyBlockLine(), visualizado na linha 15 do Código 5.7, principal método da classe MatrixAccell, implementado na unidade paralela chamada Multiply. Sempre que esse método é chamado por uma *thread* hospedeira, através do objeto accelMulti_unit, todas as unidades instanciadas deverão disparar o método *kernel* sobre o dispositivo GPU que irá calcular uma solução parcial para o problema. No Código 5.8, é possível observar a implementação da unidade paralela Multiply.

Código 5.8: Fusion: Unidade paralela multiply.

```
accelerator class MatrixAccell
{
    (...)
    parallel unit Multiply
    {
        Multiply()
        {
            int rank = getRank();
            memCpyA(matrixA_d, columnsA*rank,
                    matrixA_h, columnsA*rank,
                    columnsA, rank);
        }

        public kernel multiplyBlockLine() threads<<<nRows,columnsB>>> blocks<<<1>>>
        {
            int idx = threadIdx(x);
            int idy = threadIdx(y);
            int aBegin = rank*nRows*columnsA;
            int cBegin = rank*nRows*columnsB;
            int i, j;
            shared int As [nRows*columnsA];
            int subC = 0;

            if (idx+idy==0)
            {
                for(i=0,j=aBegin ; i<nRows*columnsA; j++ , i++)
                    As[i] = matrixA_d[j];
            }

            synchronized(threads);

            for(i=0; i<columnsA; i++)
                subC += As[columnsA*idy+i] * matrixB_d[i*columnsB+idx];

            matrixC_d[cBegin+idy*columnsB+idx] = subC;
        }

        public void getResultUnit()
        {
            memCpyA(matrixC_h, columnsB*nRows*rank,
                    matrixC_d, columnsB*nRows*rank,
                    columnsB*nRows, rank);
        }
    }
}
```

No Código 5.8, destaca-se a presença do método getResultUnit(). Quando chamado, esse método é responsável por recuperar a solução parcial calculada, realizando uma cópia da porção da solução calculada pela unidade corrente sobre o dispositivo para o hospedeiro, mais precisamente para o método getResult() do objeto accelMulti, que retorna a matriz C completa para aplicação. A chamada do

método getResult é realizada no método calculate na aplicação, no Código 5.6 (linha 21).

5.3 Segundo Estudo de Caso: Enumeração Completa

Em 2006, pesquisadores da Universidade de Berkelley estenderam a conhecida taxonomia chamada de *Dwarf Mine* [Asanovic et al. 2006] com a categoria de aplicações B&B (*Branch-and-Bound*), também conhecidos, como problemas da mochila [Boukedjar, Lalami e Baz 2012]. Essa taxonomia organiza algoritmos de reconhecido interesse científico de acordo com características relevantes como padrões de comunicação entre processos e de acesso à memória. A aplicação escolhida para esse estudo de caso encontra-se nessa categoria, pois trata-se do PCVA (Problema do Caixeiro Viajante Assimétrico).

A execução de aplicações B&B consistem de três etapas. A primeira etapa consiste no particionamento ou ramificação do problema (do inglês, *branching*), onde é dividido em sub-problemas. A segunda etapa consiste na avaliação de soluções parciais e finais, definidas através de limites inferiores e superiores, por isso conhecida como etapa delimitadora (do inglês, *bounding*). A última etapa realiza um corte no espaço de busca também conhecida como poda (do inglês, *pruning*).

A implementação apresentada nessa seção é baseada no trabalho de Pessoa 2012, no qual os autores propõem uma nova abordagem para aplicações B&B sobre GPGPU.

Uma das formas de obter paralelismo em aplicações B&B é no cálculo dos sub-problemas gerados pela etapa de ramificação. Para tal, busca-se em uma determinada profundidade da árvore de soluções um nó ativo, ou seja, um nó que representa uma solução incompleta. Esse nó é então armazenado em um conjunto de nós avaliados mas não expandidos, chamado de conjunto ativo.

Em seu trabalho, Pessoa et al. 2011 implementam a inicialização do algoritmo em um processo serial para gerar uma quantidade significativa de *threads*, a fim de obter uma ocupação considerável sobre o dispositivo GPU, povoando o conjunto ativo de nós. Esse conjunto de nós passa a representar o novo conjunto de raízes para busca da melhor solução em cada uma das ramificações. A seleção dos nós em uma determinada profundidade visa obter um maior número de ramificações, que podem ser avaliadas independentemente, sendo cada nó uma solução válida para o problema [Papadimitriou e Steiglitz 1998].

Pessoa et al. 2011. exploram o paralelismo na busca concorrente sobre as

ramificações, através de uma estratégia DFS (*depth-first search*) implementada em um *kernel* específico, chamado dfs_cuda_UB_stream e apresentado no Código 5.9. Cada nó consiste de uma raiz para o *kernel* DFS, responsável por buscar a melhor solução local em uma ramificação. A paralelização ocorre através do uso de *streams*, garantindo que as raízes serão calculadas em paralelo, através dos *kernels* DFS-BB executados cada qual sobre uma *stream* independente.

Código 5.9: Kernel DFS-BB GPGPU [Pessoa 2012]

```
1
2 __global__ void dfs_cuda_UB_stream(int N,int stream_size, int *mat_d,
3                                    short *preFixos_d, int nivelPrefixo, int upper_bound,
4                                    int *sols_d, int *melhorSol_d)
5 {
6     register int idx = blockIdx.x * blockDim.x + threadIdx.x;
7     register int flag[16];
8     register int vertice[16];
9     register int N_l = N;
10    register int i, nivel;
11    register int custo;
12    register int qtd_solucoes_thread = 0;
13    register int UB_local = upper_bound;
14    register int nivelGlobal = nivelPrefixo;
15    int stream_size_l = stream_size;
16    if (idx < stream_size_l)
17    {
18      for (i = 0; i < N_l; ++i)
19      {
20          vertice[i] = _VAZIO_;
21          flag[i] = _NAO_VISITADO_;
22      }
23      vertice[0] = 0;
24      flag[0] = _VISITADO_;
25      custo= ZERO;
26      for (i = 1; i < nivelGlobal; ++i)
27      {
28          vertice[i] = preFixos_d[idx * nivelGlobal + i];
29          flag[vertice[i]] = _VISITADO_;
30          custo += mat_d(vertice[i-1],vertice[i]);
31      }
32      nivel=nivelGlobal;
33      while (nivel >= nivelGlobal )
34      {
35        if (vertice[nivel] != _VAZIO_)
36        {
37            flag[vertice[nivel]] = _NAO_VISITADO_;
38            custo -= mat_d(vertice[anterior(nivel)],vertice[nivel]);
39        }
40        do
41        {
42            vertice[nivel]++;
```

```
43          } while (vertice[nivel] < N_l && flag[vertice[nivel]]);
44          if (vertice[nivel] < N_l)
45          {
46            custo += mat_d(vertice[anterior(nivel)],vertice[nivel]);
47            flag[vertice[nivel]] = _VISITADO_;
48            nivel++;
49            if (nivel == N_l)
50            {
51              ++qtd_solucoes_thread;
52              if (custo+mat_d(vertice[anterior(nivel)],0)<UB_local)
53              {
54                UB_local = custo + mat_d(vertice[anterior(nivel)],0);
55              }
56              nivel--;
57            }
58          }
59          else
60          {
61            vertice[nivel] = _VAZIO_;
62            nivel--;
63          }
64        }
65        sols_d[idx] = qtd_solucoes_thread;
66        melhorSol_d[idx] = UB_local;
67      }
68  }
```

Essa é uma estratégia de busca em profundidade onde os nós são analisados na ordem em que são gerados. Mais detalhes sobre estratégias de busca paralelas em algoritmos B&B podem ser analisadas no trabalho de [Grama et al. 2003].

Antes do disparo de cada *kernel* DFS-BB, entre as linhas 25 e 28 do Código 5.10, os dados são transferidos para o dispositivo, com as porções correspondentes a cada *kernel* transferidos indicando-se especificamente a *stream* na qual o *kernel* será lançado (linhas 6 a 22). O número de *streams* é calculado antes da chamada dos *kernels* DFS e depende do número de raízes criadas.

Código 5.10: Criação das streams e lançamento dos kernels DFS-BB [Pessoa 2012]

```
1  cudaStream_t vectorOfStreams[numStreams];
2
3  for(int stream_id=0; stream_id<numStreams; stream_id++)
4    cudaStreamCreate(&vectorOfStreams[stream_id]);
5
6  for(int stream_id=0; stream_id<numStreams; stream_id++)
7        cudaMemcpyAsync(&path_d[stream_id*chunk*nivelPreFixos],
8                        &path_h[stream_id*chunk*nivelPreFixos],
9                        qtd_threads_streams[stream_id]*sizeof(short)*nivelPreFixos,
10                       cudaMemcpyHostToDevice,vectorOfStreams[stream_id]);
11
12 for(int stream_id=0; stream_id<numStreams; stream_id++)
```

```
13      cudaMemcpyAsync(&melhorSol_d[stream_id*chunk],
14                      &melhorSol_h[stream_id*chunk],
15                      qtd_threads_streams[stream_id]*sizeof(int),
16                      cudaMemcpyHostToDevice,vectorOfStreams[stream_id]);
17
18 for(int stream_id=0; stream_id<numStreams; stream_id++)
19      cudaMemcpyAsync(&sols_d[stream_id*chunk],
20                      &sols_h[stream_id*chunk],
21                      qtd_threads_streams[stream_id]*sizeof(int),
22                      cudaMemcpyHostToDevice,vectorOfStreams[stream_id]);
23
24 for(int stream_id=0; stream_id<numStreams; stream_id++)
25   dfs_cuda_UB_stream<<<num_blocks,block_size,0, vectorOfStreams[stream_id]>>>
26                 (N,qtd_threads_streams[stream_id],mat_d,
27                  &path_d[stream_id*chunk*nivelPreFixos],nivelPreFixos,
28                  999999,&sols_d[stream_id*chunk], &melhorSol_d[stream_id*chunk]);
```

No código Fusion, construímos uma classe aceleradora chamada EnumerationAccel, responsável por encapsular todas as informações referentes a aplicação que serão executadas no dispositivo. Assim, os objetos aceleradores instanciados a partir dessa classe podem ser substituídos por objetos comuns sem comprometer o código da aplicação.

A classe EnumerationAccel possui uma unidade paralela chamada Enumerate, que implementa o método *kernel* dfs, o qual representa o kernel DFS-BB. O trecho de código para a unidade pode ser visto no Código 5.11. O método *kernel* dfs é muito semelhante ao *kernel* CUDA C e, por esse motivo, o código do seu corpo é omitido no Código 5.11 (linha 17). A versão completa do código da classe EnumerationAccel pode ser vista no Anexo B.2.3.

Código 5.11: Implementação da unidade paralela enumarate e do método kernel dfs

```
1  parallel unit Enumerate
2  {
3    Enumerate()
4    {
5      int rank = rank();
6      memCpyA(path_d, rank*chunk*nivelPreFixos, path_h,
7              rank*chunk*nivelPreFixos, qtd_threads_streams[rank]*nivelPreFixos);
8      memCpyA(melhorSol_d, rank*chunk, melhorSol_h, rank*chunk,
9              qtd_threads_streams[rank]);
10     memCpyA(sols_d, rank*chunk, sols_h, rank*chunk, qtd_threads_streams[rank]);
11   }
12
13   public kernel dfs (int upper_bound)
14     threads<<<192>>>
15     blocks<<<nPreFixos / numThreads(x) + (nPreFixos % numThreads(x) == 0 ? 0 : 1)>>>
16   {
17     (...)
```

```
18    }
19   }
20 }
```

Em uma computação *multi-thread* Java, as unidades paralelas do objeto acelerador são instanciadas pelas *threads* hospedeiras. No Código 5.11, é possível observar que cada unidade realiza suas próprias cópias dos dados para o dispositivo (linhas 4 a 8). As unidades são instanciadas através de um conjunto de *threads* hospedeiras criadas e instanciadas pela aplicação. A criação do conjunto de *threads* para um objeto específico pode ser visto no Código 5.12, onde size indica a quantidade de *threads* que serão criadas. Note que objeto acelerador é mapeado ao dispositivo 0.

Código 5.12: Instanciação do objeto acelerador Enumeration

```
1 enumeration = new EnumerationAccel<0> (chunk, nPreFixos, N, nivelPreFixos,
2   mat_h, path_h, melhorSol_h, sols_h, size, qtd_threads_streams);
3 ForkJoinPool forkJoinPool = new ForkJoinPool(size);
4 forkJoinPool.invoke(new TaskEnumeration(this, Enumeration) );
5 otimo_global = enumeration.getMelhorSol();
```

A instanciação da unidade ocorre dentro de cada tarefa do conjunto criado. No Código 5.13, podemos ver a instanciação na linha 9 e a chamada do método kernel paralelo na linha 16. Lembrando que o método só será executado no momento em que todas as size unidades forem instanciadas.

Código 5.13: Instanciação e disparo do objeto acelerador Enumeration

```
1 public class TaskEnumeration extends RecursiveTask<Double>
2 {
3   private EnumerationAccel accel_obj;
4   private int rank;
5
6   TaskEnumeration(EnumerationAccel accel_obj)
7   {
8     super();
9     this.accel_obj = accel_obj;
10  }
11
12  protected Object compute()
13  {
14    private EnumerationAccel:Enumerate accel_obj_unit;
15    accel_obj_unit = unit Enumerate<size> of accel_obj;
16    accel_obj_unit.dfs(999999);
17    return null;
18  }
19 }
```

Os códigos completos para essa aplicação, escritos na linguagem *Fusion* encontram-se no Anexo B.2.

5.4 Considerações

A linguagem Fusion possui a característica de encapsular nos objetos aceleradores toda complexidade associada a arquitetura GPU. Dessa forma, remove do usuário especialista a responsabilidade do uso adequado do dispositivo, preocupando-se apenas com a modelagem de sua aplicação.

Essa característica pode ser observada em ambos os estudos de caso. Os objetos aceleradores accelMulti e accel_obj são os responsáveis por todo o trabalho sobre o dispositivo, através da implementação de seus métodos utilizados em suas unidades. Assim, o usuário especialista possui um contato mínimo com a arquitetura GPU, tornando o processo de comunicação com um dispositivo GPU quase transparente para a aplicação.

Observa-se também que a utilização da linguagem Fusion fornece dois níveis de paralelismo, atendendo a requisitos de computação heterogênea *multicore/manycore*. O primeiro nível refere-se ao processador *multicore* hospedeiro, onde o usuário especialista trabalha com uma granularidade média de tarefas, executadas pelas *threads* hospedeiras. O segundo nível encontra-se encapsulado no objeto acelerador, com uma granularidade fina de tarefas, onde a divisão agora ocorre sobre os blocos de *threads* da grade computacional de cada método *kernel*.

Assim, no momento em que o usuário especialista identifica partes críticas para o desempenho de sua aplicação, poderá utilizar objetos aceleradores para buscar recursos computacionais sobre dispositivos GPUs disponíveis. Essa é uma tarefa tornada mais fácil pela simples mudança dos objetos comuns de sua aplicação, responsáveis por essas partes críticas, por objetos aceleradores que supõem-se corretamente implementados.

Um objeto acelerador considerado corretamente implementado deve permitir que o usuário especialista utilize seus métodos como se utiliza-se métodos de objetos comuns, sem se preocupar em como e onde seus problemas serão avaliados. Portanto, é responsabilidade do usuário desenvolvedor implementar de forma adequada seus objetos aceleradores, disponibilizando métodos para o objeto os quais permitam um acesso as informações essenciais para aplicação. Nesse caso, um objeto acelerador deve se comportar de forma semelhante a objetos comuns.

Os estudos ainda demonstram que a linguagem Fusion permite que *threads* da

aplicação, ditas hospedeiras, lancem trabalhos para o dispositivo GPU de forma transparente. Apesar do usuário estar estendendo unidades nas suas *threads*, na aplicação, o que poderia levar a um pensamento explícito sobre o uso de um dispositivos GPU, não necessariamente essa unidades fazem parte de um objeto acelerador. Nesse sentido, é possível associar uma unidade a uma classe interna à classe do objeto em uso, um modelo de programação comum em aplicações Java. Dessa forma, ao utilizar um objeto na linguagem Fusion, o usuário poderá estar utilizando um dispositivo GPU, desde de que esse seja uma instância de uma classe aceleradora, na qual suas unidades abrem um canal de comunicação com o dispositivo.

Os códigos completos para os estudos de caso apresentados aqui podem ser estudados nos Anexos A e B.

Capítulo 6

Conclusões e Trabalhos Futuros

A busca por novas adaptações visando flexibilizar a modelagem e o desenvolvimento das aplicações sobre as arquiteturas GPUs tem crescido a cada ano. Prova disso são as extensões de linguagens consolidadas para ligação entre processador hospedeiro e dispositivo GPU. Porém, ainda há muito para ser desenvolvido, especialmente no que diz respeito a computação paralela heterogênea envolvendo processadores *multicore* e coprocessadores *manycore* do tipo GPU.

Até o presente momento, foi possível especificar as abstrações que estão incluídas na linguagem Fusion, a qual em sua primeira versão estende a linguagem de programação Java. Para validar o protótipo da linguagem Fusion, foram elaborados estudos de caso para essa dissertação, que permitiram analisar o modelo de programação proposto.

A linguagem Fusion é considerada promissora, contemplando parte de um futuro projeto, de disponibilizar uma plataforma de programação de alto nível para computações de propósito geral sobre GPUs, baseada na orientação a objetos. As abstrações propostas na linguagem tem por premissa manter um estilo orientada a objetos no desenvolvimento GPGPU, permitindo maior abstração sobre detalhes específicos do hardware GPU e introduzindo maior flexibilidade ao desenvolvimento.

6.1 Cumprimento dos Objetivos

Durante as etapas do projeto da linguagem Fusion, foram realizados estudos em diversas áreas, através de uma coletânea de artigos, periódicos, manuais, etc, os quais permitiram a evolução do projeto, vislumbrando a complexidade das arquiteturas GPUs e as soluções adotadas para remover limitações a cada geração, justificando a preocupação em manter a elaboração e o desenvolvimento das abstrações e da

linguagem sempre pensando nas tecnologias atuais.

Os estudos relacionados às arquiteturas GPUs permitiram compreender toda a evolução das arquiteturas de aceleradores gráficos, desde sua concepção. Inicialmente tratados como unidades dedicadas ao processamento gráfico, evoluindo para aceleradores de propósito geral, com partes programáveis através de APIs específicas e limitadas, até chegarem no patamar em que se encontram, onde um dispositivo gráfico é utilizado amplamente para computações de propósito geral, com linguagens específicas para sua programação.

Alguns dos trabalhos relacionados estudados nos chamaram a atenção, tais como a linguagem JaBEE e compilador RootBeer. Esses trabalhos possuem um esquema de tradução semelhante a adotada pela linguagem Fusion, na qual é possível criar objetos na aplicação, escrita em Java, que fazem uso dos dispositivos GPUs para acelerar seu comportamento. Porém, as classes aceleradoras nessas ferramentas são definidas pela extensão de uma classe específica, que realiza a comunicação com a arquitetura CUDA e posteriormente com o dispositivo.

Não distante disso, a principal diferença na linguagem Fusion está na própria classe aceleradora, que possui essa responsabilidade, disponibilizando mecanismos para o programador do objeto implemente métodos de forma otimizada, utilizando estruturas explícitas que permitem a manipulação dos dados e das tarefas da aplicação sobre o dispositivo GPU. Como se estivesse programando diretamente sobre o mesmo, ou seja, expondo o dispositivo como um coprocessador, objetivo também da linguagem CUDA C.

As abstrações propostas pela linguagem Fusion seguem a modelagem orientada a objetos, definindo um objeto acelerador composto por unidades que comunicam-se com o dispositivo. A execução em filas de execução separadas fica implícita na definição da unidade. Porém, essa é uma configuração que pode ser alterada pelo programador do objeto, mostrando o poder e a flexibilidade que a linguagem Fusion contempla seus usuários.

Portanto, a maior preocupação da linguagem Fusion é reduzir a complexidade associada a modelagem da aplicação. Dessa forma uma aplicação *multithread* Java consegue, através dos objetos aceleradores, fazer uso da arquitetura GPU. E a implementação do objeto acelerador pode ser realizada de forma otimizada, onde o programador do objeto pode pensar no trabalho que cada *thread* da sua grade computacional irá computar.

É natural, quando se trata de paralelismo a nível de *threads*, associar a cada

thread hospedeira uma tarefa especifica. No caso da linguagem Fusion, cada *thread* da aplicação paralela poderá paralelizar sua tarefa através de uma grade computacional oferecida pelo dispositivo GPU, tomando melhor proveito de mais um nível de paralelismo, exposto pela GPU.

O usuário especialista conhece somente o primeiro nível, ou seja, o paralelismo no nível do processador hospedeiro. Por outro lado, o usuário desenvolvedor tem o poder de manipular o segundo nível de paralelismo diretamente sobre o dispositivo, através da implementação do objeto acelerador. Dessa forma, o usuário especialista preocupa-se em manter uma granularidade média dos dados ficando a cargo do desenvolvedor do objeto acelerador trabalhar com uma granularidade fina sobre o dispositivo.

Algumas das principais dificuldades encontradas no desenvolvimento da linguagem Fusion, são referentes a evolução muito rápida das arquiteturas, gerando dúvidas devido a documentações imprecisas e muitas vezes contraditórias, prejudicando a elaboração de abstrações que pudessem tirar proveito das novas tecnologias prometidas.

Alguns conceitos acabaram desmistificados, outros não, devido a busca constante por manter atualizado o desenvolvimento da linguagem Fusion, muitas vezes nos deparamos com conceitos que causaram modelagens diferentes da versão atual, mas de grande valia, demonstrando e enfatizando os principais requisitos que uma linguagem de alto nível deve seguir para o desenvolvimento GPGPU.

O atraso no início da implementação do compilador deu-se devido a constante atualização do, ainda em desenvolvimento, projeto VMKit. Com diversos problemas de versionamento e constantes trocas de bibliotecas, optou-se por uma simulação do que ainda está por ser desenvolvido, com a geração de dois códigos manualmente, um Java/JNI e outro CUDA C.

Apesar de tudo, podemos observar na modelagem dos estudos de caso que um objeto comum Java pode ser substituído por um objeto acelerador, o qual encapsula todo paralelismo e complexidades associada a computação sobre GPU. Assim, a aplicação pode obter um melhor desempenho fazendo uso de um dispositivo GPU disponível, logicamente se bem implementada seguindo conceitos de programação paralela para obter um nível paralelismo aceitável, tanto a nível de processador hospedeiro quanto do dispositivo.

Essa flexibilidade traz consigo benefícios para programação GPGPU. Uma aplicação modelada com objetos específicos para cada operação permite ao

desenvolvedor acelerar somente as partes que julga necessárias, consideradas críticas para o desempenho da aplicação como um todo, simplesmente substituindo os objetos comuns por objetos aceleradores, sendo esse um dos objetivos da linguagem Fusion, discutido durante essa dissertação.

Para essa primeira versão da linguagem Fusion, estendendo a linguagem Java, seu desempenho poderá ser comprometido pelas fases de compilação. A geração de dois códigos após a compilação do código Fusion acarretará em um tempo excedente se comparado a modelagem utilizando somente CUDA C. Esse contratempo já é esperado e poderá ser minimizado em trabalhos futuros com o desenvolvimento de um compilador otimizado, gerando diretamente código PTX a partir do código Fusion.

6.1.1 Trabalhos Futuros

Essa dissertação teve seu foco principal na definição e avaliação de abstrações linguísticas sobre a linguagem Java, que definem a linguagem Fusion, ainda não apresentando um compilador definitivo e avaliação de desempenho de uma implementação deste. A respeito da implementação de um compilador, resume-se a apresentar a arquitetura de um compilador protótipo sobre a ferramenta VMKit, com o objetivo de evidenciar sua viabilidade prática.

Dando continuidade a esse projeto, como trabalhos futuros, vislumbra-se o desenvolvimento do compilador definitivo para linguagem Fusion, com otimizações especificas para as operações que envolvem tanto a comunicação como a modelagem das aplicações sobre dispositivos GPU. De posse de um compilador, vários outros trabalhos podem ser realizados.

Avaliação de desempenho: até o presente momento, não havia disponível uma plataforma com as tecnologia *Hyper-Q* e *Dynamic Paralelism* apresentadas nesse trabalho. A placa GPU adquirida pelo nosso grupo de pesquisa, uma Geforce GTX 690, da NVIDIA, ainda utiliza processadores GK104, a qual não suporta ainda essas duas tecnologias, ao contrário do que era sugerido em sua documentação, onde é anunciada como suportando a arquitetura Kepler. Porém, trabalhos futuros poderão executar as aplicações modeladas sobre a arquitetura Kepler GK110, e assim avaliar o poder computacional da linguagem Fusion. Para isso, nosso grupo de pesquisa já adquiriu, recentemente, uma GPU Geforce GTX Titan.

Tradução direta: é desejável que o compilador seja capaz de realizar a tradução direta entre o código Fusion e o código PTX, para tal se faz necessário um estudo

mais abrangente da estrutura do código PTX para identificar como as estruturas são mapeadas e tratadas desde sua geração no código em alto nível, antes de serem definitivamente traduzidos para código de máquina específico para arquitetura alvo. O projeto VMKit, apesar de suas limitações, deverá ser utilizado na construção de uma máquina virtual capaz de gerar código intermediário para geração do código PTX, o que é de suma importância para o futuro do projeto. Busca-se a utilização do projeto LLVM e do seu compilador de saída NVPTX para geração do código PTX.

Coletor de lixo: um ponto desafiador a ser estudado consiste em compatibilizar o coletor de lixo da linguagem Java para alcançar as variáveis declaradas nas hierarquias de memória da GPU. O coletor de lixo Java realiza a reciclagem das variáveis na memória liberando o espaço daquelas que não estão sendo utilizadas de forma automática. Na arquitetura CUDA, os espaços de memória são liberados explicitamente pelo programador, o que vai de encontro a semântica Java com seu coletor de lixo automático. Atualmente, em Fusion, variáveis declaradas nas hierarquias de memória do dispositivo devem ser explicitamente desalocadas.

Otimização de estruturas: com relação a linguagem Fusion existem muitas funções que ainda devem ser desenvolvidas. Por exemplo, a arquitetura CUDA evidencia algumas estruturas para transferências e alocação de espaços específicos de memória, como as caches de constante e texturas, que não são abordadas nessa versão inicial da linguagem.

Referências Bibliográficas

Asanovic et al. 2006 ASANOVIC, K.; BODIK, R.; CATANZARO, B. C.; GEBIS, J. J.; HUSBANDS, P.; KEUTZER, K.; PATTERSON, D. A.; PLISHKER, W. L.; SHALF, J.; WILLIAMS, S. W.; YELICK, K. A. *The Landscape of Parallel Computing Research: A View from Berkeley*. [S.l.], 2006. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>.

Bell e Hoberock BELL, N.; HOBEROCK, J. Thrust: A productivity-oriented library for cuda. In: ______. *GPU Gems*. [S.l.: s.n.].

Blythe 2006 BLYTHE, D. The direct3D 10 system. *ACM Trans. Graph*, 2006. v. 25, n. 3, p. 724–734, 2006. Disponível em: <http://doi.acm.org/10.1145/1141911.1141947>.

Boukedjar, Lalami e Baz 2012 BOUKEDJAR, A.; LALAMI, M. E.; BAZ, D. E. Parallel Branch and Bound on a CPU-GPU System. In: STOTZKA, R.; SCHIFFERS, M.; COTRONIS, Y. (Ed.). *PDP - Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, Munich, Germany*. IEEE, 2012. p. 392–398. ISBN 978-1-4673-0226-5. Disponível em: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6168524>.

Buck et al. 2004 BUCK, I.; FOLEY, T.; HORN, D.; SUGERMAN, J.; FATAHALIAN, K.; HOUSTON, M.; HANRAHAN, P. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 2004. v. 23, n. 3, p. 777–786, ago. 2004. ISSN 0730-0301 (print), 1557-7368 (electronic).

CUDA 2013 CUDA. *CUDA LLVM Compiler*. 2013. Disponível em: <http://developer.nvidia.com/cuda-llvm-compiler>. Acesso em: Maio, 2013.

CUDA1.1 2012 CUDA1.1. *NVIDIA CUDA Architecture Overview Version 1.1*. 2012. Disponível em: <http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf>.

CUDA4.1 2012 CUDA4.1. *NVIDIA CUDA Programming Guide Version 4.1*. 2012. Disponível em: <http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf>. Acesso em: Dezembro, 2012.

CUDA4.2 2012 CUDA4.2. *NVIDIA CUDA Programming Guide Version 4.2*. 2012. Disponível em: <http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf>. Acesso em: Janeiro, 2013.

CUDAZone 2012 CUDAZONE. *NVIDIA Developer Zone*. 2012. Disponível em: <http://developer.nvidia.com/cuda/cuda-toolkit>. Acesso em: Maio, 2013.

Dahl 1968 DAHL, O. J. *SIMULA 67 Common Base Language*. [S.l.]: Norwegian Computing Center, 1968. ISBN B0007JZ9J6.

Dahl 2002 DAHL, O. J. The Birth of Object Orientation: the Simula Languages. In: *Software Pioneers: Contributions to Software Engineering, Programming, and Operating Systems Series*. [S.l.]: Springer, 2002. p. 79–90.

Deutsch e Goldberg 1991 DEUTSCH, L. P.; GOLDBERG, A. Smalltalk: Yesterday, Today, and Tomorrow: The trial balloon of a decade ago is now flying high. *Byte Magazine*, 1991. v. 16, n. 8, p. 108–110, 112–115, ago. 1991. ISSN 0360-5280.

Dongarra et al. 1996 DONGARRA, J.; OTTO, S. W.; SNIR, M.; WALKER, D. A Message Passing Standard for MPP and Workstation. *Communications of ACM*, 1996. v. 39, n. 7, p. 84–90, 1996.

Duran e Klemm 2012 DURAN, A.; KLEMM, M. The Intel® Many Integrated Core Architecture. In: *2012 International Conference on High Performance Computing and Simulation (HPCS)*. [S.l.]: IEEE Computer Society, 2012. p. 365–366. ISBN 978-1-4673-2359-8.

Fermi 2012 FERMI. *NVIDIA Next Generation CUDA^{TM} Compute Architecture: Fermi^{TM}*. 2012. Disponível em: <http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>. Acesso em: Agosto, 2012.

Fortran 2012 FORTRAN, P. C. *CUDA Fortran: Programming Guide and Reference*. 2012. Disponível em: <http://www.pgroup.com/doc/pgicudaforug.pdf>. Acesso em: Outubro, 2012.

Geoffray et al. 2010 GEOFFRAY, N.; THOMAS, G.; J.LAWALL; MULLER, G.; FOLLIOT, B. VMKit: a Substrate for Managed Runtime Environments. In: *Virtual Execution Environment Conference*. Pittsburgh, USA: ACM Press, 2010.

Grama et al. 2003 GRAMA, A.; KARYPIS, G.; GUPTA, A.; KUMAR, V. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. [S.l.]: Addison-Wesley, 2003. ISBN 0201648652.

Group 2012 GROUP, T. P. *PGI Inside*. 2012. Disponível em: <http://www.pgroup.com/lit/articles/insider/v1n3a2.htm>. Acesso em: Outubro, 2012.

Hamilton 2003 HAMILTON, J. Language Integration in the Common Language Runtime. *SIGPLAN Notices*, 2003. ACM, New York, NY, USA, v. 38, n. 2, p. 19–28, fev. 2003. ISSN 0362-1340.

Han e Abdelrahman 2011 HAN, T. D.; ABDELRAHMAN, T. S. hicuda: High-level gpgpu programming. *IEEE Transactions on Parallel and Distributed Systems*, 2011. IEEE Computer Society, Los Alamitos, CA, USA, v. 22, p. 78–90, 2011. ISSN 1045-9219.

Han e Abdelrahman 2012 HAN, T. D.; ABDELRAHMAN, T. S. *Projeto hiCUDA*. 2012. Disponível em: <www.eecg.utoronto.ca/t̃sa/hicuda/>. Acesso em: Agosto, 2012.

Han 2009 HAN, T. Y. D. *Directive-based General-purpose GPU Programming*. 2009. Disponível em: <http://hdl.handle.net/1807/18321>.

Herbordt et al. 2007 HERBORDT, M. C.; VANCOURT, T.; GU, Y.; SUKHWANI, B.; CONTI, A.; MODEL, J.; DISABELLO, D. Achieving High Performance with FPGA-Based Computing. *Computer*, 2007. IEEE Computer Society Press, Los Alamitos, CA, USA, v. 40, p. 50–57, March 2007. ISSN 0018-9162. Disponível em: <http://dl.acm.org/citation.cfm?id=1251558.1251716>.

Herbordt et al. 2007 HERBORDT, M. C.; VANCOURT, T.; GU, Y.; SUKHWANI, B.; CONTI, A.; MODEL, J.; DISABELLO, D. Achieving high performance with FPGA-based computing. *Computer*, 2007. v. 40, n. 3, p. 50–57, mar. 2007. ISSN 0018-9162 (print), 1558-0814 (electronic).

Illinois 2012 ILLINOIS, U. of. *The LLVM Compiler Infraestructure*. 2012. Disponível em: <llvm.org>. Acesso em: Setembro, 2012.

Illinois 2013 ILLINOIS, U. of. *LLVM Language Reference Manual*. 2013. Disponível em: <http://www.llvm.org/releases/3.1/docs/LangRef.html>. Acesso em: April, 2013.

KeplerGK110 2012 KEPLERGK110. *NVIDIA Next Generation CUDA^{TM} Compute Architecture: Kepler^{TM} GK110*. 2012. Disponível em: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>. Acesso em: Dezembro, 2012.

Kirk e Hwu 2010 KIRK, D. B.; HWU, W. mei W. *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufmann, 2010. I–XVIII, 1–258 p. ISBN 978-0-12-381472-2. Disponível em: <http://www.elsevier.com/wps/find/bookdescription.cws_home/722320/description#description>.

Klöckner 2012 KLöCKNER, A. *Project pyCUDA*. 2012. Disponível em: <mathema.tician.de/software/pycuda>. Acesso em: maio de 2012.

Kronos 2012 KRONOS. *Kronos Group*. 2012. Disponível em: <http://www.khronos.org/>. Acesso em: Janeiro, 2012.

Liang 1999 LIANG, S. *Java Native Interface: Programmer's Guide and Specification*. [S.l.]: Addison-Wesley, 1999. ISBN 0-201-32577-2.

Munshi et al. 2005 MUNSHI, A.; WONG, A.; CLINTON, A.; BRAGANZA, S.; BISHOP, W.; MCCOOL, M. A Parameterizable SIMD Stream Processor. In: *Electrical and Computer Engineering, Canadian Conference on*. [S.l.: s.n.], 2005. p. 806–811. ISSN 0840-7789.

Nguyen 2007 NGUYEN, H. (Ed.). *GPU Gems 3*. [S.l.]: Addison Wesley, 2007. 1008 p.

Nickolls e Dally 2010 NICKOLLS, J.; DALLY, W. J. The GPU Computing Era. *IEEE Micro*, 2010. v. 30, n. 2, p. 56–69, 2010. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/MM.2010.41>.

NVCC 2012 NVCC. *NVIDIA CUDA Compiler Driver NVCC*. 2012. Disponível em: <http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html>. Acesso em: Novembro, 2012.

NVIDIA 2013 NVIDIA. *NVIDIA*. 2013. Disponível em: <http://www.nvidia.com/page/home.html>. Acesso em: Maio, 2013.

NVIDIA 2012 NVIDIA, D. P. *DYNAMIC PARALLELISM IN CUDA*. 2012. Disponível em: <http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf>. Acesso em: Dezembro, 2012.

Nystrom, Clarkson e Myers 2002 NYSTROM, N.; CLARKSON, M. R.; MYERS, A. C. *Polyglot: An Extensible Compiler Framework for Java*. [S.l.], 2002. Disponível em: <http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR2002-1883; http://hdl.handle.net/1813/5859>.

OpenMP Architecture Review Board 1997 OpenMP Architecture Review Board. *OpenMP: Simple, Portable, Scalable SMP Programming*. 1997. Disponível em: <www.openmp.org>.

Owens et al. 2008 OWENS, J.; HOUSTON, M.; LUEBKE, D.; GREEN, S.; STONE, J.; PHILLIPS, J. GPU Computing. *Proceedings of the IEEE*, 2008. v. 96, n. 5, p. 879 –899, may 2008. ISSN 0018-9219.

Papadimitriou e Steiglitz 1998 PAPADIMITRIOU, C. H.; STEIGLITZ, K. *Combinatorial Optimization : Algorithms and Complexity*. Dover Publications, 1998. Paperback. ISBN 0486402584. Disponível em: <http://www.amazon.fr/exec/obidos/ASIN/0486402584/citeulike04-21>.

Patterson e Hennessy 2008 PATTERSON, D. A.; HENNESSY, J. L. Graphics and Computing GPUs. In: *Computer organization and design: the hardware/software interface*. Fourth. pub-ELSEVIER-MORGAN-KAUFMANN:adr: Elsevier/Morgan Kaufmann, 2008. cap. A, p. A–1–A–77. ISBN 0-12-374493-8.

Pessoa et al. 2011 PESSOA, T.; MURITIBA, A.; NEGREIROS, M.; CAMPOS, G. d. A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU.

In: *Computer Architecture and High Performance Computing (SBAC-PAD), 23rd International Symposium on*. [S.l.: s.n.], 2011. p. 41–47. ISSN 1550-6533.

Pessoa 2012 PESSOA, T. C. *Estratégias Paralelas Inteligentes para o Método BRANCH-and-BOUND Aplicadas ao Problema do Caixeiro Viajante Assimétrico*. Dissertação (Mestrado) — Universidade Estadual do Ceará, 2012.

Pharr e Fernando 2005 PHARR, M.; FERNANDO, R. (Ed.). *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. [S.l.]: Addison Wesley, 2005. 880 p.

Pilla e Navaux 2010 PILLA, L. L.; NAVAUX, P. Uso da Classificação Dwarf Mine para a Avaliação Comparativa entre a Arquitetura CUDA e Multicores. *WPerformance - Workshop em Desempenho de Sistemas Computacionais e de Comunicação*, 2010. v. 9, p. 1818–1830, 2010.

Pratt-Szeliga, Fawcett e Welch 2012 PRATT-SZELIGA, P. C.; FAWCETT, J. W.; WELCH, R. D. Rootbeer: Seamlessly Using GPUs from Java. In: MIN, G.; HU, J.; LIU, L. C.; YANG, L. T.; SEELAM, S.; LEFEVRE, L. (Ed.). *HPCC-ICESS - 14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, Liverpool, United Kingdom*. IEEE Computer Society, 2012. p. 375–380. ISBN 978-1-4673-2164-8. Disponível em: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6331801>.

PTX3.1 2012 PTX3.1. *Parallel Thread execution ISA Version 3.1*. 2012. Disponível em: <http://docs.nvidia.com/cuda/pdf/ptx_isa_3.1.pdf>.

Stromme, Carlson e Newhall 2012 STROMME, A.; CARLSON, R.; NEWHALL, T. Chestnut: a gpu programming language for non-experts. In: *Proceedings of the International Workshop on Programming Models and Applications for Multicores and Manycores*. New York, NY, USA: ACM, 2012. (PMAM '12), p. 156–167. ISBN 978-1-4503-1211-0. Disponível em: <http://doi.acm.org/10.1145/2141702.2141720>.

TOP 500 2013 TOP 500. 2013. Disponível em: <www.top500.org>. Acesso em: 06 maio 2013.

Vallee-Rai 2000 VALLEE-RAI, R. *Soot : A Java Bytecode Optimization Framework*. 2000. Disponível em: <http://digitool.Library.McGill.CA:80/R/?func=dbin-jump-full&object_id=30836>.

Vallee-Rai e Hendren 1998 VALLEE-RAI, R.; HENDREN, L. J. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. 1998.

Yan, Grossman e Sarkar 2009 YAN, Y.; GROSSMAN, M.; SARKAR, V. Jcuda : A programmer-friendly interface for accelerating java programs with cuda. *EuroPar Parallel Processing*, 2009. Springer, v. 5704, p. 887–899, 2009. Disponível em: <http://www.springerlink.com/index/4282177525R45375.pdf>.

Zaremba, Lin e Grover 2012 ZAREMBA, W.; LIN, Y.; GROVER, V. Jabee: framework for object-oriented java bytecode compilation and execution on graphics processor units. In: *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. New York, NY, USA: ACM, 2012. (GPGPU-5), p. 74–83. ISBN 978-1-4503-1233-2. Disponível em: <http://doi.acm.org/10.1145/2159430.2159439>.

Apêndice A

Estudo de Caso 1

A.1 Código Java

```java
import java.util.*;
import java.io.*;
public class MatrixMultiThread {

    private int nCore, nRows;
    private int rowsA, rowsB, columnsA, columnsB;
    private int matrixA [][], matrixB [][], result [][];
    private myThread[] threadPool;

    /**Constructor of application**/
    MatrixMultiThread(int rwA, int clA, int rwB, int clB){
        nCore = Runtime.getRuntime().availableProcessors();
        System.out.println("Numero de core: "+nCore );
        rowsA = rwA;
        columnsA = clA;
        rowsB = rwB;
        columnsB = clB;

        nRows = rowsA/nCore;
        buildMatrices();
    }

    /*Expects the input to the sizes of the arrays for application*/
    public static void main(String[] args){
        Scanner scan = new Scanner(System.in);
        int arg1 = Integer.parseInt(scan.nextLine());
        int arg2 = Integer.parseInt(scan.nextLine());
        int arg3 = Integer.parseInt(scan.nextLine());
        int arg4 = Integer.parseInt(scan.nextLine());

        MatrixMultiThread mmt = new MatrixMultiThread(arg1,arg2,arg3,arg4);
        mmt.calculate();
    }

```

```
35     /*Generation of arrays*/
36     private void buildMatrices(){
37 
38         matrixA = new int [rowsA][columnsA];
39         matrixB = new int [rowsB][columnsB];
40 
41         Random generator = new Random(15);
42         for(int i=0; i<rowsA; i++){
43             for(int j=0; j<columnsA; j++){
44                 matrixA[i][j] = generator.nextInt(99)+1;
45             }
46         }
47         for(int i=0; i<rowsB; i++){
48             for(int j=0; j<columnsB; j++){
49                 matrixB[i][j] = generator.nextInt(99)+1;
50             }
51         }
52         result = new int [rowsA][columnsB];
53     }
54 
55     /*Instantiates and triggers the thread group to calculate
56     the resulting matrix*/
57     private void calculate(){
58 
59         threadPool = new myThread[nCore];
60 
61 
62         long start = System.nanoTime();
63         for(int i=0; i<threadPool.length; i++){
64             threadPool[i] = new myThread(i);
65             threadPool[i].start();
66             try{
67                 threadPool[i].join();
68             }catch (InterruptedException e){}
69         }
70 
71         long end = System.nanoTime();
72         double time = (end-start)/1000000.0;
73 
74         printResult();
75 
76         System.out.println("\n Multiplication took " + time + " milliseconds.");
77     }
78 
79     private void printResult() {
80         for(int i=0; i<rowsA; i++){
81             for(int j=0; j<columnsB; j++){
82                 System.out.print(result[i][j] + " ");
83             }
84             System.out.println();
85         }
86     }
87 
```

```
88     /* Class for instantiation of threads*/
89     private class myThread extends Thread{
90         int index;
91 
92         myThread(int index){
93             this.index = index;
94         }
95         public void run(){
96             for (int h=index*nRows; h<(index*nRows)+nRows; h++){
97                 for(int i=0; i<columnsB; i++){
98                     System.out.print(""+index);
99                     for(int j=0; j<columnsA; j++)
100                        result[h][i] += matrixA[h][j] * matrixB[j][i];
101                }
102            }
103        }
104    }
105 }
```

A.2 Código CUDA C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <cuda.h>
#include <omp.h>


void buildMatrices();
void calculate();
void printResult();

int rowsA, rowsB, columnsA, columnsB, nCore, nRows;
int size_A, size_B; size_C;
int *matrixA;
int *matrixB;
int *matrixC;

void main(){
    scanf("%d",&rowsA);
    scanf("%d",&columnsA);
    scanf("%d",&rowsB);
    scanf("%d",&columnsB);

    nCore = 2;
    nRows = rowsA/nCore;

    size_A = rowsA*columnsA*sizeof(int);
    size_B = rowsB*columnsB*sizeof(int);
    size_C = rowsA*columnsB*sizeof(int);
    //Aloca matrizes no host e inicia valores

    matrixA = (int *)malloc(size_A);
    matrixB = (int *)malloc(size_B);
    matrixC = (int *)malloc(size_C);

    buldMatrices();

    calculate();
}

/**Generation of arrays**/
void buildMatrices(){
    srand(15);
    for(int i=0; i<rowsA*columnsA; i++){
        matrixA[i] = rand(99)+1;
    }
    for(int i=0; i<rowsB*columnsB; i++){
        matrixB[i] = rand(99)+1;
    }
}

```

```
52 /**Instantiates and triggers the thread group to calculate the resulting matrix**/
53 void calculate(){
54     int *matrixA_d;
55     cudaMalloc((void **) &matrixA_d, size_A);
56     int *matrixB_d;
57     cudaMalloc((void **) &matrixB_d, size_B);
58     int *matrixC_d;
59     cudaMalloc((void **) &matrixC_d, size_C);
60 
61     cudaMemcpy(matrixB_d, matrixB, size_B*sizeof(int),cudaMemcpyHostToDevice);
62 
63     dim3 dimGrid = (1);
64     dim3 dimBlock = (nRows,columnsB);
65 
66     cudaStream_t vectorOfStreams[nCore];
67 
68     for(int stream_id=0; stream_id<nCore; stream_id++)
69         cudaStreamCreate(&vectorOfStreams[stream_id]);
70 
71     for(int stream_id=0; stream_id<nCore; stream_id++)
72         cudaMemcpyAsync(matrixA_d[nRows*stream_id], matrixA[nRows*stream_id],
73                         nRows*columnsA*sizeof(int), cudaMemcpyHostToDevice,
74                         vectorOfStreams[stream_id]);
75 
76     for(int stream_id=0; stream_id<nCore; stream_id++){
77         multiply<<<dimGrid, dimBlock, vectorOfStreams[stream_id]>>>
78                     (matrixA_d[nRows*stream_id], matrixB_d, nRows,
79                     columnsA, columnsB, stream_id);
80     }
81     for(int stream_id=0; stream_id<nCore; stream_id++){
82         cudaMemcpyAsync(matrixC[nRows*stream_id], matrixC_d[nRows*stream_id],
83                         nRows*columnsA*sizeof(int), cudaMemcpyDeviceToHost,
84                         vectorOfStreams[stream_id]);
85     }
86     cudaDeviceSynchronized();
87 
88     printResult();
89 
90     free(matrixA); cudaFree(matrixA_d);
91     free(matrixB); cudaFree(matrixB_d);
92     free(matrixC); cudaFree(matrixC_d);
93 
94 }
95 
96 __global__ multiply (int *A_d, int *B_d, int *C_d, int nRows, int ncolA, int ncolB,
97         int stream){
98     int idx = threadIdx.x;
99     int idy = threadIdx.y;
100     int aBegin = stream*nRows*ncolA;
101     int cBegin = stream*nRows*ncolB;
102     int i, j;
103 
104     __shared__ int As [nRows*ncolA];
```

```
105     int subC = 0;
106 
107     if (idx+idy==0){
108         for(i=0,j=aBegin ; i<nRows*ncolA; j++ , i++)
109             As[i] = A_d[j];
110     }
111 
112     __syncthreads();
113 
114     for(i=0; i<ncolA; i++){
115         subC += As[ncolA*idy+i] * B_d[i*ncolB+idx];
116     }
117     C_d[cBegin+idy*ncolB+idx] = subC;
118 }
119 
120 void printResult() {
121     for(int i=0; i<rowsA*columnsB; i++){
122         if(i%(columnsB) == 0)
123             printf("\n");
124         printf("%d ",matrixC[i]);
125     }
126 }
```

A.3 Código Fusion

A.3.1 Classe MatrixMultiUnits

```
import java.util.Random;
import java.util.Scanner;

public class MatrixMultiUnits {
    private int rowsA, rowsB, columnsA, columnsB;

    private int matrixA [][], matrixB [][], matrixC [][];
    private myThread[] threadPool;
    private MatrixAccel accelMulti;

    private int nCore, nRows;
    /**Constructor of application**/
    MatrixMultiUnits(int rwA, int clA, int rwB, int clB){
        rowsA = rwA;
        columnsA = clA;
        rowsB = rwB;
        columnsB = clB;
        nCore = Runtime.getRuntime().availableProcessors();
        nRows = rowsA/nCore;
        buildMatrices();
    }

    /**Expects the input to the sizes of the arrays for application**/
    public static void main(String[] args){
        Scanner scan = new Scanner(System.in);

        int arg1 = Integer.parseInt(scan.nextLine());
        int arg2 = Integer.parseInt(scan.nextLine());
        int arg3 = Integer.parseInt(scan.nextLine());
        int arg4 = Integer.parseInt(scan.nextLine());
        MatrixMultiUnits mmt = new MatrixMultiUnits(arg1,arg2,arg3,arg4);
        mmt.calculate();
    }

    /**Generation of arrays**/
    private void buildMatrices(){

        matrixA = new int [rowsA][columnsA];
        matrixB = new int [rowsB][columnsB];

        Random generator = new Random(15);
        for(int i=0; i<rowsA; i++){
            for(int j=0; j<columnsA; j++){
                matrixA[i][j] = generator.nextInt(99)+1;
            }
        }
        for(int i=0; i<rowsB; i++){
            for(int j=0; j<columnsB; j++){
                matrixB[i][j] = generator.nextInt(99)+1;
```

```
50              }
51          }
52          matrixC = new int [rowsA][columnsB];
53      }
54
55      /**Instantiates and triggers the thread group
56       * to calculate the resulting matrix**/
57      private void calculate(){
58          //Number of threads equals of the rows
59          threadPool = new myThread[rowsA];
60          //Instantiates acelerator object for link GPU
61          accelMulti = new MatrixAccel(nRows);
62          accelMulti.setMatrix(matrixA, matrixB);
63
64          long start = System.nanoTime();
65
66          for(int i=0; i<rowsA; i++){
67              threadPool[i] = new myThread(i);
68              threadPool[i].start();
69              try{
70                  threadPool[i].join();
71              }catch (InterruptedException e){
72
73              }
74          }
75
76          long end = System.nanoTime();
77          double time = (end-start)/1000000.0;
78
79          System.out.println("\n Multiplication took " + time + " milliseconds.");
80
81          matrixC = accelMulti.getResult();
82          accelMulti.clearAll();
83          printResult();
84
85      }
86
87      private void printResult() {
88          for(int i=0; i<rowsA; i++){
89              for(int j=0; j<columnsB; j++){
90                  System.out.print(result[i][j] + " ");
91              }
92              System.out.println();
93          }
94      }
95
96      /** Class for instantiation of threads that trigger units in GPU device**/
97      private static class myThread extends Thread{
98          int index;
99          MatrixAccell:Multiply accelMultiUnit
100
101         myThread(int index){
102             this.index = index;
```

```
103             accelMultiUnit = unit Multiply <nCore> for accelMulti;
104         }
105         /* call of method kernel on the acelerator object  for the partial
106         solution calculation*/
107         public void run(){
108             accelMulti.multiplyBlockLine();
109         }
110     }
111 }
```

A.3.2 Classe MatrixAccel

```
/**Accelerator class MatrixAccel*/
public accelerator class MatrixAccel {
    private int rowsA;
    private int rowsB;
    private int columnsB;
    private int columnsA;
    private int nRows;

    private global int matrixB_d [];
    private global int matrixA_d [];
    private global int matrixC_d [];
    private int matrixA_h [][];
    private int matrixB_h [][];
    private int matrixC_h [][];

    MatrixAccel(int nRows){
        this.nRows = nRows;
    }
    /*receiving two arrays performs the allocation both and
    copy of matrixB to the device*/
    public void setMatrix(int matrixA_h [][], int matrixB_h [][]){
        this.matrixA_h = matrixA_h;
        this.matrixB_h = matrixB_h;
        matrixC_h = new int [rowsA][columnsB];

        rowsA = matrixA_h.length;
        columnsA = matrixA_h[0].length;
        rowsB = matrixB_h.length;
        columnsB = matrixB_h[0].length;

        matrixA_d = new int [rowsA*columnsA];
        matrixB_d = new int [rowsB*columnsB];
        matrixC_d = new int [rowsA*columnsB];

        cpyMatrixB_Device();
    }
    /**Copy matrixB on to device*/
    public void cpyMatrixB_Device(){
        memCpy(matrixB_d, matrixB_h);
    }
    /**Return matrixC_h on to application*/
    public int [][] getResult (){
        getResultUnit();
        synchronized(device);
        return matrixC_h;
    }
    /**Parallel unit responsable for partial solution*/
    parallel unit Multiply {
        multiply(){
            int rank = getRank();
            memCpyA(matrixA_d, columnsA*rank,
```

```
52                      matrixA_h, columnsA*rank,
53                      columnsA,rank);
54          }
55
56          public parallel kernel multiplyBlockLine() threads<nRows,columnsB> blocks<1>{
57              int idx = threadIdx(x);
58              int idy = threadIdx(y);
59              int aBegin = rank*nRows*columnsA;
60              int cBegin = rank*nRows*columnsB;
61              int i, j;
62              shared int As [nRows*columnsA];
63              int subC = 0;
64              if (idx+idy==0){
65                  for(i=0,j=aBegin ; i<nRows*columnsA; j++ , i++)
66                      As[i] = matrixA_d[j];
67              }
68              synchronized(threads);
69              for(i=0; i<columnsA; i++){
70                  subC += As[columnsA*idy+i] * matrixB_d[i*columnsB+idx];
71              }
72              matrixC_d[cBegin+idy*columnsB+idx] = subC;
73          }
74
75          public parallel void getResultUnit(){
76              memCpyA(matrixC_h, columnsB*nRows*rank,
77                      matrixC_d, columnsB*nRows*rank,
78                      columnsB*nRows, rank);
79          }
80      }
81  }
```

Apêndice B

Estudo de Caso 2

B.1 Código CUDA C

```
/*
 * main.c
 *
 *  Created on: 26/01/2011
 *      Author: einstein/carneiro
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <cuda.h>
#include <omp.h>

#define mat(i,j) mat_h[i*N+j]
#define mat_h(i,j) mat_h[i*N+j]
#define mat_d(i,j) mat_d[i*N_l+j]
#define mat_block(i,j) mat_block[i*N_l+j]
#define proximo(x) x+1
#define anterior(x) x-1
#define MAX 8192
#define INFINITO 999999
#define ZERO 0
#define ONE 1

#define _VAZIO_      -1
#define _VISITADO_    1
#define _NAO_VISITADO_ 0

int qtd = 0;
int custo = 0;
int N;
int melhor = INFINITO;
int upper_bound;

int mat_h[MAX];
```

```
35
36
37 #define CUDA_CHECK_RETURN(value) {                                          \
38     cudaError_t _m_cudaStat = value;                                        \
39     if (_m_cudaStat != cudaSuccess) {                                       \
40         fprintf(stderr, "Error %s at line %d in file %s\n",                 \
41                 cudaGetErrorString(_m_cudaStat), __LINE__, __FILE__);       \
42         exit(1);                                                            \
43     } }
44
45
46 static void HandleError( cudaError_t err,
47         const char *file,
48         int line ) {
49     if (err != cudaSuccess) {
50         printf( "%s in %s at line %d\n", cudaGetErrorString( err ),
51                 file, line );
52         exit( EXIT_FAILURE );
53     }
54 }
55
56
57
58 #define HANDLE_NULL( a ) {if (a == NULL) { \
59         printf( "Host memory failed in %s at line %d\n", \
60                 __FILE__, __LINE__ ); \
61                 exit( EXIT_FAILURE );}}
62
63 void read() {
64     int i;
65     scanf("%d", &N);
66     for (i = 0; i < (N * N); i++) {
67         scanf("%d", &mat_h[i]);
68     }
69
70 }
71
72 unsigned long long int calculaNPrefixos(const int nivelPrefixo, const int nVertice) {
73     unsigned long long int x = nVertice - 1;
74     int i;
75     for (i = 1; i < nivelPrefixo-1; ++i) {
76         x *= nVertice - i-1;
77     }
78     return x;
79 }
80
81 void fillFixedPaths(short* preFixo, int nivelPrefixo) {
82     char flag[50];
83     int vertice[50];
84     int cont = 0;
85     int i, nivel;
86
87
```

```
88      for (i = 0; i < N; ++i) {
89          flag[i] = 0;
90          vertice[i] = -1;
91      }
92
93      vertice[0] = 0;
94      flag[0] = 1;
95      nivel = 1;
96      while (nivel >= 1){
97
98          if (vertice[nivel] != -1) {
99              flag[vertice[nivel]] = 0;
100         }
101
102         do {
103             vertice[nivel]++;
104         } while (vertice[nivel] < N && flag[vertice[nivel]]);
105
106         if (vertice[nivel] < N) {
107             flag[vertice[nivel]] = 1;
108             nivel++;
109             if (nivel == nivelPrefixo) {
110                 for (i = 0; i < nivelPrefixo; ++i) {
111                     preFixo[cont * nivelPrefixo + i] = vertice[i];
112                 }
113                 cont++;
114                 nivel--;
115             }
116         } else {
117             vertice[nivel] = -1;
118             nivel--;
119         }
120     }
121 }
122
123
124 __global__ void dfs_cuda_UB_stream(int N,int stream_size, int *mat_d,
125     short *preFixos_d, int nivelPrefixo, int upper_bound, int *sols_d,
126     int *melhorSol_d)
127     {
128
129     register int idx = blockIdx.x * blockDim.x + threadIdx.x;
130     register int flag[16];
131     register int vertice[16];
132
133     register int N_l = N;
134
135     register int i, nivel;
136     register int custo;
137     register int qtd_solucoes_thread = 0;
138     register int UB_local = upper_bound;
139     register int nivelGlobal = nivelPrefixo;
140     int stream_size_l = stream_size;
```

```
141
142     if (idx < stream_size_l) {
143
144         for (i = 0; i < N_l; ++i) {
145             vertice[i] = _VAZIO_;
146             flag[i] = _NAO_VISITADO_;
147         }
148
149         vertice[0] = 0;
150         flag[0] = _VISITADO_;
151         custo= ZERO;
152
153         for (i = 1; i < nivelGlobal; ++i) {
154
155             vertice[i] = preFixos_d[idx * nivelGlobal + i];
156
157             flag[vertice[i]] = _VISITADO_;
158             custo += mat_d(vertice[i-1],vertice[i]);
159         }
160
161         nivel=nivelGlobal;
162
163         while (nivel >= nivelGlobal ) {
164             if (vertice[nivel] != _VAZIO_) {
165                 flag[vertice[nivel]] = _NAO_VISITADO_;
166                 custo -= mat_d(vertice[anterior(nivel)],vertice[nivel]);
167             }
168
169             do {
170                 vertice[nivel]++;
171             } while (vertice[nivel] < N_l && flag[vertice[nivel]]);
172
173             if (vertice[nivel] < N_l) {
174                 custo += mat_d(vertice[anterior(nivel)],vertice[nivel]);
175                 flag[vertice[nivel]] = _VISITADO_;
176                 nivel++;
177
178                 if (nivel == N_l) {
179                     ++qtd_solucoes_thread;
180                     if (custo + mat_d(vertice[anterior(nivel)],0) < UB_local) {
181                         UB_local = custo + mat_d(vertice[anterior(nivel)],0);
182                     }
183                     nivel--;
184                 }
185             }
186             else {
187                 vertice[nivel] = _VAZIO_;
188                 nivel--;
189             }
190         }
191         sols_d[idx] = qtd_solucoes_thread;
192         melhorSol_d[idx] = UB_local;
193     }
```

```
194 }
195
196 void checkCUDAError(const char *msg) {
197     cudaError_t err = cudaGetLastError();
198     if (cudaSuccess != err) {
199         fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString(err));
200         exit(EXIT_FAILURE);
201     }
202 }
203
204
205 int callCompleteEnumStreams(const int nivelPreFixos){
206
207     int *mat_d;
208     int otimo_global = INFINITO;
209     int *qtd_threads_streams;
210     int qtd_sols_global = 0;
211     int nPreFixos = calculaNPrefixos(nivelPreFixos,N);
212     int block_size =192;
213
214     int *sols_h, *sols_d;
215     int *melhorSol_h, *melhorSol_d;
216
217     short * path_h = (short*) malloc(sizeof(short) * nPreFixos * nivelPreFixos);
218     short * path_d;
219
220     /* Variaveis para os streams*/
221     const int chunk = 192*10;
222     const int numStreams = nPreFixos / chunk + (nPreFixos % chunk == 0 ? 0 : 1);
223     const int num_blocks = chunk/block_size + (chunk % block_size == 0 ? 0 : 1);
224     int resto = 0;
225
226     resto = (nPreFixos % chunk);
227
228     qtd_threads_streams = (int*)malloc(sizeof(int)*numStreams);
229
230     /*
231      * Setando qtd de threads do stream
232      * */
233     if(numStreams>1){
234         for(int i = 0; i<numStreams-1 / block_size;++i){
235             qtd_threads_streams[i] = chunk;
236         }
237         if(resto>0){
238             qtd_threads_streams[numStreams-1] = resto;
239         }
240     }
241     else
242         qtd_threads_streams[0] = resto;
243
244     CUDA_CHECK_RETURN( cudaMalloc((void **) &path_d,
245                         nPreFixos*nivelPreFixos*sizeof(short)));
246     sols_h = (int*)malloc(sizeof(int)*nPreFixos);
```

```
247     melhorSol_h = (int*)malloc(sizeof(int)*nPreFixos);
248
249     CUDA_CHECK_RETURN( cudaMalloc((void **) &mat_d, N * N * sizeof(int)));
250
251     fillFixedPaths(path_h, nivelPreFixos);
252
253     CUDA_CHECK_RETURN( cudaMemcpy(mat_d, mat_h, N * N * sizeof(int),
254                        cudaMemcpyHostToDevice));
255     for(int i = 0; i<nPreFixos; ++i)
256         melhorSol_h[i] = INFINITO;
257
258
259     CUDA_CHECK_RETURN( cudaMalloc((void **) &melhorSol_d, sizeof(int)*nPreFixos));
260     CUDA_CHECK_RETURN( cudaMalloc((void **) &sols_d, sizeof(int)*nPreFixos));
261
262     cudaStream_t vectorOfStreams[numStreams];
263
264     for(int stream_id=0; stream_id<numStreams; stream_id++)
265             cudaStreamCreate(&vectorOfStreams[stream_id]);
266
267     for(int stream_id=0; stream_id<numStreams; stream_id++)
268          cudaMemcpyAsync(&path_d[stream_id*chunk*nivelPreFixos],
269                         &path_h[stream_id*chunk*nivelPreFixos],
270                         qtd_threads_streams[stream_id]*sizeof(short)*nivelPreFixos,
271                         cudaMemcpyHostToDevice,vectorOfStreams[stream_id]);
272
273     for(int stream_id=0; stream_id<numStreams; stream_id++){
274         cudaMemcpyAsync(&melhorSol_d[stream_id*chunk], &melhorSol_h[stream_id*chunk],
275                         qtd_threads_streams[stream_id]*sizeof(int),
276                         cudaMemcpyHostToDevice, vectorOfStreams[stream_id]);
277
278     }
279     for(int stream_id=0; stream_id<numStreams; stream_id++){
280         cudaMemcpyAsync(&sols_d[stream_id*chunk], &sols_h[stream_id*chunk],
281                         qtd_threads_streams[stream_id]*sizeof(int),
282                         cudaMemcpyHostToDevice,vectorOfStreams[stream_id]);
283     }
284
285     for(int stream_id=0; stream_id<numStreams; stream_id++){
286         dfs_cuda_UB_stream<<<num_blocks,block_size,0,vectorOfStreams[stream_id]>>>
287                         (N,qtd_threads_streams[stream_id],mat_d,
288                         &path_d[stream_id*chunk*nivelPreFixos], nivelPreFixos,999999,
289                         &sols_d[stream_id*chunk],&melhorSol_d[stream_id*chunk]);
290     }
291
292     for(int stream_id=0; stream_id<numStreams; stream_id++)
293         cudaMemcpyAsync(&sols_h[stream_id*chunk],&sols_d[stream_id*chunk],
294                         qtd_threads_streams[stream_id]*sizeof(int),
295                         cudaMemcpyDeviceToHost,vectorOfStreams[stream_id]);
296
297     for(int stream_id=0;stream_id<numStreams; stream_id++)
298         cudaMemcpyAsync(&melhorSol_h[stream_id*chunk],&melhorSol_d[stream_id*chunk],
299                         qtd_threads_streams[stream_id]*sizeof(int),
```

```
300                             cudaMemcpyDeviceToHost,vectorOfStreams[stream_id]);
301
302     cudaDeviceSynchronize();
303
304     for(int i = 0; i<nPreFixos; ++i){
305             qtd_sols_global+=sols_h[i];
306             if(melhorSol_h[i]<otimo_global)
307                 otimo_global = melhorSol_h[i];
308     }
309
310     printf("\n\n\n\t niveis preenchidos: %d.\n",nivelPreFixos);
311
312     printf("\t Numero de streams: %d.\n",numStreams);
313     printf("\t Tamanho do stream: %d.\n",chunk);
314     printf("\nQuantidade de solucoes encontradas: %d.", qtd_sols_global);
315     printf("\n\tOtimo global: %d.\n\n", otimo_global);
316
317     CUDA_CHECK_RETURN( cudaFree(mat_d));
318     CUDA_CHECK_RETURN( cudaFree(sols_d));
319     CUDA_CHECK_RETURN( cudaFree(path_d));
320     CUDA_CHECK_RETURN( cudaFree(melhorSol_d));
321
322     return otimo_global;
323 }
324
325 int main() {
326
327     read();
328     int niveis = 5;
329     printf("\n\nEnumeracao com streams:\n\n");
330     callCompleteEnumStreams(niveis);
331
332     return 0;
333 }
```

B.2 Código Fusion

B.2.1 Classe Application

```
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Scanner;
import java.util.concurrent.ForkJoinPool;

public class Application {
    static final int MAX = 8192;
    static final int INFINITO = 999999;
    private Scanner scan = new Scanner(System.in);
    private int mat_h [] = new int [MAX];
    private int N;
    private int chunk;
    private int qtd_threads_streams [];
    private int sols_h [];
    private int melhorSol_h [];
    private EnumerationAccel enumeration;

    Application(){}

    public void read() {
        int i;
        N = scan.nextInt();
        for (i = 0; i < (N * N); i++) {
            mat_h[i] = scan.nextInt();
        }
    }

    int calculateNPrefixos(int nivelPrefixo, int nVertice) {
        int x = nVertice - 1;
        int i;
        for (i = 1; i < nivelPrefixo-1; ++i) {
            x *= nVertice - i-1;
        }
        return x;
    }

    void fillFixedPaths(short preFixo[], int nivelPrefixo) {
        boolean flag[] = new boolean[50];
        int vertice[] = new int[50];
        int cont = 0;
        int i, nivel;
        for (i = 0; i < N; ++i) {
            flag[i] = false;
            vertice[i] = -1;
        }

        vertice[0] = 0;
        flag[0] = true;
        nivel = 1;
```

```
50          while (nivel >= 1) {
51              if (vertice[nivel] != -1) {
52                  flag[vertice[nivel]] = false;
53              }
54
55              do {
56                  vertice[nivel]++;
57              } while (vertice[nivel] < N && flag[vertice[nivel]]); //
58
59              if (vertice[nivel] < N) {
60                  flag[vertice[nivel]] = true;
61                  nivel++;
62                  if (nivel == nivelPrefixo) {
63                      for (i = 0; i < nivelPrefixo; ++i) {
64                          preFixo[cont * nivelPrefixo + i] = (short) vertice[i];
65                  }
66                      cont++;
67                      nivel--;
68                  }
69              } else {
70                  vertice[nivel] = -1;
71                  nivel--;
72              }
73          }
74      }
75      public void callCompleteEnumStreams (int nivelPreFixos){
76          int otimo_global = INFINITO;
77          int qtd_sols_global = 0;
78          int nPreFixos = calculateNPrefixos(nivelPreFixos,N);
79          int block_size =192;
80
81          short path_h [] = new short [nPreFixos*nivelPreFixos];
82
83          chunk = 192*10;
84          int numStreams = nPreFixos / chunk + (nPreFixos % chunk == 0 ? 0 : 1);
85          int num_blocks = chunk/block_size + (chunk % block_size == 0 ? 0 : 1);
86          int resto = 0;
87
88          resto = (nPreFixos % chunk);
89
90          qtd_threads_streams = new int [numStreams];
91
92          if(numStreams >1){
93              for(int i = 0; i<numStreams-1 / block_size;++i){
94                  qtd_threads_streams[i] = chunk;
95              }
96              if(resto>0){
97                  qtd_threads_streams[numStreams-1] = resto;
98              }
99          }
100         else
101             qtd_threads_streams[0] = resto;
102
```

```
103         int sols_h [] = new int[nPreFixos];
104         int melhorSol_h [] = new int[nPreFixos];
105 
106         fillFixedPaths(path_h, nivelPreFixos);
107 
108         for(int i = 0; i<nPreFixos; ++i)
109             melhorSol_h[i] = INFINITO;
110         int size = numStreams;
111         enumeration = new EnumerationAccel<0> (chunk, nPreFixos, N, nivelPreFixos,
112             mat_h, path_h, melhorSol_h, sols_h, size, qtd_threads_streams);
113 
114         ForkJoinPool forkJoinPool = new ForkJoinPool(size);
115         forkJoinPool.invoke(new TaskEnumeration(this, enumaration) );
116 
117         qtd_sols_global = enumeration.getQtdSols();
118         otimo_global = enumeration.getOtimoGlobal();
119 
120         System.out.println("\n\n\n\t niveis preenchidos: \n"+nivelPreFixos
121                 +"\tNumero de streams:\n"+numStreams+"\t Tamanho do stream:\n"+chunk
122                 +"\nQuantidade de solucoes encontradas:"+ qtd_sols_global
123                 +"\n\tOtimo global: %d.\n\n"+otimo_global);
124 
125         enumeration.clearAll();
126 
127     }
128 
129     public static void main(String args[]) {
130         Application app = new Application ();
131         app.read();
132         int niveis = 5;
133         System.out.println("\n\nEnumeracao com streams:\n\n");
134         app.callCompleteEnumStreams(niveis);
135     }
136 }
```

B.2.2 Classe TaskEnumaration

```
1
2 import java.util.concurrent.RecursiveTask;
3
4
5 public class TaskEnumeration extends RecursiveTask<Double>{
6     private EnumerationAccel:Enumarate accel_obj_unit;
7     private int rank;
8
9     TaskEnumeration(EnumerationAccel accel_obj) {
10        super();
11        accel_obj_unit = unit Enumerate<size> for accel_obj;
12    }
13
14    @Override
15    protected Object compute() {
16        accel_obj_unit.dfs();
17        return null;
18    }
19 }
```

B.2.3 Classe EnumerationAccel

```
public accelerator class EnumerationAccel {
    static final int INFINITO = 999999;
    static final int ZERO = 0;
    static final int ONE = 1;
    static final int _VAZIO_= -1;
    static final boolean _VISITADO_= true;
    static final boolean _NAO_VISITADO_= false;
    int qtd = 0;
    int custo = 0;
    int melhor = INFINITO;
    int upper_bound;

    private int chunk;
    private int nPreFixos;
    private int nn;
    private int N;
    private int nivelPreFixos;
    private global int mat_d [];
    private global int path_d [];
    private int path_h [];
    private global int melhorSol_d [];
    private int melhorSol_h [];
    private global int sols_d [];
    private int sols_h[];
    private int size;
    private int qtd_threads_streams[];

    /*Constructor Enumarate class*/
    EnumerationAccel(int chunk, int nPreFixos, int N,
            int nivelPrefixos, int mat_h[], int path_h[],
            int melhorSol_h[], int sols_h[], int size,
            int qtd_threads_streams[]){
        this.chunk = chunk;
        this.nPreFixos = nPreFixos;
        this.nn = (int) Math.pow(N,2);
        this.N = N;
        this.nivelPreFixos = nivelPreFixos;
        this.mat_d = new int [nn];
        memCpy(mat_d, mat_h, nn);
        this.path_d = new int [nPreFixos*nivelPreFixos];
        this.path_h = path_h;
        this.melhorSol_d = new int [nPreFixos];
        this.melhorSol_h = melhorSol_h;
        this.sols_d = new int [nPreFixos];
        this.size = size;
        this.qtd_threads_streams = qtd_threads_streams;

    }

    /*Host Method getQtdSols*/
    public int getQtdSols(){
```

```
52          int qtd_sols_global = 0;
53          getSolUnit();
54          for(int i = 0; i<nPreFixos; ++i){
55              qtd_sols_global+=sols_h[i];
56          }
57          return qtd_sols_global;
58      }
59      /*Host Method getOtimoGlobal*/
60      public int getOtimoGlobal (){
61          int otimo_global = INFINITO;
62          getMelhorSolUnit();
63          synchronized(device(0));
64          for(int i = 0; i<nPreFixos; ++i){
65              if(melhorSol_h[i]<otimo_global)
66                  otimo_global = melhorSol_h[i];
67          }
68          return otimo_global;
69      }
70      /*Parallel Unit Enumarate*/
71      parallel unit Enumerate {
72          Enumerate() {
73              int rank = rank();
74              memCpyA(path_d, rank*chunk*nivelPreFixos, path_h,
75                      rank*chunk*nivelPreFixos,qtd_threads_streams[rank]*nivelPreFixos);
76              memCpyA(melhorSol_d, rank*chunk, melhorSol_h, rank*chunk,
77                      qtd_threads_streams[rank]);
78              memCpyA(sols_d,rank*chunk, sols_h,rank*chunk, qtd_threads_streams[rank]);
79
80          }
81          /*Parallel kernel method dfs*/
82          public parallel kernel dfs (int upper_bound)
83              threads<192>
84              blocks<nPreFixos/numThreads(x)+(nPreFixos % numThreads(x) == 0 ? 0 : 1)>
85          {
86            int stream_size = qtd_threads_streams[rank];
87            int idx = blockIdx(x) * blockDim(x) + threadIdx(x);
88            boolean flag[] = new boolean [16];
89            int vertice [] = new int [16];
90
91            int N_l = N;
92
93            int i, nivel;
94            int custo;
95            int qtd_solucoes_thread = 0;
96            int UB_local = upper_bound;
97            int nivelGlobal = nivelPreFixos;
98            int stream_size_l = stream_size;
99
100           if (idx < stream_size_l) {
101               for (i = 0; i < N_l; ++i) {
102                   vertice[i] = _VAZIO_;
103                   flag[i] = _NAO_VISITADO_;
104               }
```

```
105                 vertice[0] = 0;
106                 flag[0] = _VISITADO_;
107                 custo= ZERO;
108                 for (i = 1; i < nivelGlobal; ++i) {
109                     vertice[i] = path_d[idx * nivelGlobal + i];
110                     flag[vertice[i]] = _VISITADO_;
111                     custo += mat_d[vertice[i-1]*N_l+vertice[i]];
112                 }
113                 nivel=nivelGlobal;
114                 while (nivel >= nivelGlobal ) {
115                     if (vertice[nivel] != _VAZIO_) {
116                         flag[vertice[nivel]] = _NAO_VISITADO_;
117                         custo -= mat_d[vertice[nivel-1] * N_l + vertice[nivel]];
118                     }
119                     do {
120                         vertice[nivel]++;
121                     } while ((vertice[nivel] < N_l)||flag[vertice[nivel]] );
122                     if (vertice[nivel] < N_l) {
123                         custo += mat_d[vertice[nivel-1] * N_l + vertice[nivel]];
124                         flag[vertice[nivel]] = _VISITADO_;
125                         nivel++;
126                         if (nivel == N_l) {
127                             ++qtd_solucoes_thread;
128                             if (custo + mat_d[vertice[nivel-1] * N_l + 0] < UB_local){
129                                 UB_local = custo + mat_d[vertice[nivel-1]* N_l + 0];
130                             }
131                             nivel--;
132                         }
133                     }
134                     else {
135                         vertice[nivel] = _VAZIO_;
136                         nivel--;
137                     }
138                 }
139                 sols_d[idx] = qtd_solucoes_thread;
140                 melhorSol_d[idx] = UB_local;
141             }
142         }
143         /*Parallel host method dfs*/
144         void parallel getMelhorSolUnit(){
145             memCpyA(melhorSol_h, rank*chunk, melhorSol_d, rank*chunk,
146                     qtd_threads_streams[rank]);
147         }
148         /*Parallel host method dfs*/
149         void parallel getSolUnit(){
150             memCpyA(sols_h,rank*chunk, sols_d,rank*chunk, qtd_threads_streams[rank]);
151         }
152     }
153 }
```