UTILIZAÇÃO DE TÉCNICAS DE ANÁLISE ESTÁTICA E DINÂMICA PARA OTIMIZAÇÃO DE APLICAÇÕES DE PROPÓSITO GERAL EM GPUS

BRUNO ROCHA COUTINHO

UTILIZAÇÃO DE TÉCNICAS DE ANÁLISE ESTÁTICA E DINÂMICA PARA OTIMIZAÇÃO DE APLICAÇÕES DE PROPÓSITO GERAL EM GPUS

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Orientador: Wagner Meira Junior
Coorientador: Fernando Magno Quintão Pereira

Belo Horizonte
Junho de 2011

Coutinho, Bruno Rocha.

C871u Utilização de técnicas de análise estática e dinâmica para otimização de aplicações de propósito geral em GPUs / Bruno Rocha Coutinho — Belo Horizonte, 2011.
xx, 102 f.: il.; 29cm.

Tese (doutorado) — Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Wagner Meira Junior
Coorientador: Fernando Magno Quintão Pereira

1. Computação – Teses. 2. Computação gráfica – Teses. 3. Processamento paralelo (Computadores) – Teses.
I. Orientador. II Coorientador. III Título.

519.6*83(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Utilização de técnicas de análise estática e dinâmica para otimização de aplicações de propósito geral em GPUs

BRUNO ROCHA COUTINHO

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. WAGNER MEIRA JÚNIOR - Orientador
Departamento de Ciência da Computação - UFMG

PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Co-orientador
Departamento de Ciência da Computação - UFMG

PROF. RENATO ANTÔNIO CELSO FERREIRA
Departamento de Ciência da Computação - UFMG

PROFA. MARIZA ANDRADE DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

PROF. ALBERTO FERREIRA DE SOUZA
Departamento de Informática - UFES

PROF. PHILIPPE OLIVIER ALEXANDRE NAVAUX
Instituto de Informática - UFRGS

Belo Horizonte, 06 de julho de 2011.

Resumo

Temos assistido a um grande avanço no uso de dispositivos projetados especificamente para a execução de aplicações paralelas. Exemplos desta classe de *hardware* são as placas de processamento gráfico, como as GPUs produzidas pela NVIDIA. A utilização de GPUs para acelerar aplicações de propósito geral deu origem a uma nova linha de desenvolvimento de aplicações paralelas, popularmente conhecida como GPGPU. Embora recentes, aplicações GPGPU já foram utilizadas para aumentar o desempenho de programas tradicionais em até 100 vezes. Contudo, a programação nestes ambientes apresenta diversos desafios, em decorrência de fatores como o complicado arranjo de memória, a grande flexibilidade de escalonamento de *threads* e a própria natureza das aplicações que precisam ser paralelizadas, as quais, muitas vezes, não se encaixam no modelo de programação SIMD das GPUs. A fim de mitigar estes desafios, permitindo que desenvolvedores não especialistas possam usufruir do grande poder computacional no ambiente GPGPU, nós propomos novas técnicas para a avaliação de desempenho e otimização neste ambiente. Nossas técnicas são baseadas no uso conjunto de análises estáticas e dinâmicas de programas. As técnicas de análise estática provêem ao desenvolvedor uma visão global da aplicação paralela; porém, esta visão é muitas vezes desnecessariamente conservadora. Por outro lado, as técnicas de análise dinâmica fornecem ao programador uma visão precisa da aplicação sob análise; contudo, esta visão é local, e depende de fatores tais como os dados de entrada e o ambiente de execução. O intuito do presente trabalho é combinar esses dois universos de análise de programas a fim de que desenvolvedores possam obter mais informações do que seria possível usando cada estratégia em separado.

Abstract

We have witnessed large advances in the use of devices designed specifically for running parallel applications. Examples of this class of hardware are graphics processing cards, as those produced by NVIDIA. The use of GPUs to accelerate general purpose applications gave rise to a new line of parallel applications development, popularly known as GPGPU. Although recent, GPGPU applications achieved performance up to 100 times faster then traditional programs. However, programming in these environments presents several challenges, due to factors such as complex memory organization, large flexibility when scheduling threads and the very nature of applications that need to be parallelized, which often is not fit into the GPU's SIMD programming model. In order to mitigate these challenges, allowing non-specialist developers to take advantage of the large computing power of the GPGPU environment, we propose new techniques for performance evaluation and optimization in this environment. Our techniques are based on the combined use of static and dynamic code analysis. Static analysis techniques provide the developer with a global vision of the parallel application, but this view is often unnecessarily conservative. On the other hand, dynamic analysis techniques provide a accurate view of the application under review, however this view is local and depends on factors such as the code input and execution environment. We hope that these two universes of code analysis can be combined so that developers can obtain more information than is possible using each strategy separately.

Lista de Figuras

Lista de Tabelas

List of Algoritmos

Sumário

Capítulo 1

Introdução

Existe uma constante busca por desempenho em Ciência da Computação. Este tão necessário desempenho pode ser alcançado via novas tecnologias de hardware. A Lei de Moore, reza que, a cada 18 meses, o número de transístores por unidade de área dobra de tamanho. Como um corolário, temos que o numero de operações matemáticas executadas por unidade de tempo destes processadores tende também tende a dobrar neste período de tempo [Asanovic et al., 2006]. Infelizmente, este corolário não tem sido observado em processadores mais recentes. Em particular, desde 2004-2005, temos assistido a uma dramática desaceleração do número de operações por segundo de um núcleo de processamento, quando comparada com o número de transístores que ele possui [Asanovic et al., 2006]. Uma saída para este problema é a computação paralela.

A computação paralela sempre foi um alvo de pesquisa entre os grupos de desenvolvimento de software e hardware em todo o mundo. Existem diversas formas de se obter paralelismo. Por exemplo, podemos equipar computadores com múltiplos núcleos de processamento, ou podemos interligar vários computadores de forma que eles trabalhem em conjunto. Tais computadores podem estar conectados por redes locais muito rápidas, ou mesmo por redes distribuídas em um vasto espaço geográfico. Atualmente um novo elemento passou a fazer parte do leque de possibilidades do desenvolvedor de software paralelo: as *placas gráficas*, ou GPUs, como são comumente conhecidas.

As primeiras GPUs eram dispositivos de função fixa, ou seja executavam sempre as mesmas operações (geração de imagens em um buffer, para exibição em um monitor), variando-se apenas a descrição da imagem [Buck et al., 2004]. A partir de 2001 elas passaram a contar com unidades programáveis bastante simples, chamadas de *shaders*. Porém essas unidades tinham muitas limitações quanto aos dados que podiam acessar (geralmente somente os dados produzidos pelo estágio anterior do pipeline) e as instruções que podiam executar, sendo que inicialmente não eram Turing

completas [Microsoft, 2011a, Microsoft, 2011b]. Em 2006 foi criada uma GPU com unidades de processamento unificadas, ou seja, capazes de executar código de qualquer tipo de *shader*. Deste modo, os pesquisadores começaram a verificar que as GPUs se tornaram muito flexíveis e que poderiam usar esse hardware para acelerar aplicações paralelas fora do contexto da computação gráfica. Isso levou à crescente popularidade de ambientes de programação para GPUs, notadamente CUDA [CUDA, 2011] e OpenCL [Khronos OpenCL Working Group, 2008].

Executar aplicações de propósito geral em GPUs é atrativo porque esses processadores são massivamente paralelos. Por exemplo, a Nvidia Geforce GTX 580 tem 16 multiprocessadores, construídos de 32 unidades de processamento e cada um sendo capaz de executar 1536 *threads*. Esse tipo de hardware permite que uma aplicação em GPU execute até 100 vezes mais rápido que uma aplicação não otimizada [Ryoo et al., 2008a] para CPU e até 14 vezes mais rápido que a melhor implementação em CPU [Lee et al., 2010]. Essa discrepância tende a continuar à medida que hardware mais novo integra melhor CPU e GPU [Seiler et al., 2008] e novos modelos de hardware heterogêneo são introduzidos [Saha et al., 2009].

GPUs são massivamente paralelas; contudo, devido a restrições em sua arquitetura, nem toda aplicação pode se beneficiar do seu poder computacional. Nesses processadores, as *threads* são organizadas em grupos que executam em *lock-step*, chamados *warps* no jargão da Nvidia, ou *wavefronts*, no jargão da AMD. Essa arquitetura, um dos elementos básicos da taxonomia de Flynn [Flynn, 1972], é conhecido como *Single Instruction Stream, Multiple Data Stream* ou SIMD. Para entender melhor as regras da execução de *threads* do mesmo *warp*, podemos imaginar que exite um decodificador de instruções que a cada ciclo decodifica uma instrução e a despacha para várias unidades de execução, onde cada unidade de execução executará a instrução sobre os dados de uma *thread* diferente. Por exemplo, a Geforce GTX 580 possui 16 unidades de processamento SIMD chamadas de *CUDA cores*, cada uma capaz de executar 48 *warps* de 32 *threads*. Como a frequência da unidade de controle é metade da unidade de execução, um *CUDA core* têm duas unidades de controle (um para os *warps* pares e outro para os ímpares) onde cada uma envia uma instrução para ser executada por 16 unidades de execução em 2 ciclos.

Essa forma de implementação do hardware permite economizar espaço que seria utilizado com unidades de controle, mas restringe todas as *threads* do *warp* a executar a mesma instrução. Aplicações regulares como as que trabalham com matrizes [Asanovic et al., 2006] executam muito bem em GPUs, pois temos sempre a mesma operação sendo aplicada de forma independente em dados diferentes. Contudo, nem toda aplicação é tão regular e eventualmente *threads* de um mesmo *warp* podem to-

mar caminhos diferentes em um desvio. Quando isso acontece, dizemos que elas são divergentes.

Como o decodificador de instruções só pode despachar uma instrução de cada vez, unidades que executam as *threads* que foram para um dos caminhos vão ter que esperar enquanto as unidades responsáveis pelas *threads* que foram para o outro executam. Assim, as divergências podem ser uma grande fonte de degradação de desempenho. Por exemplo, Baghsorkhi *et al.*, usando um modelo analítico do desempenho das GPUs da Nvidia, descobriram que um terço do tempo de execução de uma aplicação do CUDA SDK (software development kit) é desperdiçado com divergências [Baghsorkhi et al., 2010].

Otimizações de código para evitar divergências são difíceis por várias razões. Primeiro, alguns algoritmos são inerentemente divergentes e suas *threads* naturalmente não irão concordar no caminho a ser tomado em desvios. Segundo, procurar desvios divergentes força o desenvolvedor a lidar com uma tarefa tediosa, que pode necessitar de entendimento aprofundado de código que pode ser grande e difícil de entender. São tais problemas que este presente trabalho visa mitigar.

1.1 Tese

É fato notório que as linguagens de programação tornam-se cada vez mais expressivas, disponibilizando aos desenvolvedores abstrações tais como orientação por objetos, funções de alta ordem, tipos genéricos. Por outro lado, o *hardware* em que programas escritos em tais linguagens serão executados torna-se cada vez mais complexo, por exemplo, provendo *pipelines* mais longos, mais núcleos de execução, instruções mais poderosas. Arquiteturas SIMD são um caso particular de *hardware* complexo.

Programadores normalmente confiam em compiladores otimizadores ou *profilers* de programas para que suas aplicações possam obter o máximo de desempenho do *hardware*. Nós acreditamos que ambientes de execução SIMD possuem caraterísticas tão peculiares que levam ao desenvolvimento de técnicas de compilação e *profiling* que não são normalmente vistas fora deste ambiente. Assim, podemos declarar a nossa tese da seguinte forma:

> A otimização de aplicações de propósito geral em GPUs demanda o desenvolvimento de novas técnicas de análises estáticas e dinâmicas, bem como a sua integração, em particular para analisar e otimizar divergências.

Esta tese pode ser dividida em quatro objetivos específicos, listados abaixo:

1. O modelo SIMD demanda técnicas de *profiling* diferentes das técnicas utilizadas no *profiling* de aplicações sequencias.

2. Um *profiler* construído especificamente para mensurar divergências em programas SIMD é uma ferramenta extremamente útil para ajudar os desenvolvedores a construírem código mais eficiente.

3. O modelo SIMD requer técnicas de otimização de código diferentes das técnicas de otimização e análise tradicionais.

4. É possível obter ganhos reais de desempenho via otimizações automáticas construídas especificamente para o modelo SIMD.

Para demonstrar a tese, submetemos várias aplicações presentes na suite de *benchmarks* Rodinia [Che et al., 2009] e no CUDA sdk [SDK, 2011] a análises estáticas e dinâmicas além de realizar otimizações manuais e automáticas em seus códigos e verificar os ganhos de desempenho.

1.2 Contribuições

Deste trabalho resultam as seguintes contribuições:

Definição de uma semântica de execução SIMD: O Capítulo 3 descreve uma formalização do modelo de execução SIMD por meio de uma linguagem simples chamada μ-SIMD. Embora já existam outras formalizações, como Bougé [Bougé & Levaire, 1992] e Farrell [Farrell & Kieronska, 1996], elas descrevem o programa em uma linguagem de alto nível, enquanto nossas análises e otimizações tratam programas em linguagem de montagem. Também foi implementado um interpretador de μ-SIMD em Prolog para verificar a formalização.

Mapa de divergências: O Capítulo 4 descreve uma técnica para monitoração dinâmica de divergências chamada mapa de divergências. Essa técnica monitora a localização e o volume de divergências. Para cada desvio no código, são usados dois contadores: o primeiro deles marca o número de *warps* que visitaram o desvio (tomando o desvio ou não) e o outro conta quantos *warps* que, tendo visitado aquele desvio, apresentaram uma divergência. Isso mostra ao desenvolvedor da aplicação os trechos do código onde ocorre degradação de desempenho causada por divergências e pode ser usado para prover informações ao compilador para utilização em otimizações guiadas por monitoração de execução.

Análise de Divergências: O Capítulo 5 descreve uma técnica de análise estática de divergências. Essa técnica analisa as dependências das variáveis do programa e verifica se um desvio é divergente através das variáveis que são utilizadas no cálculo da condição do desvio. Ela assume que os argumentos da chamada do *kernel* e a memória da GPU estão livres de divergências e a variável *threadId* (um registrador especial, que retorna o número da *thread* que o lê), normalmente usada para dividir os dados entre as *threads*, é divergente (uma variável divergente possui valores diferentes entre as *threads* do *warp*). A partir daí *threadId* torna divergente as variáveis que dependem dele e todos os desvios cujo cálculo da condição use variáveis divergentes são considerados divergentes. Embora essa análise gere falsos positivos, ela não apresenta falsos negativos, garantindo que os desvios que utilizam apenas variáveis que não dependem de *threadId* não são divergentes.

Unificação de caminhos divergentes: O Capítulo 6 descreve uma otimização que usa a análise de divergências descrita anteriormente para fundir caminhos divergentes de uma estrutura de controle *if/else* reaproveitando instruções presentes em ambos os caminhos. Essa otimização usa o algoritmo de sequenciamento Smith-Waterman [Smith & Waterman, 1981], com algumas modificações, para identificar sequências de instruções comuns aos caminhos divergentes.

As implementações do *profiler*, da análise de divergências e da unificação de caminhos divergentes trabalham com programas na linguagem de montagem PTX [PTX, 2010] para GPUs Nvidia, gerados a partir de código CUDA [CUDA, 2011]. Mas elas podem ser portadas para qualquer arquitetura SIMD que suporte execução parcial, ou seja, onde exista um mecanismo para especificar um subconjunto das unidades de processamento em SIMD que vai executar uma instrução.

Além disso, esse trabalho resultou nas seguintes publicações: um artigo na conferência International Symposium on Computer Architecture and High Performance Computing (SBAC) [Coutinho et al., 2010], que ganhou prêmio de melhor artigo; uma submissão no periódico Concurrency and Computation: Practice and Experience (CCPE) [Coutinho et al., 2012] e um artigo na conferência Parallel Architectures and Compilation Techniques (PACT) [Coutinho et al., 2011], cuja taxa de aceitação foi de 16%.

Além dos trabalhos citados acima, este trabalho contém outras duas partes. O Capítulo 2 descreve o contexto em que situa-se este trabalho, explicando os princípios que norteiam o funcionamento de GPUs, bem como a história da evolução destas placas

e os trabalhos relacionados. Finalmente, o Capítulo 7 conclui este texto, acrescentando ao mesmo nossas considerações finais.

Capítulo 2

Background

2.1 A roda da reencarnação do *hardware*

A roda da reencarnação é um termo cunhado por Myer e Sutherland [Myer & Sutherland, 1968]. É interessante notar que esse conceito é tão antigo, que precede o Unix e que a evolução de seu sistema de processamento gráfico parece muito com a história das GPUs atuais. Esse termo se refere ao seguinte processo, que acontece de forma muito comum no projeto de *hardware*:

- Cria-se um dispositivo acelerador para realizar alguma tarefa muito comum ou dispendiosa e liberar a CPU para outras atividades.
- O dispositivo fica cada vez mais poderoso, eventualmente tornando-se Turing completo[1].
- Percebe-se que o processador do dispositivo passa a maior parte do tempo executando uma tarefa repetitiva. Adiciona-se *hardware* para liberar o processador do dispositivo para outras tarefas, essencialmente criando um acelerador do acelerador e reiniciando o ciclo.

Exemplos de situações em que este fenômeno ocorre incluem:

Computação gráfica: Foi projetando um processador gráfico que Myer e Sutherland perceberam a "roda da reencarnação". Na computação gráfica já se deu a "volta" na roda muitas vezes, com inúmeros dispositivos. A história das GPUs e sua relação com a roda serão explicados em detalhes na Seção 2.2.

[1] Em termos gerais, um processador *Turing completo* pode executar todos programas que uma máquina de Turing é capaz de executar.

Processamento matemático: A arquitetura x86 começou sem suporte a operações de ponto flutuante. Para sanar essa deficiência, em 1987 foi lançado um coprocessador para realizar essa tarefa, o infame 8087. Já a o acelerador matemático ClearSpeed CSX700 [ClearSpeed, 2011] é Turing completo, além de ter suporte a inteiros.

Estrada e saída: Os primeiros discos para PCs endereçavam os setores do disco pelo método CHS (Cylinder-Head-Sector), onde havia uma correspondência direta entre o endereço de um setor e sua localização física no disco. Posteriormente, o endereçamento passou a ser feito pode meio de endereços lógicos e foi incluído um pequeno processador para fazer a tradução entre endereços lógicos e físicos. Eventualmente foi estudada a possibilidade de realizar parte do processamento de uma aplicação no controlador do disco para diminuir a quantidade de dados transferida entre o disco e o computador [Acharya et al., 1998].

Comunicação: Já foram implementadas várias interfaces de redes com hardware para executar parte do processamento de pacotes no lugar do processador. Atualmente as placas disponíveis no mercado fazem no mínimo o cálculo da soma de verificação do pacote. Um tipo de aceleração mais complexa é o *TCP Offload Engine* (TOE), que consiste em executar o protocolo TCP (e protocolos abaixo dele) diretamente na interface de rede [Mogul, 2003]. Normalmente um TOE é implementado usando um processador de propósito geral que executa um sistema operacional embarcado. A interface de rede da interconexão para *clusters* Quadrics é implementada utilizando processadores RISC, com instruções extras para facilitar o processamento de pacotes [Petrini et al., 2002].

Gerenciamento: O objetivo aqui não e ganhar velocidade, mas ser capaz de controlar uma máquina remotamente, mesmo que seu sistema operacional esteja travado. Para ter essa capacidade, vários servidores possuem "controladores de gerenciamento", sendo que vários desses controladores são praticamente computadores separados. Por exemplo, o módulo de gerência remota Intel RMM3 [rmm3, 2010] possui um processador ARM9, 64 MB de memória, interface de rede própria, executa uma versão de Linux para sistemas embarcados, e ainda possui um acelerador criptográfico. Assim, um controlador de gerenciamento corresponde à "uma volta e meia" na roda da reencarnação.

Impressão: Impressoras de rede possuem processadores próprios para evitar a necessidade de ter uma máquina dedicada como servidor de impressão. Elas chegam

a ter seu próprio sistema operacional e, segundo analistas de segurança, podem ser utilizadas para invadir uma rede de computadores.

2.2 História das GPUs

Os computadores de propósito geral que compramos nas lojas normalmente vêm equipados com uma placa de processamento gráfico. É este hardware super especializado que representa na tela do computador as janelas, botões e menus dos aplicativos mais comuns e também as belas imagens 3D dos jogos de computador. Algumas destas placas são programáveis, isto é, nós, usuários, podemos criar programas que serão executados neste hardware. Isto, é claro, faz perfeito sentido, afinal, pode-se imaginar que uma placa de processamento gráfico tenha um enorme poder computacional, afinal, as imagens que são exibidas nos monitores de nossos computadores são cada vez mais exuberantes e complexas.

Mas nem sempre foi assim. Até meados da década de 90, não existia quase nenhum hardware gráfico em computadores pessoais IBM PC. Em 1984, mais ou menos na mesma época em que a SGI lançou seu primeiro terminal gráfico, a IBM lançou o IBM Professional Graphics Controller para PCs, mas ele foi um fracasso devido a seu alto custo e falta de software compatível. Em 1987, a IBM definiu o padrão VGA (*video graphics array*). As placas de vídeo VGA eram praticamente isto: uma área contígua de dados (um vetor, também chamado de *frame buffer*) que podem ser lidos e representados em uma tela. A CPU, é claro, era a responsável por preencher esta memória. E tal preenchimento deveria acontecer várias vezes a cada segundo. Inicialmente, esta carga sobre a CPU não era um terrível problema. Os aplicativos gráficos de então eram muito simples, normalmente bidimensionais. À medida que a complexidade das cenas 2D foi aumentando, foram implementadas acelerações 2D, como operações para combinar vários *bitmaps* em uma imagem e operações de desenho de figuras geométricas, como retângulos, triângulos, círculos e arcos.

Esta época saudosa, contudo, não durou muito. Com o surgimento de bibliotecas 3D, como OpenGL [Shreiner et al., 2011], surgiram novas aplicações. Jogos digitais e aplicações de CAD passaram a exigir processamento de imagens tridimensionais; usuários tornavam-se sempre mais exigentes. Esta pressão contínua sobre os aplicativos gráficos fez com que engenheiros de hardware movessem algumas das atividades de processamento para o que seriam as primeiras placas gráficas.

Algumas atividades de processamento gráfico são muito intensivas. Rasterização, isto é, a conversão de imagens em formato vetorial (ou seja, descritas por meio de

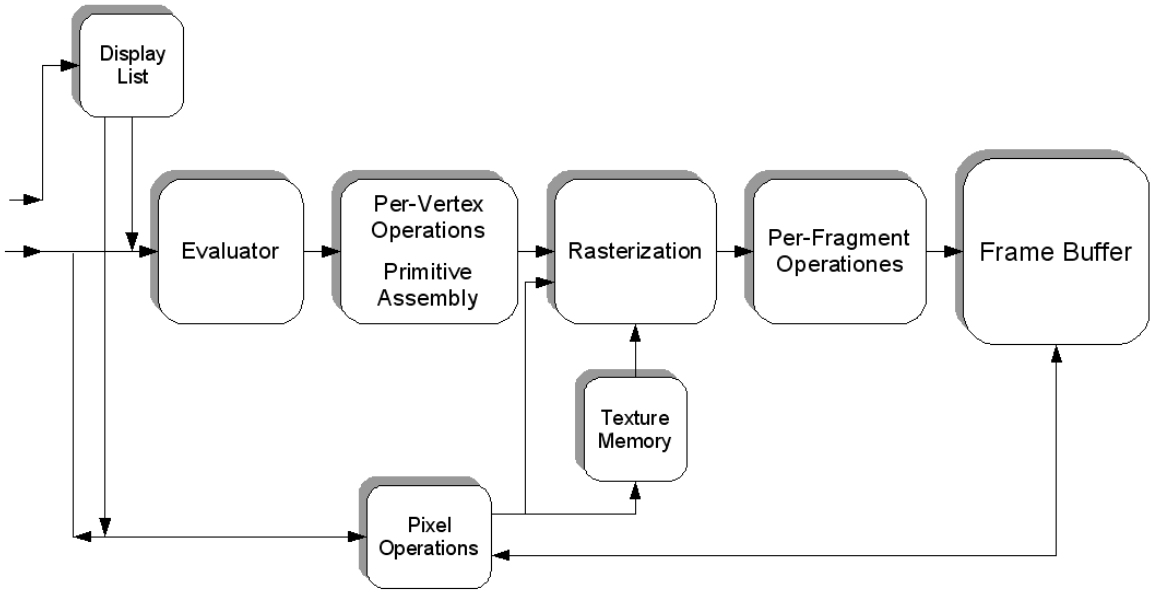

Figura 2.1: Versão simplificada do *pipeline* inicial do OpenGL [Shreiner et al., 2011]. Todos os estágios eram de função fixa. Para permitir a criação de efeitos especiais foram adicionados estágios programáveis, chamados de *shaders*.

primitivas geométricas como pontos, linhas e triângulos) em uma matriz de pixels (para exibição em um monitor, por exemplo), é uma delas. Esta contínua pressão sobre os aplicativos gráficos fez os engenheiros de hardware moverem atividades de processamento gráfico, como os últimos estágios do *pipeline* OpenGL (Figura 2.1), para o que seriam as primeiras placas aceleradoras 3D comercialmente bem sucedidas para computadores IBM PC.

A Nvidia criou o primeiro chip com uma unidade de transformação (conversão das coordenadas em três dimensões do objeto no mundo virtual em coordenadas de duas dimensões da janela) e iluminação integrada. Antes essa funcionalidade estava restrita a dispositivos muito caros, para desenho assistido por computador, e era implementada em um chip separado. Para enfatizar essa diferença, a Nvidia cunhou o termo *Graphic Processing Unit* (GPU), posteriormente generalizado para incluir qualquer acelerador 3D.

Shaders, ou processadores de sombras, vieram logo a seguir. O sombreamento contribui para que os usuários tenham a impressão de profundidade ao visualizarem imagens tridimensionais em uma tela bidimensional. *Shaders*, contudo, possuem uma diferença teórica muito importante, quando comparados com rasterizadores: há muitos, muitos algoritmos diferentes de sombreamento, ao passo que o processo de rasterização

não possui tantas variações. Não seria viável, em termos de custo financeiro, implementar todos estes algoritmos de processamento em hardware. A solução encontrada pelos arquitetos e engenheiros da época foi criar sombreadores programáveis. Foram criadas linguagens para a programação destes sombreadores, como a OpenGL Shading Language, também conhecida como GLSL e Cg [Mark et al., 2003]. Embora fossem programáveis, inicialmente esses sombreadores não eram Turing completos e seus programas não podiam ter laços [Microsoft, 2011a, Microsoft, 2011b]. Existem 3 tipos de *shaders*:

Vertex shaders: é executado uma vez para cada vértice da imagem. Seu propósito é transformar a posição do vértice em 3D no espaço virtual para as coordenadas 2D nas quais ele vai aparecer na tela. Eles podem manipular propriedades dos vértices existentes, mas não podem criar novos vértices.

Geometry shaders: são um tipo relativamente novo de shader. Recebe a saída do *vertex shader* e pode alterá-la, inclusive por meio da criação de novos vértices.

Pixel shaders: calculam a cor e outros atributes de um pixel.

Com surgimento dos *shaders*, foram feitos vários trabalhos para utilizar o poder computacional das GPUs para computação científica [Thompson et al., 2002, Harris et al., 2002, Bolz et al., 2003, Krüger & Westermann, 2003, Fan et al., 2004]. Mas programar uma aplicação científica usando linguagens e APIs gráficas era muito trabalhoso, além de exigir uma curva de aprendizado muito alta. Foram criadas linguagens de propósito geral para GPUs como Brook [Buck et al., 2004], que amenizavam esses problemas, mas o poder de expressão dessas linguagens era limitado pela falta de flexibilidade do *hardware*, o que impediu sua disseminação.

O salto de *shaders* configuráveis para processadores de propósito geral não foi exatamente simples, contudo foi o resultado de uma busca legítima e bastante natural pelo melhor aproveitamento dos recursos computacionais disponíveis. Foram incorporadas cada vez mais funcionalidades aos *shaders* até que se tornaram Turing completos. Eventualmente alguém teve a ideia de utilizá-la para processamento de propósito geral e foi criado o termo GPGPU (*General-Purpose Computation on Graphics Processing Unit*).

Diferentes aplicações podem necessitar quantidades diferentes de recursos em cada estágio do *pipeline gráfico*. Jogos normalmente necessitam mais de texturas, aplicações CAD de polígonos e um mesmo jogo pode ter demandas diferentes dependendo da cena. Assim, teve-se a ideia de criar uma GPU "unificada", com unidades de processamento

"genéricas", que pudessem executar qualquer tipo de *shader* e vários estágios do *pipeline gráfico*, de forma que o número de unidades responsáveis por cada estágio seja escolhido dinamicamente, conforme a necessidade. Além disso essa GPU teria de ser capaz de executar código de propósito geral e ser programável utilizando linguagens de propósito geral, como a linguagem C. Então a Nvidia desenvolveu a Geforce 8800 GTX, que embora projetada principalmente para processar imagens, possuía vários componentes típicos de processadores de propósito geral, como instruções de ponto flutuante no padrão IEEE 754 [Nickolls & Dally, 2010]. Para facilitar o aproveitamento do poder computacional dessa arquitetura em aplicações de propósito geral, foi desenvolvida a linguagem CUDA [CUDA, 2011], sigla para *Compute Unified Device Architecture*.

As tarefas de processamento gráfico foram migrando, pouco a pouco, da CPU em direção à placa gráfica, que acabou tornando-se um hardware de propósito geral. O curioso desta história é que, por motivo de eficiência, foram mantidos módulos especializados e não programáveis nas GPUs. Foram mantidas, por exemplo, funções de rasterização implementadas em hardware, como era o caso das antigas GPUs. Aparentemente a *roda da reencarnação* deu "uma volta e meia" na história das placas gráficas.

2.3 Arquitetura das GPUs

Nesta seção descreveremos o modelo de programação das GPUs da Nvidia, CUDA. No modelo de programação CUDA, a GPU, chamada de dispositivo, atua como um coprocessador, acelerando computações de uma determinada máquina, chamada de hospedeira. A GPU possui um grande número de processadores, que têm a habilidade de acessar diretamente a memória global do dispositivo. Isso permite um modelo de programação mais flexível que o de gerações de GPUs anteriores. Tal modelo facilita o desempenho de *kernels* paralelos.

GPUs proveem melhor desempenho para tarefas regulares que têm muita computação por acesso à memória. Ganhos de desempenho alcançam, neste caso, 100x, quando comparado com o uso de somente um core de uma CPU. Porém, mesmo em aplicações com pouca computação por acesso à memória, GPUs conseguem acelerações em torno 10x devido à grande banda de memória (de 70 a 192 GB/s, dependendo do modelo) e à capacidade de ocultar a latência de memória utilizando um grande número de *threads*.

A seguir, na Seção 2.3.2 apresentaremos a microarquitetura das GPUs e as entidades de hardware. Já na Seção 2.3.1, discutiremos as entidades de software e como elas se relacionam com as entidades de hardware. Finalmente, na Seção 2.3.3 descreveremos

brevemente como uma aplicação CUDA é compilada.

2.3.1 Entidades de *Software*

As *threads* executando na GPU são organizadas em uma hierarquia de três níveis. No nível mais alto, todas as *threads* executando a chamada de *kernel* formam um *grid*. Cada *grid* consiste de vários blocos de *threads*, que por sua vez podem ter várias *threads*.

2.3.1.1 *Grid*

Um *grid* é o nível mais alto da hierarquia de paralelismo da GPU. Ele é constituído por todas as *threads* executando uma chamada de *kernel*. Cada *grid* consiste de muitos blocos de *threads*, podendo ter até 2×10^{16} blocos distribuídos em duas dimensões. Cada bloco, neste caso, tem coordenadas únicas. O número de blocos no *grid* e sua distribuição são definidos explicitamente pela aplicação como parâmetros da chamada do *kernel*.

2.3.1.2 Bloco de *Threads*

Um bloco de *threads* é uma matriz de 3 dimensões, podendo ter até 512 *threads*. Assim como o grid, o número de *threads* e sua distribuição são definidos pela aplicação como parâmetros da chamada do *kernel* e cada *thread* tem coordenadas únicas dentro do *kernel*. Os blocos de *threads* são distribuídos entre os SMs e uma vez atribuídos a um SM não podem migrar. Até 8 blocos podem residir em um SM.

Threads de um mesmo bloco podem compartilhar dados através da memória compartilhada (uma memória de baixa latência da GPU). A sincronização de *threads* dá-se via uma primitiva de barreira. *Threads* de blocos diferentes são completamente independentes, assim sua sincronização só pode ser efetuada com segurança através do término do *kernel*.

2.3.1.3 *Warp*

O *warp* é um grupo de 32 *threads* que executam em modo SIMD. Durante a execução as *threads* de um bloco são agrupadas em *warps* de 32 *threads*, que são a menor unidade de escalonamento da GPU. Os *warps* são formados por faixas contínuas de *threads* dentro de um bloco: as primeiras 32 *threads* formam o primeiro *warp*, as 32 seguintes o segundo e assim por diante. Se o número de *threads* em um bloco não for divisível pelo tamanho do *warp*, as unidades de processamento que executariam as posições remanescentes ficarão ociosas.

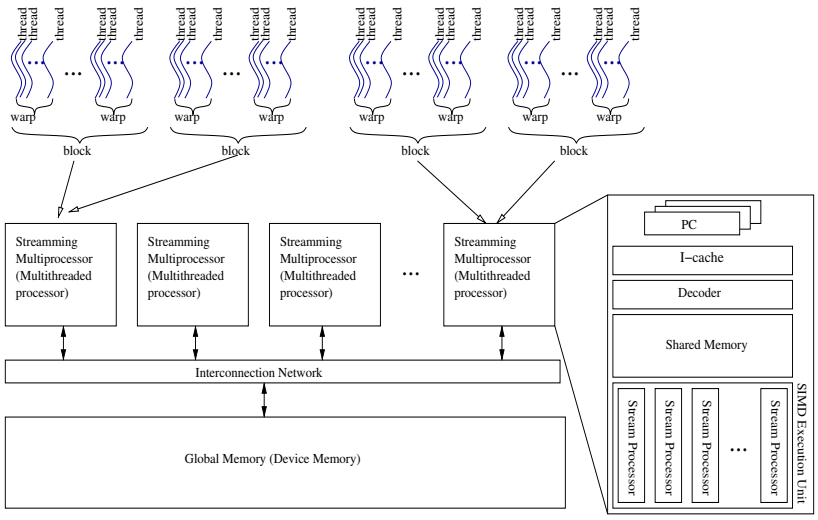

Figura 2.2: Arquitetura Nvidia Fermi [Nickolls & Dally, 2010].

Apesar de não serem explicitamente declarados no código CUDA, o conhecimento dos *warps* permite a criação de várias otimizações, como as descritas no Capítulo 6.

2.3.2 Entidades de *Hardware*

Nesta seção apresentaremos a arquitetura da GPU Nvidia Fermi, (a mais recente em 2011) representada na Figura 2.2. Ela consiste de 16 *streaming multiprocessors* (SMs), contendo 32 cores ou *streaming processors* (SPs) cada. Os cores executam a mesma instrução, cada um sobre dados de uma *thread* diferente, o que a Nvidia chamou de modelo SIMT: *Single Instruction, Multiple Thread*. Isso permite que unidades de instruções sejam compartilhadas entre os cores, economizando área do chip para ser utilizada em mais cores. Isso permite que as GPUs tenham maior área do chip voltada para o processamento de instruções que as CPUs, que têm a maior parte da área do chip dedicada a cache e controle, como mostra a Figura 2.3.

Além disso, a GPU utiliza um grande número de *threads* para esconder a latência de operações matemáticas e acessos à memória, permitindo o uso de caches menores e ALUs mais simples. Por exemplo: a latência de um acesso à memória em uma GPU como a Nvidia Geforce GTX 280 (arqutetura Tesla, anterior à Fermi) é de cerca de 440 ciclos, e a de uma operação matemática é de 24 ciclos [Papadopoulou et al., 2008], ela pode utilizar até 32 *warps* (1024 *threads*) para esconder estas latências. Estas mudanças de paradigma geram a grande vantagem, em termos de desempenho, das GPUs em relação às CPUs.

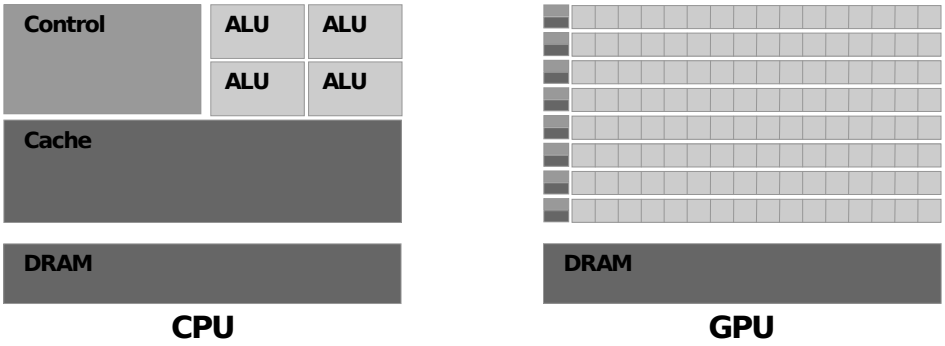

Figura 2.3: Diagrama mostrando tamanho de varias partes da CPU e GPU. A GPU dedica mais transistores ao unidades lógico-aritméticas (ALUs me inglês), enquanto as CPUs utilizam maior parte dos transistores com grandes unidades de controle (para serem capazes de executar instruções fora de ordem) e cache [CUDA, 2011].

Cada core tem uma unidade de ponto flutuante de 32 bits[2], capaz de realizar operações *multiply-add* além das operações mais comuns de ponto flutuante e operações com inteiros de 32 bits. Além disso, cada multiprocessador tem duas unidades especiais (SFUs) para execução de operações mais complexas, como seno, cosseno, raiz quadrada, inverso $(1/x)$, exponencial e logaritmo.

Considerando que todas ALUs e SFUs são totalmente otimizadas, sendo, no melhor caso, capazes de iniciar uma instrução a cada ciclo, o desempenho teórico da GPU GeForce GTX 580 (a GPU para jogos mais potente com um chip da arquitetura Fermi em 2011) pode ser calculado pela seguinte fórmula:

$$Op/s = OPI \times IPC \times F \tag{2.1}$$

Onde OPI é o número de operações de ponto flutuante por ciclo IPC é o número de instruções por ciclo que a GPU executa, F a frequência de operação da GPU. O melhor caso dessa GPU é usando a instrução *multiply-add*, que executa duas operações de ponto flutuante por instrução, resultando em um OPI de 2 operações por instrução e pode ser disparada uma vez por cada SP a cada ciclo para precisão simples. A GeForce GTX 580 possui 16 *streaming multiprocessors*, cada um com 32 *streaming processors*, que pode executar um *multiply-add* a cada ciclo, assim temos um IPC de $16 \times 32 \times 1 = 512$ instruções por ciclo. Como os SPs dessa GPU têm uma frequência de 1.544 GHz, a Equação 2.1 para precisão simples pode ser expandida como:

[2]Placas baseadas no chip GT200 ou superior são capazes de realizar cálculos de ponto flutuante de 64 bits.

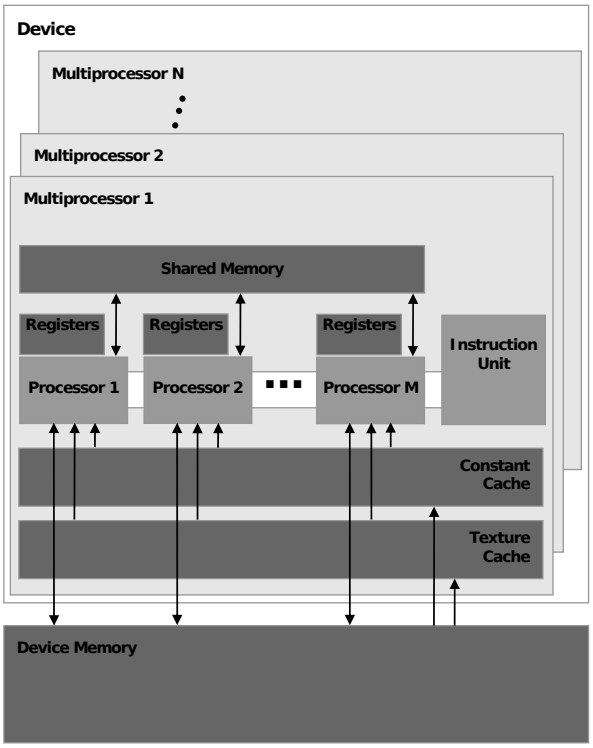

Figura 2.4: Espaços de memória das GPUs da Nvidia [CUDA, 2011].

$$\begin{aligned} Op/s &= OPI \times IPC \times F \\ &= 2 \times 512 \times 1.544 \\ &= 1581.056 GFLOPS/s \end{aligned} \tag{2.2}$$

2.3.2.1 *Streaming Multiprocessor* (SM)

O *Streaming Multiprocessor* (SM) é uma unidade unificada para execução de programas gráficas e de computação, capaz de executar *shaders* de vértices, geometria e *pixels*, além de programas computação paralelos [Lindholm et al., 2008]. Ela contêm 32 cores ou *streaming processors* (SPs), que se dividem em dois grupos de 16, onde cada grupo executa um *warp* (grupos de *threads* que executam de maneira síncrona) de 32 *threads* a cada 2 ciclos (a frequência da unidade de instruções é metade da frequência dos SPs). Isso permite que cada multiprocessador possua apenas duas unidades de instruções (uma para os *warps* pares e outra para os ímpares), cada uma comandando um grupo de 16 cores.) multiprocessador pode abrigar até 1536 *threads*, em 48 *warps*, controladas através de um *scoreboard*, podendo trocar de *warp* com custo zero, para esconder

latências de instruções e acessos à memória [Brodtkorb et al., 2010]. O multiprocessador é capaz de disparar somente uma instrução de cada vez para todas as *threads* de um *warp*. Caso *threads* de um mesmo *warp* sigam caminhos diferentes durante a execução, o multiprocessador terá que executar o *warp* em várias passadas, executando a cada vez as *threads* que seguiram um mesmo caminho. Geralmente é desejável agrupar as *threads* para evitar essa situação.

Para que essas *threads* tenham dados com que possam trabalhar, o multiprocessador possui um banco de registradores de 128 KB, composto por 32768 registradores de 32 bits, que podem são divididos entre todas as *threads* e uma memória de 64 KB, que pode ser dividida entre cache L1 e memória compartilhada, um espaço de memória mais rápida que pode ser utilizada para armazenar dados que serão utilizados com muita frequência. Os outros espaços de memória estão representados na Figura 2.4. No código fonte, as variáveis têm atributos extras para indicar se residem na memória global, local, compartilhada ou de constantes. A memória de texturas é acessada via chamadas de função que são transformadas em instruções especiais. A largura de banda da memória é muito alta, entre 70 e 159 GB/s, mas ela pode saturar se muitas *threads* acessam-na em um curto período de tempo. Além disso, se o acessos à memória feitos pelas *threads* de um *warp* não forem vetores contínuos e alinhados de 16 palavras (o tamanho da palavra depende do tipo de dados que as *threads* estão acessando) pode-se não conseguir utilizar toda a banda de memória disponível. Para obter um bom desempenho também são necessárias otimizações para reutilização de dados. Uma destas otimizações consiste em unir vários acessos em vetores de 16 palavras.

Existem vários fatores que podem limitar o número de *threads* executando simultaneamente na GPU. Primeiro, o número máximo de *threads* que podem ser executadas em um multiprocessador é 1536. Segundo, cada bloco de *threads* utiliza recursos como registradores e memória compartilhada, fazendo com que o esgotamento de algum desses recursos no SM limite ainda mais o número de blocos de *threads*, (consequentemente o número de *threads*) que podem residir no SM. Para permitir que aplicações que utilizam todas as *threads* tenham um número razoável de registradores por *thread* e que as aplicações que não utilizam todas as *threads* disponíveis possam aproveitar os registradores que ficariam ociosos, cada SM tem 32K registradores que são particionados dinamicamente entre as *threads* que estão executando nele.

Estas limitações têm duas consequências. Primeiro, otimizações podem ter impacto negativo em alguns casos porque, devido ao grande número de *threads*, pequenas mudanças (como a utilização de um registrador a mais por *thread*) podem ter efeitos multiplicativos na utilização de recursos, diminuindo o número de blocos que podem residir no SM. Segundo, máximos locais são comuns durante a otimização de código.

Mesmo após ser alcançado um código com desempenho satisfatório ainda deve-se testar configurações com grandes variações [Ryoo et al., 2008b].

2.3.2.2 *Streaming Processor* (SP)

Os 32 *streaming processors* de um SM executam instruções em modo SIMT. Eles trabalham no dobro da frequência do resto da GPU, aumentando dobrando seu desempenho sem necessariamente ter que dobrar a frequência de todas as partes, o que causaria um aumento muito grande no consumo de energia da GPU. Para as instruções mais simples, como adição, multiplicação e *multiply-add* de ponto flutuante de precisão simples, além de somas e operações lógicas em inteiros de 32 bits, o SP consegue disparar uma instrução por ciclo. Além disso, arquitetura Fermi os 2 grupos de 16 SMs que compõem um SP podem se juntar para realizar uma operação de ponto flutuante de precisão dupla, o que permite que o desempenho em precisão dupla da GPU seja metade da precisão simples [Nickolls & Dally, 2010].

2.3.3 Compilação do código CUDA

ANSI C, estendido com certas construções e palavras-chave, é o modelo de programação CUDA. A GPU é tratada como um coprocessador que executa o código do *kernel* em paralelo. O programador gera um fonte único, contendo código tanto para o *host* (CPU) como para o dispositivo (GPU).

Esses códigos são separados e compilados como mostrado na Figura 2.5. Cada programa CUDA consiste de múltiplos trechos de código que são executados tanto na CPU, quanto na GPU. Trechos que têm pouco ou nenhum paralelismo são executadas na CPU. Eles são expressos em ANSI C e compiladas pelo compilador C como mostrado na Figura 2.5. Trechos que têm mais paralelismo são implementadas como *kernels* que executam na GPU. Cada um desses *kernels* define código que será executado pelas *threads* invocadas. Os *kernels* são compilados pelo compilador de C CUDA da Nvidia (NVCC) e pelo gerador de código para GPU.

Existem algumas restrições nos *kernels*: não pode haver funções com número de argumentos variáveis e para GPUs anteriores à arquitetura Fermi também não pode ter recursão ou ponteiros para funções. Transferências de dados entre a memória da máquina e memória global da GPU são efetuadas por meio de funções fornecidas pela API CUDA. A execução de um *kernel* é iniciada por meio de uma chamada específica por parte da CPU.

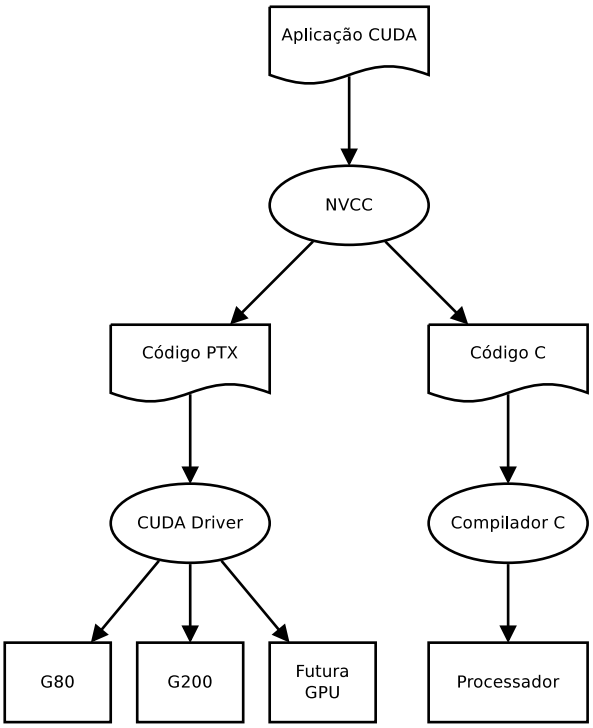

Figura 2.5: Compilação de código CUDA.

2.4 Trabalhos Relacionados

Nesta seção nós descrevemos os principais trabalhos relacionados à pesquisa que estamos propondo. Nós agrupamos estes trabalhos em dois grupos. O primeiro grupo, descrito na Seção 2.4.1, aborda a pesquisa feita com relação à mensuração dinâmica de desempenho de aplicações paralelas. Dentre estes trabalhos, citamos as técnicas de *profiling* dinâmico, inicialmente desenvolvidas para aplicações sequenciais, e posteriormente utilizadas no mundo paralelo. O segundo grupo de trabalhos relacionados, descrito na Seção 2.4.2, contém pesquisas feitas em análise estática de código. Dentre estes trabalhos, citamos os projetos de *profiling* estático, desenvolvidos inicialmente para a predição do resultado de desvios condicionais, e que hoje pretendemos utilizar para entender o comportamento de aplicações paralelas.

2.4.1 Análise Dinâmicas de Desempenho em CPUs Paralelas

A literatura contém uma vasta gama de trabalhos relacionados à mensuração de desempenho de aplicações em geral [Graham et al., 1982a, Nethercote & Seward, 2007, Srivastava & Eustace, 1994, Magnusson et al., 2002] e de aplicações paralelas em particular [Miller et al., 1995, Ji et al., 1998, Xu et al., 1999, Eyerman & Eeckhout, 2009,

Tallent & Mellor-Crummey, 2009]. Pode-se dizer que a análise de desempenho de aplicações sequenciais teve início com o surgimento de técnicas de *profiling* dinâmico. Um exemplo dentre as primeiras ferramentas que utilizavam tais técnicas é `gprof` [Graham et al., 1982a]. Por outro lado, técnicas de análise de desempenho em aplicações paralelas são muito posteriores. As primeiras pesquisas nesta área começaram em meados dos anos 80, com ferramentas como PIE [Segall & Rudolph, 1985] para instrumentação baseada em *tracing* de aplicações executando em redes de *workstations*. Posteriormente apareceram as primeiras ferramentas para *profiling* em computadores multiprocessados [Davis & Hennessy, 1988, Sites & Agarwal, 1988] (inicialmente chamados de multicomputadores) no final da década de 80. Desde o lançamento dos primeiros computadores paralelos, muitas outras técnicas e ferramentas de mensuração foram criadas [Miller et al., 1995, Ji et al., 1998, Xu et al., 1999, Eyerman & Eeckhout, 2009, Tallent & Mellor-Crummey, 2009]. Tais ferramentas apresentam diferentes compromissos em termos de precisão, automação, nível de intrusão e, no caso de simuladores, tempo de simulação (que pode chegar a 1000 vezes o tempo de execução na máquina real).

2.4.1.1 Primórdios: análises em aplicações sequenciais

`Gprof` [Graham et al., 1982a] pode ser considerada a primeira ferramenta de *profiling* dinâmico de grande popularidade. Este aplicativo, surgido durante os primeiros anos da década de 80, estendeu a ferramenta `prof` [prof, 1979] do Unix, considerada o estado da arte na época. Ao contrário de `prof`, `gprof` é capaz de criar o grafo de chamadas das funções que compõem uma aplicação. Por exemplo, se um SGBD gastou maior parte do tempo de execução executando a chamada de sistema `write()`, saberemos que o tempo de execução é dominado por entrada e saída. Por outro lado, também é importante saber como este tempo de execução está distribuído. Desta forma, se no SGBD 70% do tempo gasto em chamadas `write()` é feito por chamadas `appendToJournal()` e 30% é feito por chamadas `commitChanges()`, então saberemos que o tempo de execução é dominado pela gravação dos dados no disco.

`Gprof`, o trabalho pioneiro de Graham *et al.* permitiu o surgimento de diversas novas técnicas de otimização de código. No início da década de 90, Chang *et al.* [Chang et al., 1991] mostraram como informações obtidas em tempo de execução, via *profiling* dinâmico podem ser usadas para melhorar a qualidade do código gerado durante a compilação. O trabalho de Chang *et al.* é um exemplo canônico de um otimizador que se beneficia de informação de *profiling*. Outro exemplo de otimizações de código guiadas por *profiling* pode ser encontrado no trabalho seminal de Wall *et*

al. [Wall, 1992]. Neste caso, o *profiler* é utilizado para decidir quais variáveis devem ser mantidas em registradores, em vez de serem mapeadas para a memória.

Uma forma de melhorar técnicas de *profiling* dinâmico é através da redução dos níveis de intrusão no código sob análise, ou seja, a redução da quantidade de código necessário para a instrumentação. Uma abordagem que tenta diminuir a intrusão e ainda assim ter alta precisão é a simulação do processador. Dois trabalhos muito notáveis nessa linha são Valgrind [Nethercote & Seward, 2007] e Simics [Magnusson et al., 2002]. O Valgrind consegue instrumentar binários sem modificá-los. Já o Simics é tão detalhado que pode simular com precisão grandes *pipelines* com execução fora de ordem, como aquele presente no Pentium 4. É possível, por exemplo, executar sistemas operacionais completos usando a arquitetura emulada pelo Simics. O problema desse tipo de ferramenta é o tempo de simulação. Valgrind pode ser até 100 vezes mais lento que uma execução normal dependendo da análise feita [Nethercote & Seward, 2007], e Simics pode ser até 1000 vezes mais lento dependendo do nível de detalhe da simulação [Magnusson et al., 2002].

Hoje ferramentas de *profiling* são utilizadas em uma ampla variedade de domínios. Por exemplo, Shark [Apple, 2009], uma ferramenta utilizada pela *Apple*, possibilita inspecionar tanto o código fonte de uma aplicação quanto o equivalente código *assembly*. Outra ferramenta largamente utilizada é Dtrace [Cantrill et al., 2004], um instrumentador de programas completos, aplicável desde núcleos de sistemas operacionais até aplicações inteiras. Uma das grandes contribuições de Dtrace é permitir a inserção de instrumentação durante a execução de programas. Atualmente existem *profilers* que medem, não somente o tempo de execução ou o uso de memória, mas também o consumo de energia. Um exemplo desta última linha de aplicações é PowerTop [PowerTop, 2009]. Além disto, ferramentas clássicas como `gprof` e valgrind nunca deixaram de ser utilizadas e são atualmente distribuídas juntamente com as distribuições mais famosas do Linux [Dalheimer & Welsh, 2005].

Porém, em 2005 percebeu-se que as oportunidades de melhoria de desempenho de um único núcleo de processamento (*core*) estão praticamente esgotadas [Asanovic et al., 2006]. A partir dessa data, aplicações que não utilizarem múltiplos núcleos só obterão melhorias de desempenho mínimas, a um ritmo muito mais lento que anteriormente. Isso causou uma expansão sem precedentes de hardware e técnicas de programação paralelos. Algumas das técnicas de medição descritas nesta seção podem ser utilizadas neste paradigma [Magnusson et al., 2002, Cantrill et al., 2004, PowerTop, 2009], porém acreditamos que novas estratégias são necessárias.

2.4.1.2 Desenvolvimentos: análise de aplicações paralelas

A mensuração de desempenho de aplicações paralelas possui uma diferença fundamental com relação à mesma atividade restrita a aplicações sequenciais: a iteração entre linhas de execução diferentes durante a execução da aplicação. Assim, as primeiras ferramentas de *profiling* usadas em aplicações paralelas em geral produziam *traces* ou registros de execução por *thread* ou processo. Tais relatórios podiam então ser analisados após a execução do programa sob análise. Uma das primeiras estratégias de *profiling* de aplicações paralelas de que se tem notícia foi IPS [Miller & Yang, 1987], uma ferramenta usada na análise de aplicações distribuídas baseadas em troca de mensagens. Uma das grandes inovações de IPS foi a *análise de caminho crítico*, que identificava os pontos de programa e os eventos que causavam degradação de desempenho. A partir de IPS foi desenvolvida a ferramenta IPS-2 [Miller et al., 1990], que adicionava ao seu predecessor uma interface gráfica, instrumentação mais eficiente, reduzindo o nível de intrusão e novas técnicas de análise: suavização de curvas (suprime picos resultantes da granularidade da medição), segmentação (identifica mudança de fase em uma métrica) e combinação de dados (correlaciona várias métricas para identificar mudança de fase na aplicação).

Tanto IPS quanto IPS-2 eram ferramentas voltadas à análise de aplicações baseadas em troca de mensagem, existem, contudo, diversas ferramentas e técnicas específicas para sistemas baseados em memória compartilhada. Por exemplo, Helen Davis e John Hennessy fizeram uma das primeiras ferramentas para análise do comportamento de sincronização de programas [Davis & Hennessy, 1988]. Os métodos de Davis estão voltados para aplicações executando em memória compartilhada, analisando, por exemplo, o tempo de espera em *locks*. Outra ferramenta voltada para a análise de aplicações paralelas em memória compartilha é a extensão para multi-processadores da ferramenta ATUM desenvolvida por Sites e Agarwal [Sites & Agarwal, 1988]. O sistema ATUM original [Agarwal et al., 1986] foi largamente utilizado em meados dos anos oitenta para mensurar o acesso a caches de dados. Sua extensão paralela [Sites & Agarwal, 1988] foi desenvolvida com propósito similar, porém, nesta caso, a existência de múltiplos processadores era levada em consideração para inferir o uso da hierarquia de memória.

A evolução de sistemas de mensuração de desempenho de aplicações paralelas continuou durante os anos 90. Crovella criou a análise de predicados [Crovella & LeBlanc, 1993], para identificar em que tipo de atividade o tempo de uma aplicação paralela foi gasto. Uma das grandes inovações de Crovella foi dividir o tempo de execução da aplicação em diversas categorias: comunicação, processamento, tempo ocioso, etc. Uma outra ferramenta da época foi Pa-

radyn [Hollingsworth & Miller, 1993, Miller et al., 1995], que pode inserir instrumentação dinamicamente, restringindo a intrusão a partes específicas do código sob análise. Isso permite que ele faça inicialmente uma instrumentação de baixa intrusão para procurar problemas de alto nível e depois insira instrumentação mais detalhada apenas nos módulos que apresentarem problemas para descobrir a causa do prolema detectado. Tal fato permitiu a Paradyn poder analisar aplicações que executavam durante muito tempo, e que, de outro modo, produziriam registros de execução muito grandes. Paradyn foi posteriormente modificada por Xu *et al.* [Xu et al., 1999] para instrumentar processos que possuem várias *threads*. Para fazer isso, foram criadas *thread conscious locks* para evitar que uma *thread* entre em deadlock com ela mesma caso ocorra uma troca de contexto enquanto estiver executando uma seção crítica da instrumentação e a rotina que instrumenta essa troca de contexto tente trancar a mesma estrutura utilizada na seção crítica. Também foi necessário criar temporizadores virtuais para evitar fazer chamadas de sistema de alto custo a cada troca de contexto entre *threads*. No fim dos anos 90, surgiram diversas ferramentas que, não somente medem tempo de espera de *threads*, mas também analisam as dependências entre elas [Ji et al., 1998].

Atualmente, sistemas de medição de desempenho em aplicações paralelas são ainda área ativa de pesquisa. Aguillera *et al.* [Aguilera et al., 2003] desenvolveram técnicas que analisam sistemas distribuídos (como aplicações de 3 camadas que rodam em *clusters* de servidores) que possuem módulos de código fechado via análise da troca de mensagens realizada durante a execução, chegando a descobrir erros de programação nos sistemas analizados.

Um dos grandes objetivos da pesquisa atual é permitir que programadores tenham um melhor entendimento do código paralelo, por exemplo, quais fatores estão causando a degradação de desempenho. Huck e Malony [Huck & Malony, 2005] adicionaram ao conjunto de ferramentas TAU [Shende & Malony, 2006] uma ferramenta que utiliza mineração de dados para correlacionar várias métricas e sumarizar grandes quantidades de dados obtidos na instrumentação de uma aplicação paralela. Já Carnival [Meira, 1997] além de utilizar mineração de dados, tem uma análise de causa e efeito para geração automática de explicações para os custos oriundos da paralelização. Tallent *et al.* [Tallent & Mellor-Crummey, 2009] criaram técnicas para analisar aplicações escritas em linguagens que fazem *job stealing*. Nessas linguagens, as *threads* podem roubar funções da pilha das outras e, assim, a pilha de execução das *threads* não reflete o que está no código fonte. A ferramenta de Tallent *et al.* [Tallent & Mellor-Crummey, 2009] foi implementada como um gerador de dados para o popular sistema *HPCView* [Mellor-Crummey et al., 2002]. Em outra linha de pesquisa, Eyerman *et al.* [Eyerman & Eeckhout, 2009] criaram uma técnica que permite

um processador SMT reportar os ciclos gastos em cada *thread* e usar essa informação no escalonamento delas.

Recentemente temos observado o surgimento de novas técnicas de construção de *hardware* paralelo: as GPUs. Estas plataformas diferem dos sistemas descritos nesta seção conforme descrito na Seção 2.3. A análise de desempenho em GPUs é ainda um desafio, pois este *hardware* apresenta um número muito grande de *threads* executando simultaneamente, e os sistemas descritos nesta seção foram utilizados na medição de sistemas multi-processados com um pequeno número de unidades de processamento. São tais faltas que o trabalho apresentado nesta dissertação visa suprir.

2.4.1.3 O presente: análise de GPGPUs

GPGPUs constituem uma das mais recentes inovações da indústria de *hardware* paralelo. Com o surgimento dos shaders e da linguagem CG [Mark et al., 2003], apareceram vários trabalhos que portavam aplicações de propósito geral para a GPU e realizavam avaliações de desempenho simples (como tempo de execução por tamanho da entrada) e eventualmente comparavam o tempo de execução com uma implementação que usava somente a CPU [Harris et al., 2002, Thompson et al., 2002, Bolz et al., 2003, Krüger & Westermann, 2003, Fan et al., 2004]. Mas como os fabricantes de GPUs divulgavam poucos detalhes da arquitetura desses dispositivos, não era possível fazer análises mais profundas.

Ryoo *et al.* [Ryoo et al., 2008a] fizeram uma das primeiras análises de desempenho de aplicações de propósito geral em GPU. Nesse trabalho foi estudada a arquitetura da GPU Nvidia GeForce 8800 GTX, suas funcionalidades e estratégias de otimização adotadas pelos desenvolvedores para atingir alto desempenho. Os princípios citados pelos autores são: (1) ocultar a latência de acessos à memória utilizando muitas *threads* simultaneamente; (2) otimizar a utilização da memória compartilhada, pois ela pode limitar o número de *threads* executando simultaneamente; (3) agrupar as *threads* de um *warp* para evitar os custos de sair da execução SIMD e (4) evitar conflitos entre *threads* durante acessos à memória. Este trabalho descreve, então, uma série de princípios para melhorar o desempenho de aplicações baseadas em GPUs; porém, ele não fornece detalhes quantitativos sobre o impacto de cada um destes princípios na aplicação. É nossa intenção fornecer tais detalhes, via *profiling* dinâmico e análise estática do código fonte das aplicações paralelas.

O NVIDIA Compute Visual Profiler [NVP, 2010] (CVP) é uma ferramenta criada pela NVIDIA para ajudar o desenvolvimento de aplicações CUDA. Esta ferramenta analisa aplicações paralelas via contadores como: acessos à memória glocal agrupados

e não agrupados; leituras e escritas à memória local; e desvios divergentes. Embora uma ferramenta largamente utilizada, CVP possui três deficiências que esta tese busca sanar:

- CVP utiliza contadores de eventos tais como acessos à memória e desvios executados que acontecem durante a execução do *kernel*; porém, CVP não mostra o impacto desses eventos no desempenho da aplicação. Por exemplo, nós queremos informar ao programador que a aplicação gastou 20% de seu tempo de execução em acessos à memória.

- A análise é feita apenas em um multiprocessador, o que dificulta o estudo de aplicações irregulares, onde *warps* executando em diferentes multiprocessadores podem não ter o mesmo comportamento. Por isto, CVP não consegue detectar desbalanceamento de carga entre processadores.

- CVP mostra o perfil do *kernel* como um todo, não sendo capaz de analisar individualmente cada *warp*. Isto implica que CVP não consegue indicar pontos onde ocorre desbalanceamento de carga dentro de um mesmo processador.

Em [Boyer et al., 2008] foi desenvolvido um sistema automatizado para análise de aplicações CUDA, o qual é voltado para duas classes de problemas comuns encontrados: condição de corrida, e conflitos de bancos de memória compartilhada. A abordagem utilizada para analisar a aplicação é instrumentação automática do código CUDA, que é executado em modo de emulação (executando na CPU). Embora este *profiler* seja capaz de identificar condições de corrida, e a ocorrência de conflitos de acesso à memória, ele apresenta problemas semelhantes à CVP. Assim, ele não pode medir o impacto dos conflitos de memória no desempenho da aplicação, mesmo porque ele não simula a hierarquia de memória da GPU, usando a memória da CPU para identificar conflitos.

Em [Hong & Kim, 2009] foi proposto o primeiro modelo analítico de desempenho para GPU. Ao contrário do modelo Roofline [Williams et al., 2009], que abstrai quase todos os detalhes da arquitetura, esse modelo leva em conta muitos detalhes, chegando a diferenciar se acessos à memória são agrupados (*coalesced*) ou não. Ele só não leva em conta as regras de agrupamento de memória mais relaxadas da versão 1.3 da arquitetura, pois seria necessário analisar sequências de acesso à memória. Este modelo foi favorecido pela arquitetura das GPUs da Nvidia:

- Cada unidade de processamento pode executar[3] uma instrução por ciclo[4].
- Executa instruções em ordem.
- As GPUs da NVIDIA existentes até a publicação do artigo só utilizam caches nos espaços de memória de textura e constantes, tornando os tempos de acesso constantes para a maioria dos espaços de memória.
- A única primitiva de sincronização é a barreira. Essa primitiva é simples de caracterizar.

Dentre os pontos deste modelo que nós gostaríamos de melhorar, podemos citar:

- Este modelo não considera variações durante a execução do programa, só a média geral.
- O modelo assume que os escalonadores de execução e requisições de memória são justos com todos os *warps*, ou seja, não tendem a atender algum *warp* primeiro.
- O modelo não leva em conta o impacto de caches no desempenho do acesso à memória.
- O modelo não leva em conta conflitos causados pela largura da banda de acesso à memória. Isto é, caso várias *threads* leiam a mesma área de memória, a banda de envio de dados disponível pode não ser grande o suficiente para enviar dados simultaneamente a todas as *threads*.
- O modelo assume que instruções de maior custo como seno e raiz quadrada têm o mesmo custo que instruções mais simples.
- O modelo não leva em conta dependências de dados. Isto é, ele não leva em conta a latência para os resultados tornarem-se disponíveis.
- O modelo assume que o multiprocessador pode executar novos *warps* sempre que houver recursos suficientes, porém a granularidade da GPU é medida em blocos.

Em [Bakhoda et al., 2009] foi implementado um simulador da microarquitetura de uma GPU para avaliação de desempenho de aplicações em CUDA. O simulador executa o conjunto de instruções PTX [PTX, 2010]. Este simulador se propõe a ser uma plataforma que possibilite a instrumentação e análise de aplicações CUDA. São justamente estes tipos de análises que esta tese procura definir e desenvolver.

[3] O termo mais preciso seria emitir, pois cada instrução vai demorar pelo menos 24 ciclos para chagar ao final do pipeline [Papadopoulou et al., 2008].

[4] Somente para as instruções mais comuns [Papadopoulou et al., 2008].

2.4.2 Análise estática

A análise estática consiste de um conjunto de técnicas que extraem informação do código fonte de uma aplicação [Aho et al., 2008, Appel & Palsberg, 2003, Torczon & Cooper, 2007], sem que seja necessário executar a mesma. Técnicas de análise estática já foram utilizadas em uma vasta gama de situações, dentre as quais citam-se os três exemplos abaixo, extraídos de Hall *et al.* [Hall et al., 2009]:

- Detecção de falhas de segurança em programas [Foster, 2002, Evans & Larochelle, 2002].
- Detecção de trechos de código que podem ser otimizados [Pereira & Palsberg, 2008].
- Detecção de condições de corrida em aplicações paralelas [Cherem et al., 2008].

Um dos objetivos desta tese é utilizar técnicas de análise estática para detectar situações que comprometem a performance de aplicações escritas em CUDA. A próxima seção mostra alguns dos usos de análise estática no desenvolvimento de aplicações paralelas.

2.4.2.1 *Profiling* estático

Profiling estático é uma alternativa ao *profiling* dinâmico. Duas vantagens da alternativa estática com relação à sua contra-parte dinâmica são menores níveis de intrusão no código sob análise, e independência de entradas específicas. Uma das técnicas de *profiling* estático é a predição de desvios [Ball & Larus, 1993]. Esta análise é utilizada para evitar ciclos ociosos no *pipeline* de processadores que fazem previsão de desvios. Caso um desvio seja previsto corretamente não haverá ciclos ociosos; caso contrário, o processador deve anular todas instruções disparadas desde a instrução de desvio. Quanto maior o *pipeline* do processador maior é o numero de estágios (e de instruções disparadas) entre os dois eventos. Wu *et al.* [Wu & Larus, 1994] apresentaram um conjunto de heurísticas para refinar os resultados obtidos por Ball *et al.* [Ball & Larus, 1993]. Enquanto a predição de desvios de Ball *et al.* associam valores binários as arestas do grafo de fluxo de controle do programa analisado, indicando se aquele caminho será tomado ou não, a análise de Wu *et al.* é mais precisa. Ela associa probabilidades de execução a cada aresta do CFG do programa. A partir destas arestas é possível determinar um número aproximado de vezes em que cada parte do programa será executada. Embora trabalhos essenciais à literatura de compiladores, as análises de Wu *et al.* e Ball *et al.* foram desenvolvidas para programas sequenciais. Nosso objetivo é empregar análises similares em programas paralelos.

2.4.2.2 Análise estática de programas paralelos

Análise estática de programas paralelos ainda é uma área pouco estudada na literatura; porém, diversos tipos de análises estáticas foram já utilizadas neste domínio [Asanovic et al., 2006]. Por exemplo, Cherem *et al.* utilizaram um sistema de tipos para determinar que determinados programas não possuem condições de corrida [Cherem et al., 2008]. Análise estática foi empregada com sucesso em arquiteturas SIMD (*Single Instruction, Multiple Data*). Um exemplo notável é a técnica conhecida como *software pipelining* [Lam, 1988]. Esta análise permite que instruções no interior de laços sejam arranjadas de modo a poderem ser executadas em paralelo. *Software pipelining* utiliza uma estrutura de dados que será útil nas análises que nós propomos desenvolver nesta tese: o grafo de dependência de dados.

Atualmente, uma das análises mais populares utilizadas em programas paralelos é MHP (*may happen in parallel*) [Agarwal et al., 2007]. O objetivo desta análise é indicar se é possível que dois comandos sejam executados em paralelo. De acordo com Agarwal *et al.*, MHP foi utilizada com sucesso em várias ferramentas que detectam condições de corrida [Choi et al., 2002, Krinke, 1998, Masticola & Ryder, 1991, Naumovich et al., 1999]. Por outro lado, não conhecemos qualquer trabalho que tenha utilizado análise estática para mensurar o desempenho de aplicações paralelas baseadas em CUDA/GPU. É um dos objetivos desta tese sanar tal deficiência, desenvolvendo uma análise estática para verificar se um desvio será divergente, descrita no Capítulo 5. Essa análise poderá ser utilizada para por uma técnica de otimização automática de código divergente, como a do Capítulo 6, para determinar quais trechos de código devem ser otimizados.

Capítulo 3

Semântica de execução SIMD

Neste capítulo descreveremos uma máquina SIMD abstrata, a fim de formalizar as várias análises e optimizações que introduziremos em capítulos posteriores. Demarcaremos assim, de forma não ambígua, o modelo de execução ao qual nossas técnicas se aplicam. Adotamos o mesmo modelo de execução SIMD independentemente descrito, primeiro por Bougé [Bougé & Levaire, 1992], e posteriormente por Farrell [Farrell & Kieronska, 1996]. Neste modelo, temos uma série de *elementos de processamento* (PEs) executando instruções em sincronia, sujeitas à *execução parcial*: Às vezes, um programa pode necessitar que apenas um subconjunto dos PEs execute uma instrução. Nas palavras de Farrel *et al.* "Todos os PEs executam a mesma instrução ao mesmo tempo com o estado interno de cada PE estando ativo ou inativo." [Farrell & Kieronska, 1996, p.40].

O arquétipo de uma máquina SIMD é o computador ILLIAC IV [Barnes et al., 1968, Bouknight et al., 1972], precursor de uma longa linha de sistemas SIMD. Essa caracterização da execução SIMD também é adequada para muitas linguagens antigas de paralelismo de dados [Abel et al., 1969, Bouknight et al., 1972, Brockmann & Wanka, 1997, Hoogvorst et al., 1991, Sun-Yuan Kung et al., 1982, Lawrie et al., 1975, Perrott, 1979]. Além disso, desenvolvimentos recentes em placas gráficas trouxeram novos membros a esta família. O modelo de execução *Single Instruction Multiple Threads* (SIMT) [Patterson & Hennessy, 2008], um termo cunhado pela Nvidia, essencialmente descreve uma máquina com vários conjuntos de processadores, onde os processadores de cada conjunto operam em SIMD e CUDA é uma linguagem de programação que coordena muitos processadores SIMD. Nesta seção, formalizaremos o modelo de execução SIMD através de uma linguagem simples, que chamamos de μ-SIMD. Não reutilizamos a semântica formal da Bougé ou Farrell porque ambos assumem uma linguagem de alto nível, enquanto que nossas análises

(Rótulos – $Lbl \subset \mathbb{N}$)	::=	$\{l_1, l_2, \ldots, \}$
(Variáveis – *Var*)	::=	$\texttt{tid} \cup \{v_1, v_2, \ldots, \}$
(Instruções – *Inst*)	::=	
– (desvio condicional)	|	$\texttt{branch}(v, l)$
– (desvio incondicional)	|	$\texttt{jump}(l)$
– (escrita na memória compartilhada)	|	$\texttt{store}(v, v_x)$
– (leitura da memória compartilhada)	|	$\texttt{load}(v, v_x)$
– (incremento atômico)	|	$\texttt{atominc}(v, v_x)$
– (operação binária)	|	$\texttt{binop}(v_1, v_2, v_3)$
– (carga de imediato)	|	$\texttt{const}(v, n)$
– (barreira de sincronização)	|	`sync`
– (término da execução)	|	`stop`

Figura 3.1: Sintaxe das instruções μ-SIMD.

(Memória local)	$\sigma \subset Var \mapsto \mathbb{Z}$
(Vetor compartilhado)	$\Sigma \subset \mathbb{N} \mapsto \mathbb{Z}$
(PEs ativos)	$\Theta \subset (\mathbb{N} \times \sigma)$
(Programa)	$P \subset Lbl \mapsto Inst$
(Pilha de sincronização)	$\Pi \subset Lbl \times \Theta \times Lbl \times \Theta \times \Pi$

Figura 3.2: Elementos que constituem o estado de uma programa μ-SIMD.

e otimizações são melhor descritas usando linguagem de montagem. Observe que nosso modelo não lida com instruções vetoriais como as extensões SSE da Intel, que são popularmente chamados de SIMD, porque estes sistemas normalmente não têm suporte para a execução parcial. Nós escrevemos um interpretador de μ-SIMD em Prolog e alguns programas exemplo.

Sintaxe e Semântica de μ-Simd: A Figura 3.1 apresenta as instruções μ-SIMD. Usamos o *opcode* `binop` para descrever operações binárias (operações com dois operandos) mais comuns, como adição e multiplicação.

Definimos uma máquina abstrata para executar programas μ-SIMD. O estado M dessa máquina é determinado por uma tupla de cinco elementos: $(\Theta, \Sigma, \Pi, P, pc)$, que definimos na Figura 3.2. Cada elemento de processamento é um par (t, σ), onde t é um número natural que identifica o PE e é referido pela variável especial `tid`. O símbolo σ representa a memória local do PE. A memória local é uma função que mapeia nomes de variáveis para números inteiros. Ela é individual de cada PE, contudo as funções de todos os PEs têm o mesmo domínio de nomes de variáveis. Assim, uma variável $v \in \sigma$ na verdade, denota um vetor de variáveis, cada uma delas privada e pertencente

$$\text{(SP)}\quad \frac{P[pc] = \texttt{stop}}{(\Theta, \Sigma, \emptyset, P, pc) \to (\Theta, \Sigma)}$$

$$\text{(BT)}\quad \frac{P[pc] = \texttt{branch}(v, l) \qquad \mathbf{split}(\Theta, v) = (\emptyset, \Theta) \qquad \mathbf{push}(\Pi, \emptyset, pc, l) = \Pi' \qquad (\Theta, \Sigma, \Pi', P, l) \to (\Theta', \Sigma')}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$\text{(BF)}\quad \frac{P[pc] = \texttt{branch}(v, l) \qquad \mathbf{split}(\Theta, v) = (\Theta, \emptyset) \qquad \mathbf{push}(\Pi, \emptyset, pc, l) = \Pi' \qquad (\Theta, \Sigma, \Pi', P, pc + 1) \to (\Theta', \Sigma')}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$\text{(BD)}\quad \frac{\begin{array}{c} P[pc] = \texttt{branch}(v, l) \\ \mathbf{split}(\Theta, v) = (\Theta_0, \Theta_n) \qquad \mathbf{push}(\Pi, \Theta_n, pc, l) = \Pi' \\ (\Theta_0, \Sigma, \Pi', P, pc + 1) \to (\Theta', \Sigma') \end{array}}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')} \tag{3.1}$$

$$\text{(SS)}\quad \frac{P[pc] = \texttt{sync} \qquad \Theta_n \neq \emptyset \qquad (\Theta_n, \Sigma, (pc', \Theta_0, l, \emptyset) : \Pi, P, l) \to (\Theta', \Sigma')}{(\Theta, \Sigma, (pc', \emptyset, l, \Theta_n) : \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$\text{(SP)}\quad \frac{P[pc] = \texttt{sync} \qquad (\Theta_n, \Sigma, (\_, \emptyset, \_, \Theta_0) : \Pi, P, pc + 1) \to (\Theta', \Sigma')}{(\Theta_0 \cup \Theta_n, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$\text{(JP)}\quad \frac{P[pc] = \texttt{jump}(l) \qquad (\Theta, \Sigma, \Pi, P, l) \to (\Theta', \Sigma')}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$\text{(IT)}\quad \frac{P[pc] = \iota \qquad \iota \notin \{\texttt{stop}, \texttt{branch}, \texttt{sync}, \texttt{jump}\} \qquad (\Theta, \Sigma, \iota) \to (\Theta', \Sigma') \qquad (\Theta', \Sigma', \Pi, pc + 1, \Theta'', \Sigma'')}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta'', \Sigma'')}$$

Figura 3.3: Semântica de μ-SIMD: operações de fluxo de controle. Por concisão, quando duas hipóteses forem verdadeiras, usa-se a de cima.

a um dos PEs. Um PE não tem acesso à memória local de outro PE, mas eles podem se comunicar através do vetor compartilhado Σ. Usamos Θ para designar o conjunto de PEs ativos.

Um programa P é um mapa de rótulos para instruções. O estado do programa contém um *program counter* (pc), que é o rótulo da próxima instrução a ser executada. Finalmente, a nossa máquina contém uma *pilha de sincronização* Π. Cada nó de Π é uma tupla $(l_{id}, \Theta_{done}, l_{next}, \Theta_{todo})$ que denota um *ponto de sincronização*, ou seja, um ponto onde PEs divergentes devem sincronizar. Esses nós são empilhados quando os

$$\textbf{split}(\Theta, v) = (\Theta_0, \Theta_n)$$
$$\textbf{where}$$
$$\Theta_0 = \{(t,\sigma) \mid (t,\sigma) \in \Theta \ \textbf{and} \ \sigma[v] = 0\}$$
$$\Theta_n = \{(t,\sigma) \mid (t,\sigma) \in \Theta \ \textbf{and} \ \sigma[v] \neq 0\}$$

$$\textbf{push}([], \Theta_n, pc, l) = [(pc, [], l, \Theta_n)]$$

$$\textbf{push}((pc', [], l', \Theta'_n) : \Pi, \Theta_n, pc, l) = \Pi' \ \textbf{if} \ pc \neq pc'$$
$$\textbf{where}$$
$$\Pi' = (pc, [], l, \Theta_n) : (pc', [], l', \Theta'_n) : \Pi$$

$$\textbf{push}((pc, [], l, \Theta'_n) : \Pi, \Theta_n, pc, l) = (pc, [], l, \Theta_n \cup \Theta'_n) : \Pi$$

Figura 3.4: Funções auxiliares usadas na definição de μ-SIMD.

PEs divergem em um desvio, tomando caminhos diferentes. O rótulo l_{id} representa o desvio condicional que causou a divergência, Θ_{done} são os PEs que atingiram o ponto de sincronização, enquanto que Θ_{todo} são os PEs aguardando para executar. O rótulo l_{next} indica a instrução onde Θ_{todo} vão continuar sua execução.

O resultado da execução de uma máquina abstrata μ-SIMD é um par (Θ, Σ). A Figura 3.3 descreve a semântica das instruções que alteram o fluxo de controle do programa, por exemplo: desvios condicionais e incondicionais, barreiras de sincronização e pontos de parada. Dizemos que um programa termina se $P[pc] = \texttt{stop}$. A semântica dos desvios condicionais é mais elaborada. Ao chegar a uma instrução de desvio $\texttt{branch}(v, l)$, v é avaliado na memória local de cada PE ativo. Se $\sigma(v) = 0$ para todos PEs, então a regra BF move o fluxo para a próxima instrução, ou seja, $pc + 1$. Da mesma forma, se $\sigma(v) \neq 0$ para todos PEs, então na regra BT desvia-se para a instrução em $P[l]$. Finalmente, se tivermos valores distintos entre os PEs, então dizemos que o desvio é *divergente*. Neste caso, como descrito na regra BD, executa-se os PEs que vão executar o trecho "then" e mantém-se os outros PEs na pilha de sincronização, para executá-los mais tarde. Observe que mesmo as regras para desvios não-divergentes podem atualizar o topo da pilha de sincronização. A atualização da pilha é efetuada pela função **push** na Figura 3.4. Fazemos desta forma para evitar que máquina a abstrata fique travada tentando desempilhar um nó quando passar por uma barreira. Segundo a regra SS, se chegarmos na barreira com um grupo Θ_n de PEs aguardando para executar, então retomaremos sua execução no trecho "else", e colocaremos o conjunto previamente ativo em espera. Finalmente, se chegarmos à barreira sem PEs esperando para executar, na regra SP sincroniza-se os PEs do conjunto "done" com os PEs ativos e a execução continua na instrução seguinte à barreira.

$$\frac{\Sigma(n_x) = n}{\Sigma \vdash n_x = n} \qquad t, \sigma \vdash \texttt{tid} = t \qquad \frac{v \neq \texttt{tid} \quad \sigma(v) = n}{t, \sigma \vdash v = n} \tag{3.2}$$

(3.3)

$$\text{(Tl)} \quad \frac{(t, \sigma, \Sigma, \iota) \to (\sigma', \Sigma') \qquad (\Theta, \Sigma', \iota) \to (\Theta', \Sigma")}{(\{(t, \sigma)\} \cup \Theta, \Sigma, \iota) \to (\{(t, \sigma')\} \cup \Theta', \Sigma")}$$

$$\text{(Ct)} \quad (t, \sigma, \Sigma, \texttt{const}(v, n)) \to (\sigma \setminus [v \mapsto n], \Sigma)$$

$$\text{(Ld)} \quad \frac{t, \sigma \vdash v_x = n_x \qquad \Sigma \vdash n_x = n}{(t, \sigma, \Sigma, \texttt{load}(v, v_x)) \to (\sigma \setminus [v \mapsto n], \Sigma)}$$

$$\text{(St)} \quad \frac{t, \sigma \vdash v_x = n_x \qquad t, \sigma \vdash v = n}{(t, \sigma, \Sigma, \texttt{store}(v, v_x)) \to (\sigma, \Sigma \setminus [n_x \mapsto n])} \tag{3.4}$$

$$\text{(At)} \quad \frac{t, \sigma \vdash v_x = n_x \qquad \Sigma \vdash n_x = n \qquad n' = n + 1}{(t, \sigma, \Sigma, \texttt{atominc}(v, v_x)) \to (\sigma \setminus [v \mapsto n'], \Sigma \setminus [n_x \mapsto n'])}$$

$$\text{(Bp)} \quad \frac{t, \sigma \vdash v_2 = n_2 \qquad t, \sigma \vdash v_3 = n_3 \qquad v_1 = n_2 \otimes n_3}{(t, \sigma, \Sigma, \texttt{binop}(v_1, v_2, v_3)) \to (\sigma \setminus [v_1 \mapsto n_1], \Sigma)}$$

Figura 3.5: Semântica de μ-Simd: operações de manipulação de dados e aritméticas.

A Figura 3.5 mostra a semântica do restante das instruções μ-Simd. Uma tupla $(t, \sigma, \Sigma, \iota)$ é usada para denotar a execução de uma instrução ι por um PE (t, σ). Segundo a regra Tl, todos PEs ativos executam a mesma instrução ao mesmo tempo. Modelamos esse fenômeno mostrando que a ordem na qual diferentes PEs processam ι é irrelevante. Assim, uma instrução como $\texttt{const}(v, n)$ faz com que cada PE atribua o valor n à sua variável local v. A instrução `store` pode resultar em condição de corrida, ou seja, dois PEs tentando escrever sobre o mesmo local no vetor compartilhado. Neste caso, o resultado é indefinido, devido à regra Tl. Garantimos atualizações atômicas através da instrução $\texttt{atominc}(v, v_x)$, que lê o valor de $\Sigma(\sigma(v_x))$, incrementa e o armazena de volta. Esse resultado também é copiado para $\sigma(v)$, como descrito na regra At. Na regra Bp usamos o símbolo $\otimes$ para avaliar operações binárias como adição e multiplicação.

3.1 Conclusão

Acreditamos que a nossa linguagem μ-Simd é a primeira descrição formal, em termos de sintaxe e semântica, de uma linguagem SIMD de baixo nível. Esta linguagem, embora

bastante simples e limitada, é poderosa o suficiente para descrever elementos típicos de uma arquitetura SIMD, tais como:

- comunicação entre diferentes PEs, o que se dá via memória compartilhada;
- execução em passo único, o que se dá devido à existência de um único contador de instruções compartilhado entre todos os PEs;
- execução divergente, que permite o tratamento de desvios condicionais. Alcançamos tal propriedade via uma pilha que armazena PEs dormentes, enquanto seus pares ativos executam;
- sincronização de PEs divergentes, o que se dá via uma instrução de sincronização.

Tendo definido μ-SIMD, utilizaremos esta linguagem nos exemplos e algoritmos que introduziremos nos próximos capítulos.

Capítulo 4

Análise de Divergências via Monitoração Dinâmica

Como detectar a localização e o volume das divergências que influenciam o desempenho de uma aplicação? Divergências ocorrem somente em desvios condicionais, porém, nem todo desvio condicional é divergente, e, dentre aqueles que o são, certamente divergências ocorrem em diferentes intensidades. Existem duas abordagens para a detecção de divergências: análise estática e *profiling*. A análise estática constitui-se de uma ferramenta que estuda o código do programa tentando provar propriedades, como quais desvios condicionais são passíveis de causar divergências e quais nunca o farão. Já o *profiling* é um método dinâmico: monitora-se um programa em execução, medindo-se coisas como quais desvios condicionais divergiram e quantas vezes o fizeram. Falaremos de *profiling* primeiro, deixando a análise estática para o Capítulo 5.

Profilers são aliados de longa data de desenvolvedores de aplicações de alto desempenho. Desde a introdução de `gprof`, uma ferramenta extremamente popular [Graham et al., 1982b], muitos outros *profilers* foram desenvolvidos, com vários propósitos, que vão desde guiar os compiladores otimizadores [Chang et al., 1991] até a depuração de erros [Nethercote & Seward, 2007]. Os primeiros *profilers*, naturalmente, eram destinados a medir o desempenho de aplicações sequenciais [Graham et al., 1982b, Magnusson et al., 2002, Nethercote & Seward, 2007, Srivastava & Eustace, 1994]. Trabalhos subsequentes, contundo, estenderam o escopo de tais ferramentas para analisar programas paralelos [Eyerman & Eeckhout, 2009, Ji et al., 1998, Miller et al., 1995, Tallent & Mellor-Crummey, 2009, Xu et al., 1999]. Ainda assim, *profilers* de aplicações paralelas, tais como os trabalhos desenvolvidos por Tallent *et al.* [Tallent & Mellor-Crummey, 2009] ou Eyerman e Eeckhout [Eyerman & Eeckhout, 2009], são utilizados em modelos de execução onde

threads trabalham de forma relativamente independente uma das outras. *Profilers* que atuem com sucesso sobre o ambiente SIMT das GPUs ainda são raridade.

Existem pelo menos dois *profilers* que capturam o comportamento divergente de programas CUDA. Um deles é o NVIDIA Compute Visual Profiler (CVP)[1]. Esta ferramenta permite ao desenvolvedor CUDA aferir vários números referentes à execução do programa paralelo, incluindo a quantidade de divergências que ocorreram durante esta execução. Contudo, CVP não aponta os pontos de programa em que as divergências aconteceram, em vez disto, informando um valor absoluto de divergências. Tal informação pode, contudo, ser obtida usando-se o *profiler* desenvolvido por Coutinho *et al.* [Coutinho et al., 2010], o qual descrevemos a seguir.

4.1 Divergências

Explicaremos o conceito de divergências mostrando como esse fenômeno ocorre em uma implementação do algoritmo de ordenação bitônica [Batcher, 1968][2]. O código, que obtemos de Cederman *et al.* [Cederman & Tsigas, 2009], é apresentado na Figura 4.1; onde focaremos na parte cinza. Rotulamos três desvios condicionais interessantes com os valores do mapa de divergências que obtemos através da estratégia de monitoração descrita na Seção 4.2. O *mapa de divergências* é uma função que mapeia desvios em pares de inteiros. O primeiro inteiro denota o número de divergências ao passo que o segundo representa o número de vezes que aquele desvio foi visitado.

A Figura 4.2 mostra uma representação simplificada do grafo de fluxo de controle do código PTX desse *kernel*. PTX [PTX, 2010] é uma linguagem de montagem de alto nível usada para representar código CUDA. Ela abstrai os registradores da máquina, instruções emuladas e interface de chamada de funções. Já o grafo de fluxo de controle é uma representação de código de máquina utilizada por compiladores para análise de código. Cada um dos blocos de instruções (L_1 a L_7) é chamado de *Bloco Básico*. Ele é uma sequencia de instruções, onde um desvio (se existir) pode ser apenas a última instrução da sequencia. O *grafo de fluxo de controle* é um grafo onde os nós são blocos básicos e as arestas representam caminhos entre blocos que podem ser percorridos durante a execução do programa. Incrementamos o grafo da Figura 4.2 com o endereço de cada instrução. Também adicionamos à figura os valores do mapa de divergências dos três desvios apresentados na Figura 4.1.

O algoritmo de ordenação bitônica não é muito usado em CPUs por ter complexidade $O(n \ln^2 n)$ quando executado sequencialmente; entretanto sua habilidade de

[1] `http://forums.nvidia.com/index.php?showtopic=57443`

[2] `http://www.cse.chalmers.se/research/group/dcs/gpuqsortdcs.html`

```
__global__ static void bitonicSort(int * values) {
  extern __shared__ int shared[];
  const unsigned int tid = threadIdx.x;
  shared[tid] = values[tid];
  __syncthreads();
  for (unsigned int k = 2; k <= NUM; k *= 2) {
    for (unsigned int j = k / 2; j>0; j /= 2) {
      unsigned int ixj = tid ^ j;
      if (ixj > tid) {
        if ((tid & k) == 0) {
          if (shared[tid] > shared[ixj]) {
            swap(shared[tid], shared[ixj]);
          }
        } else {
          if (shared[tid] < shared[ixj]) {
            swap(shared[tid], shared[ixj]);
          }
        }
      }
      __syncthreads();
    }
  }
  values[tid] = shared[tid];
}
```

7,329,816 / 28,574,321

15,403,445 / 20,490,780

4,651,153 / 8,083,541

Figura 4.1: A implementação da ordenação bitônica. Esse código está disponível no CUDA SDK.

executar $n/2$ comparações em paralelo, (onde n é o tamanho do arquivo) e executar sempre as mesmas comparações, independente da entrada torna esse algoritmo muito atrativo para uso em GPUs. Contudo, estamos restritos ao modelo de execução SIMD. Digamos que, por exemplo, nossa máquina SIMD possua quatro unidades de execução, mas apenas um decodificador de instruções. Invariavelmente a condição `(tid & k) == 0`, onde `tid` é o identificador da *thread*, será verdadeira em algumas delas e falsa em outras. Nesse caso, teremos uma divergência.

A Figura 4.3 apresenta um exemplo de execução em uma configuração com um único *warp* de quatro *threads*, e.g. $w = \{t_0, t_1, t_2, t_3\}$. Cada linha representa um ciclo, onde uma instrução do *kernel* será despachada. Representamos essa instrução como ι_i, indicando que a i-ésima instrução do *kernel* foi executada. Caso ocorram divergências, algumas *threads* não executarão nenhuma instrução. Indicamos esse fato utilizando o símbolo $\bullet$ para representar *threads* ociosas. Dada uma execução em que o vetor de

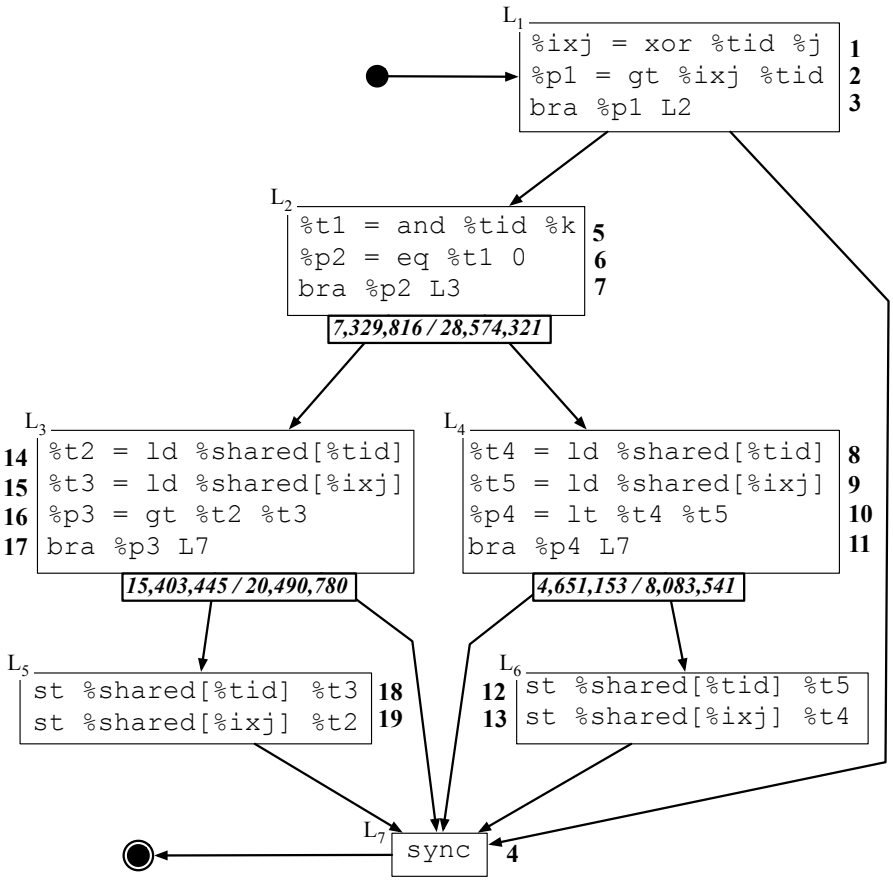

Figura 4.2: Representação em PTX (simplificada) do grafo de fluxo de controle de parte do *kernel* de ordenação bitônica.

entrada, `values`, é $\{4, 3, 2, 1\}$ e que o id da *thread* t_n é $n, 0 \leq n \leq 3$. Quando $k = 2$ e $j = 1$, então ocorrem duas divergências. A primeira separação acontece no ciclo $i = 3$, devido ao desvio condicional `bra %p1, L2`, que separa t_0 e t_2 de t_1 e t_3. Ela acontece porque a condição `%ixj > %tid` é verdadeira somente para t_0 e t_2. A segunda divergência acontece no ciclo $i = 6$, uma vez que o desvio `bra %p2, L3` e separa t_0 de t_2.

Após a primeira iteração do laço externo da Figura 4.1, quando $k = 2$, e $j = 1$, temos o seguinte mapa de divergências: $\iota_3 \mapsto (1, 1), \iota_7 \mapsto (1, 1)$. Após a segunda iteração, quando $k = 4$, e $j = 2$, ocorre uma nova divergência, resultando em $\iota_3 \mapsto (2, 2), \iota_7 \mapsto (1, 1)$. Esse mapa de divergências indica que a instrução ι_3 (`bra %p1, L2`) foi visitada por dois *warps* e em cada visita ocorreu uma divergência.

ciclo	instrução	t_0	t_1	t_2	t_3
1	`xor`	ι_1	ι_1	ι_1	ι_1
2	`gt`	ι_2	ι_2	ι_2	ι_2
3	`bra`	ι_3	ι_3	ι_3	ι_3
4	`and`	ι_5	•	ι_5	•
5	`eq`	ι_6	•	ι_6	•
6	`bra`	ι_7	•	ι_7	•
7	`load`	ι_{14}	•	•	•
8	`load`	ι_{15}	•	•	•
9	`gt`	ι_{16}	•	•	•
10	`bra`	ι_{17}	•	•	•
11	`load`	•	•	ι_8	•
12	`load`	•	•	ι_9	•
13	`lt`	•	•	ι_{10}	•
14	`bra`	•	•	ι_{11}	•
15	`store`	ι_{18}	•	•	•
16	`store`	ι_{19}	•	•	•
17	`store`	•	•	ι_{12}	•
18	`store`	•	•	ι_{13}	•
19	`sync`	ι_4	ι_4	ι_4	ι_4

Figura 4.3: Exemplo de execução do programa da Figura 4.2.

4.2 Monitoração de divergências

Implementamos o mapa de divergências como dois vetores de inteiros, τ e δ, de modo que $\tau[b]$ armazena o número de *warps* que visitaram o bloco básico b e $\delta[b]$ armazena o número de divergências que ocorreram no desvio no fim de b. Para facilitar a instrumentação de um programa, implementamos uma transformação de código que insere automaticamente a instrumentação. A instrumentação tem três fases:

Inicialização: reserva memória da GPU para armazenamento dos vetores e inicialização dos mesmos com zeros.

Mensuração: contagem do número de visitas e divergências, deixando os resultados armazenados nos vetores.

Leitura: cópia dos cados para a memória da CPU e acumulação com resultados e outras execuções de *kernel*.

Como as fases de inicialização e leitura são triviais, descreveremos apenas a fase de mensuração.

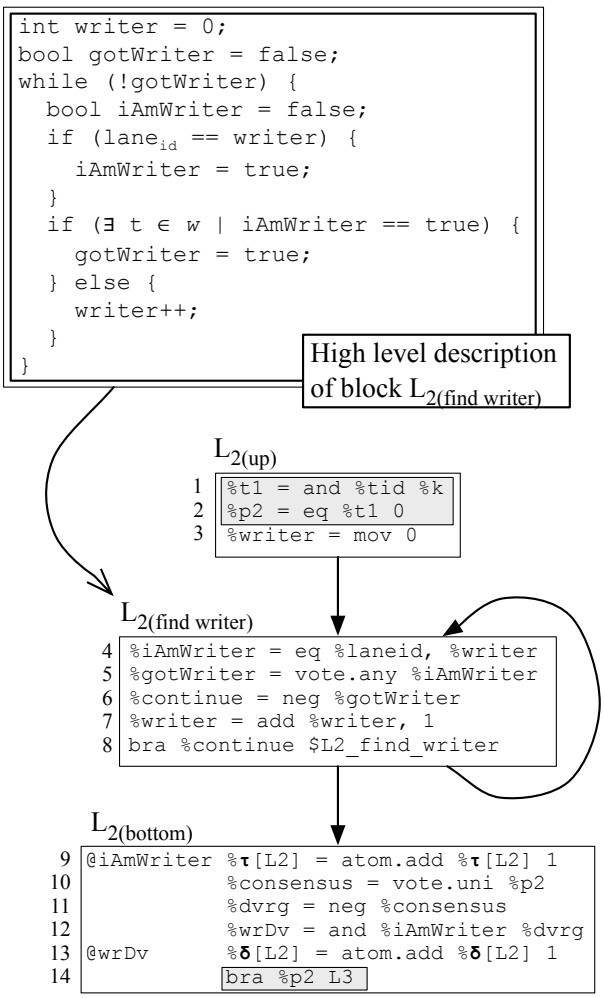

Figura 4.4: Código inserido para instrumentar bloco L_2 do código da Figura 4.2. Código original em cinza.

A transformação de código insere o código da Figura 4.4 em cada desvio condicional. Esta figura mostra a instrumentação do bloco básico L_2 do exemplo da Figura 4.2. Cada bloco básico b é dividido em três novos blocos: b_{up}, b_{bottom} e $b_{find\ writer}$. Note que nem todo desvio condicional é divergente e podemos usar a análise estática descrita no Capítulo 5 para evitar inserir instrumentação em desvios que a análise pode provar que nunca divergem. O código da instrumentação realiza as seguintes tarefas:

- Selecionar uma *thread* (a *thread* escritora) para atualizar os vetores δ e τ. Esse passo é realizado nos blocos b_{up} e $b_{find\ writer}$.

- Detecção e registro da divergência. Realizada no bloco b_{bottom}.

A *thread* escritora é a com o menor identificador entre as *threads* ativas do *warp*. Na Figura 4.4 a variável `%laneid` contém o número da *thread* no *warp*. Não podemos simplesmente escolher a *thread* onde `%laneid = 0`, pois ela pode estar inativa devido a uma divergência anterior. Assim, nos blocos b_{up} e $b_{find\ writer}$ é feita uma varredura à procura da *thread* ativa com menor `%laneid`.

Uma vez que foi escolhida uma *thread* escritora, incrementa-se o vetor τ para indicar uma visita e é realizada uma detecção de divergências via votação. A instrução PTX `%p = vote.uni.pred, %q` faz com que `%p` seja verdadeiro se e somente se todas as *threads* tenham o mesmo valor para o predicado `q`. Assim pode-se verificar se elas divergem usando-se o predicado, que é a condição do desvio potencialmente divergente. Se todas tiverem o predicado com o mesmo valor não ocorrerá uma divergência. Caso contrário a divergência deve ser registrada incrementando-se a posição relativa ao desvio no vetor δ. A instrução `@wrDv %`$\delta[L_2]$ `= atom.add %`$\delta[L_2]$ `1`, na 13ª linha do código da Figura 4.4, adiciona um à $\delta[L_2]$ se, e somente se a condição `@wrDv` for verdadeira. O que ocorrerá se a *thread* for a escritora e não houver consenso.

O resultado da instrumentação é uma tupla para cada bloco básico que contém um desvio condicional b, com os seguintes valores:

Id do bloco básico: Posição de b no código PTX do *kernel*.

Número de visitas durante a execução: valor de $\tau[b]$.

Número de divergências durante a execução: valor de $\delta[b]$.

A instrumentação requer a serialização dos *warps* nos desvios condicionais porque a atualização do mapa de divergências é realizada por meio de operações atômicas. Assim, no pior caso, o custo da instrumentação de um desvio será proporcional ao número de *warps* do *kernel* multiplicado pelo número de *threads* em um *warp*. Contudo, como divergências aninhadas são raras, a *thread* com o menor identificador no *warp* estará sempre presente entre as *threads* ativas. Assim, o laço no bloco $L_{2(find\ writer)}$, na Figura 4.4, tende a encontrar uma *thread* escritora válida já na primeira iteração. Será apresentada uma avaliação empírica do custo da instrumentação na Seção 4.4.

4.3 Otimizando o algoritmo de ordenação bitônica

Nesta seção, descreveremos como usamos as informações do mapa de divergências para melhorar o desempenho da execução do algoritmo de ordenação bitônica visto na seção anterior. A implementação do quicksort de Cederman *et al.* [Cederman & Tsigas, 2009]

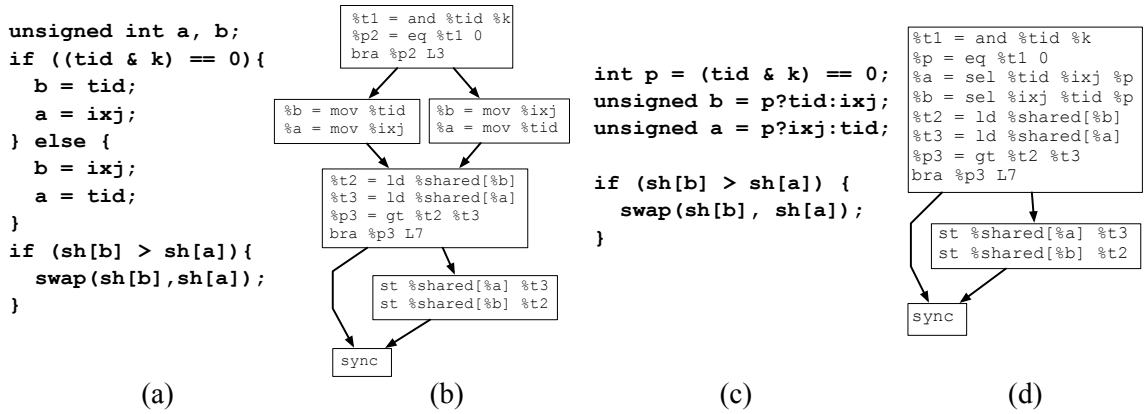

Figura 4.5: (a-b) Código fonte e PTX após a primeira otimização. (c-d) Código fonte e PTX após a segunda otimização.

usa o quicksort tradicional para particionar um grande vetor de números em pequenos blocos, que são então ordenados através do algoritmo apresentado na Figura 4.1. A aceleração que apresentamos nesta seção se refere ao tempo de execução do *kernel* quicksort como um todo. Contudo, salientamos que apenas o algoritmo de ordenação bitônica foi modificado. Assim, conseguimos 6-10% de aceleração modificando menos que 10-12 instruções em um código com 895 instruções. A Figura 4.1 mostra que os desvios condicionais dos laços aninhados sofrem muitas divergências. A condição `(tid & k) == 0`, em particular, é visitada por *warps* 28 milhões de vezes e um terço das visitas diverge. O mapa também mostra que divergências são comuns em ambos os desvios de ambos os caminhos, notadamente `shared[tid] > shared[ixj]` e `shared[tid] < shared[ixj]`.

Mostraremos como melhorar o desempenho deste algoritmo utilizando as informações fornecidas pela nossa instrumentação. Com tal intuito, comecemos por notar que a sequencia de instruções formada pelo bloco L_3 seguido pelo bloco L_5 é muito semelhante à sequencia $L_4 + L_6$. A única diferença vem dos cálculos de condições feitos nos pontos do programa 10 e 16. Armados com esta observação, fundimos os blocos L_5 com L_6 e L_3 com L_4, usando o truque de manipulação de índices que nos dá o programa na Figura 4.5 (a). As divergências ainda podem acontecer, porém, enquanto o maior caminho divergente na Figura 4.2 contém sete instruções, o pior caso de divergência na Figura 4.5 (b) contém cinco instruções. Essa otimização resulta numa aceleração de 6,75%.

O código da Figura 4.5 (b) ainda apresenta oportunidades de otimização. Pode-

mos usar os seletores ternários disponíveis em CUDA para fundir os blocos básicos que fazem a manipulação de índices, removendo a divergência gerada pelo desvio `bra %p2 L3`. A Figura 4.5 (c) apresenta o código fonte modificado é o código PTX gerado é mostrado na Figura 4.5 (d). A instrução `%a = sel %tid %ixj %p` copia o valor de `%tid` à `%a` se `%p` é verdadeiro, caso contrário, copia o valor de `%ixj`. Nesse novo programa, o maior caminho divergente tem somente duas instruções. Esta otimização final obteve uma aceleração de 9.2%.

4.3.1 Otimizando o algoritmo SRAD do benchmark Rodinia

O algoritmo SRAD [Yu & Acton, 2002] da suite Rodínia foi outro caso interessante que encontramos entre os *benchmarks* disponíveis publicamente. SRAD é uma sigla para *Speckle Reducing Anisotropic Diffusion*, algo como Difusão Anisotrópica Redutora de Ruído. Esse algoritmo é usado para remover o ruído das imagens, preservando características importantes da imagem. O algoritmo precisa para processar cada pixel de uma imagem, que é representada como uma matriz bidimensional. A implementação presente no Rodinia 1.0 trata os pixels da borda como casos especiais. Assim, ele contém dois desvios divergentes; um com quatro caminhos possíveis e o outro com oito. A Figura 4.6 apresenta o menor desses desvios, como está na implementação original.

Como podemos ver na Figura 4.6, cada desvio simplesmente codifica uma atribuição diferente para as variáveis `cn`, `cs`, `cw` e `ce`. Usando o mesmo truque de manipulação de índices da seção anterior, fomos capazes de agrupar todas essas atribuições em um conjunto mais amigável para arquiteturas SIMD. Primeiramente decidimos quais valores serão usados como fontes, em segundo lugar encontramos quais endereços serão usados como destinos. A Figura 4.7 apresenta o código otimizado. Modificando os dois grandes desvios divergentes de SRAD, obtivemos uma aceleração de 11,5% e relatamos esses ganhos para os mantenedores do Rodínia.

GPUs precisam de novas técnicas de otimização As técnicas de otimização que aumentam a eficiência *kernels* CUDA podem não ser aplicáveis em programas sequenciais. Por exemplo, o *kernel* na Figura 4.5 (d) possui uma sequencia de 10 instruções quando a condição do desvio é falsa. Por outro lado, a sequência original, na Figura 4.2, contém 9 instruções. Para medir o desempenho relativo entre essas duas versões em uma máquina sequencial, compilamos os dois para x86, usando `gcc -O0` para preservar as sequências de instruções originais, e executamos em uma máquina Intel x86, com frequência de 1.2 GHz e 2 GB de memória. Em todas execução o algoritmo teve

```
if (ty == BLOCK_SIZE -1 && tx == BLOCK_SIZE - 1) { // SE border
    cn  = cc;
    cs  = south_c[ty][tx];
    cw  = cc;
    ce  = east_c[ty][tx];
} else if (tx == BLOCK_SIZE -1) { // Eastern border
    cn  = cc;
    cs  = c_cuda_temp[ty+1][tx];
    cw  = cc;
    ce  = east_c[ty][tx];
} else if (ty == BLOCK_SIZE -1) { // Southern border
    cn  = cc;
    cs  = south_c[ty][tx];
    cw  = cc;
    ce  = c_cuda_temp[ty][tx+1];
} else { // The data elements which are not on the borders
    cn  = cc;
    cs  = c_cuda_temp[ty+1][tx];
    cw  = cc;
    ce  = c_cuda_temp[ty][tx+1];
}
```

Figura 4.6: Um desvio divergente encontrado no algoritmo SRAD do benchmark Rodinia [Che et al., 2009].

```
float* ps = (ty == BLOCK_SIZE -1)? &(south_c[ty][tx]) :
    &(c_cuda_temp[ty+1][tx]);
float* pe  = (tx == BLOCK_SIZE -1)? &(east_c[ty][tx]) :
    &(c_cuda_temp[ty][tx+1]);

cn = cc;
cw = cc;
cs = *ps;
ce = *pe;
```

Figura 4.7: Resultado da otimização dos desvios da Figura 4.6.

que ordenar um vetor com 4 milhões de inteiros. Observamos que o código original, equivalente ao da Figura 4.2, é 19% mais rápido que o da Figura 4.5 (a) e 11% mais rápido que o da Figura 4.5 (c), o oposto do que acontece na GPU. Obtivemos um resultado diferente com `gcc -O1`: o compilador produz o mesmo código para os fontes das Figuras 4.5 (a) e (c), que é 6% mais rápido que o programa original.

4.4 Experimentos

Foram avaliados experimentalmente 20 aplicações CUDA disponíveis publicamente, instrumentando 80 *kernels* e encontrando divergência em 67 deles. Os experimentos foram executados em uma GPU Nvidia GeForce GTX 470, com 448 *CUDA cores*. Foram avaliadas as aplicações abaixo, sendo que quicksort foi obtido de [Cederman & Tsigas, 2009], Cudaseg está disponível em [Mostofi, 2009], scan faz parte do CUDA SDK [SDK, 2011] e as outras aplicações são parte do *Rodinia Benchmark Suite* [Che et al., 2009].

DirectX Texture Compression (DXTC): Também conhecido como *S3 Texture Compression*, é um grupo de algoritmos de compressão de texturas com perdas, originalmente desenvolvido por Iourcha et al. da S3 Graphics, Ltd. A implementação presente no CUDA SDK [Castaño, 2007] realiza compressões de alta qualidade no formato DXT1. Foi utilizado o algoritmo *cluster fit* descrito por Simon Brown [Brown, 2006].

Finite-difference time-domain (FDTD3d) O Método de diferenças finitas no domínio do tempo é uma técnica de modelagem eletromagnética muito popular. Como é um método no domínio do tempo, suas soluções podem abranger uma ampla faixa de frequências em uma simulação e ele pode tratar propriedades não-lineares de materiais de uma forma natural, ao contrário de técnicas no domínio da frequência, que têm que trabalhar com um número limitado de frequências e são obrigadas as aproximar propriedades não-lineares ou tratá-as de forma especial. O método FDTD pertence à classe de métodos de modelagem numérica baseada em grade, com equações diferenciais no domínio do tempo. As equações de Maxwell dependentes de tempo (na forma de derivadas parciais) são discretizadas por meio de aproximações utilizando a diferença central para as derivadas parciais do espaço e do tempo. As equações de diferenças finitas resultantes são resolvidas de maneira alternada: os componentes do vetor campo elétrico em um volume de espaço são resolvidos em um dado instante no tempo, em seguida, os componentes do vetor campo magnético no mesmo volume são resolvidos no instante seguinte de tempo, e o processo é repetido várias vezes até acabar a simulação. Existem simulações em uma ,duas ou três dimensões, a simulação no CUDA SDK é de três dimensões.

Mersene Twister: Um algoritmo para geração de números pseudo-aleatórios desenvolvido por Makoto Matsumoto e Takuji Nishi-

mura [Matsumoto & Nishimura, 1998b]. Esse algoritmo possui várias propriedades interessantes, como período longo, uso eficiente de memória, boas propriedades de distribuição e alto desempenho. Porém a aceleração desse algoritmo por meio de GPU não é trivial. Esse algoritmo, como a maioria dos geradores pseudo-aleatórios, é iterativo, sendo difícil de paralelizar iterações de uma única instância, porém, uma aplicação necessita ter milhares de *threads* para utilizar adequadamente os recursos da GPU. Uma solução é ter muitas instâncias do gerador executando em paralelo. Para evitar que essas instâncias executando em paralelo gerem sequências correlacionadas, a implementação em CUDA desse gerador utiliza a biblioteca dcmt [Matsumoto & Nishimura, 1998a], criada por Matsumoto e Nishimura. Essa biblioteca recebe o identificador de uma *thread* e codifica esse valor nos parâmetros do Mersenne Twister, de modo que cada *thread* possa atualizar sua instância do gerador independente das outras e o resultado final ainda tenha boa aleatoriedade.

Scan: Soma de prefixos [Nguyen et al., 2007].

Backpropagation: O Algoritmo de Propagação reversa é um algoritmo de aprendizado de máquina que treina os pesos de nodos conectados em uma rede neuronal multicamadas. Essa aplicação possui duas fases: a fase direta, na qual as ativações são propagadas da camada de entrada para a camada de saída e a fase reversa, onde a diferença (erro) entre os valores observados e corretos na camada de saída são propagados de trás para frente para ajustar os pesos. O código para GPU é baseado em uma implementação da Universidade de Carnegie Mellon [Che et al., 2009].

Breadth-First Search (BFS): Busca em largura em grafos. A implementação usada no Rodínia do BFS [Harish & Narayanan, 2007] atravessa o gráfico em níveis, realizando uma chamada de *kernel* a cada nível e uma vez por nível é visitado, ele não será mais visitado novamente. Ela utiliza dois vetores de booleanos (com uma posição para cada vértice), um diz se o vértice já foi visitado e o outro diz se ele faz pate da fronteira do BFS (a fronteira corresponde aos nós que estão no nível que está sendo processado). Cada *thread* é responsável por um vértice do grafo, a cada iteração, os vértices que estão na fronteira são mascados como visitados e seus vizinhos são adicionados à fronteira da próxima iteração se não foram visitados anteriormente.

Computational Fluid Dynamics (CFD): Resolvedor de Dinâmica de fluídos. Foi utilizado código implementado por Corrigan *et al.* [Corrigan et al., 2009]. Esse

programa é um resolvedor de equações eulerianas em três dimensões para fluxo comprimível em volumes finitos em grades não estruturadas. A utilização da memória da GPU é melhorada reduzindo-se os acessos à memória global e sobreposição computação redundante, além de uma representação de dados mais apropriada. O resolvedor tem duas versões: uma com fluxos pré-computados e outra com computações de fluxos redundantes. Nos testes foram utilizadas a versão de fluxos redundantes, que apesar de fazer cálculos redundantes é mais rápida. Ler os valores pré-calculados causa muitos acessos a memória, o que acaba sendo mais lento que recalculá-los.

HeartWall: Essa aplicação [Szafaryn et al., 2009] monitora o movimento do coração de um rato em uma sequência de 104 imagens de ultrassom com resolução de 609x590 *pixels*. Inicialmente, o programa executa vários passos de processamento de imagens – detecção de bordas, SRAD despeckling (também faz parte do Rodinia, descrito mais à frente), transformação morfológica e dilatação – na primeira imagem para detectar as formas parciais das paredes internas e externas do coração. Para fins de rastreamento, a aplicação reconstrói formas completas aproximadas das paredes do coração gerando elipses que são sobrepostas sobre a imagem e amostradas para marcar pontos nas paredes do coração. No estágio final, o programa acompanha a mudança de forma das paredes detectando o movimento dos pontos pela sequência de imagens.

HotSpot: HotSpot [Huang et al., 2006] é uma ferramenta muito usada para estimar a temperatura de processadores baseado no desenho e estimativas da potência de geração de calor de cada parte do *chip*. A simulação térmica resolve iterativamente uma séria de equações diferenciais por bloco. Cada célula da grade representa a temperatura média da área correspondente no chip. Esse código é uma reimplementação do resolvedor de equações diferenciais termais transientes do HotSpot. As entradas do programa são a potência de geração de calor e as temperaturas iniciais.

Needleman-Wunsch: Needleman-Wunsch é um método de otimização global não linear de alinhamento de sequências de DNA parecido com o algoritmo Smith-Waterman [Smith & Waterman, 1981] que usamos para fundir caminhos divergentes, descrito na Seção 6.1. Os potenciais pares de sequências são organizados em uma matriz 2D. No primeiro passo, o algoritmo preenche a matriz da extremidade superior esquerda até a inferior direita, passo a passo. O alinhamento ótimo é o caminho pela matriz com a maior pontuação, onde a pontuação de uma

célula é o valor do caminho com maior peso que termina naquela célula. Assim, o valor de cada célula depende dos valores das células à esquerda, diagonal superior esquerda e acima. Em um segundo passo, o melhor caminho é percorrido de trás para frente para se deduzir o alinhamento ótimo.

Speckle Reducing Anisotropic Diffusion (SRAD): Método de difusão para aplicações de processamento de imagens de ultra-som e radar, baseado em equações diferenciais parciais (EDPs). Ele é usado para remover ruído correlacionado localmente, conhecido como *speckles*, sem destruir características importantes da imagem. As entradas do programa são imagens de ultrassom e o valor de cada posição da matriz depende dos seus quatro vizinhos.

Stream Cluster: Esta é uma versão modificada do programa *streamcluster* disponível no benchmark *Parsec* [Bienia et al., 2008], desenvolvido na Universidade de Princeton. Dada uma sequência de pontos de entrada, este programa encontra um número predeterminado de medianas, de modo tal que cada ponto é associado ao centro mais próximo. A qualidade do agrupamento é medida pela métrica da soma dos quadrados das distâncias.

Cudaseg: Segmentação de imagens biomédicas.

Quicksort: Algoritmo de ordenação quicksort [Cederman & Tsigas, 2009]. Quando a partição se torna pequena, utiliza o algoritmo de ordenação bitônica usado como exemplo da Seção 4.1.

Nessa seção apresentaremos apenas resultados de *kernels* que apresentaram pelo menos um desvio divergente. Dessas aplicações, várias possuem *kernels* com tamanho significativo, a Tabela 4.1 apresenta o tamanho de cada *kernel*.

Medindo o custo da instrumentação: A Tabela 4.1 fornece uma ideia do custo imposto pela instrumentação. A coluna BC informa a fonte de onde foi obtida a aplicação. "S" significa que a aplicação foi obtida do Cuda SDK [SDK, 2011], "R" que dizer que que foi obtida do Rodinia [Che et al., 2009] e "O" significa que foi obtida de outra fonte. A coluna LoC é o número de linhas de código do *kernel* original; OSz é tamanho do *kernel*, em instruções PTX; ISz é tamanho do código instrumentado (também em instruções PTX); TmO é tempo de execução do código original (em μs). OvI é a proporção entre o tempo de execução do código instrumentado e o do código original (como percentual), de forma que 200% significa que o código instrumentado executou no dobro do tempo do original. Finalmente, TND são as divergências encontradas. Pode-se ver que

BC	Aplicação	*Kernel*	LoC	OSz	ISz	TmO	OvI (%)	TND
S	DXTC	`dc.cs`	420	977	1316	28K	1099	1.5M
S	FDTD3d	`fd.fl`	114	1425	1907	94K	275	9.3M
S	MersenneTwister	`mr.bu`	16	90	156	422K	302	37.8M
S	MersenneTwister	`mr.ru`	48	174	214	430K	289	0
S	Reduction	`rn.26`	56	79	171	56K	744	6.4K
S	Reduction	`rn.32`	56	62	128	1K	116	100
S	ScalarProd	`sd.su`	47	97	241	120	757	1.5K
S	Scan	`sc.pn`	24	138	256	64K	153,411	2.2M
S	Scan	`sc.ud`	19	40	67	20K	23,179	197.3K
S	threadFenceReduction	`tn.sn`	112	151	334	6K	504	12.9K
S	threadFenceReduction	`tn.ml`	62	74	153	73	402	64
R	Backpropagation	`bp.ld`	33	102	178	1038	42,857	126.4K
R	Backpropagation	`bp.as`	55	94	121	1202	643	1
R	BFS	`bf.Kl`	18	57	136	1673	3,727	1.0M
R	BFS	`bf.K2`	13	28	68	1326	10,734	126.9K
R	CFD	`cf.cx`	127	1039	1209	125K	100	9.3M
R	Heartwall	`hw.kl`	1327	1675	3392	418K	14,007	78.5M
R	Hotspot	`hp.cp`	112	236	471	409	54,321	45.3K
R	Needleman-Wunsch	`nw.n1`	80	274	379	79K	3,712	5.2M
R	Needleman-Wunsch	`nw.n2`	83	277	382	78K	3,780	5.6M
R	SRAD	`sr.s1`	154	285	468	8K	4,061	721.7K
R	SRAD	`sr.s2`	99	133	212	5K	5,461	328.6K
R	StreamCluster	`st.pl`	44	107	199	2M	12,687	2.9M
O	Cudaset	`cs.ui`	103	393	615	79M	545	184.2M
O	Quicksort	`qs.Lt`	270	568	1336	2M	939	72.1M
O	Quicksort	`qs.P1`	111	231	453	133K	886	2.9M
O	Quicksort	`qs.P2`	66	90	208	227K	765	2.6M
O	Quicksort	`qs.P3`	27	43	96	78	177	1
	Totais (quando aplicável)		3696	8939	14866	85M		416.7M

Tabela 4.1: **Custo da instrumentação.** LoC: linhas de código CUDA do *kernel* original; OSz: tamanho do *kernel*, em instruções PTX; ISz: tamanho do código instrumentado (PTX); TmO: tempo de execução do código original (μs); OvI: aumento do tempo de execução devido à instrumentação (%), 100% é o tempo de execução do código original; TND: divergências encontradas.

a abordagem de instrumentação utilizada adiciona um custo substancial para o *kernel* alvo. Em termos de tamanho do código, o programa instrumentado tende a crescer entre 2 e 3 vezes. O custo é ainda maior em termos de tempo, multiplicando o tempo de execução por um fator de até 1.500 na soma de prefixos (`scan.pn`). Esse custo é perceptível mesmo em *kernels* que não apresentam divergências, como `mr.ru`. No entanto, estes números são semelhantes aos de outras instrumentações [Mytkowicz et al., 2010]. Mais importante: essa instrumentação não altera a semântica do programa, já que não utiliza nenhuma de suas variáveis. Assim, observando as divergências não mudamos o padrão de divergências original do programa.

B	Aplicação	*Kernel*	Blocos Básicos	Desvios Condicionais	Desvios Divergentes	MD / NV	MV
S	DXTC	`dc.cs`	47	25	14	663K/1.0M	16.2M
S	FDTD3d	`fd.fl`	74	36	21	944K/944K	944K
S	MersenneTwister	`mr.bu`	9	4	1	38M/38M	38M
S	MersenneTwister	`mr.ru`	4	2	0	0	76M
S	Reduction	`rn.26`	14	6	1	6.4K/51.7K	26M
S	Reduction	`rn.32`	10	4	1	100/100	100
S	ScalarProd	`sd.su`	19	10	2	1.3K/20.4K	32.7K
S	Scan	`sc.pn`	16	8	3	1M/14M	14M
S	Scan	`sc.ud`	3	1	1	197K/1.5M	1.5M
S	threadFenceReduction	`tn.sn`	27	13	3	6.4K/25.6K	1.6M
S	threadFenceReduction	`tn.ml`	12	5	1	64/256	16.3K
R	Backpropagation	`bp.ld`	10	5	3	61K/131K	131K
R	Backpropagation	`bp.as`	2	1	1	1/32K	32K
R	BFS	`bf.K1`	8	5	3	126K/375K	375K
R	BFS	`bf.K2`	4	2	1	126K/375K	375K
R	CFD	`cf.cx`	26	12	2	9.1M/18.2M	18.2M
R	Heartwall	`hw.kl`	221	131	62	76M/144M	144M
R	Hotspot	`hp.cp`	34	17	9	15K/15K	15K
R	Needleman-Wunsch	`nw.n1`	17	7	5	2M/2.1M	2.1M
R	Needleman-Wunsch	`nw.n2`	17	7	5	2M/2.1M	2.1M
R	SRAD	`sr.s1`	32	13	8	9.2M/9.2M	9.2M
R	SRAD	`sr.s2`	13	5	3	9.2M/9.2M	9.2M
R	StreamCluster	`st.pl`	14	6	2	2.6M/3.3M	844M
O	Cudaseg	`cs.ui`	44	16	11	47M/366M	366M
O	Quicksort	`qs.Lt`	101	58	34	25M/32M	32M
O	Quicksort	`qs.P1`	32	16	8	2M/2M	2M
O	Quicksort	`qs.P2`	14	8	4	2M/2M	2M
O	Quicksort	`qs.P3`	5	3	1	1/15	57
	Totais (quando aplicável)		829	426	210		1606M

Tabela 4.2: **Ocorrência de divergências em aplicações de propósito geral.** B: Benchmarks; S: CUDA sdk, R: Rodinia, O: outro; MD: número máximo de divergências; NV: número de vezes que o desvio MD foi visitado; MV: número máximo de vezes que algum desvio foi visitado.

Divergências: A Tabela 4.2 apresenta o número de divergências nos programas testados. A coluna B informa a fonte de onde foi obtida a aplicação. Assim como na Tabela 4.1, "S" significa que a aplicação foi obtida do Cuda SDK [SDK, 2011], "R" que dizer que que foi obtida do Rodinia [Che et al., 2009] e "O" significa que foi obtida de outra fonte. MD é o número de divergências do bloco básico mais divergente, NV é o número de visitas desse mesmo bloco e MV é o número de visitas do boco básico mais visitado. Pode-se notar que uma grande proporção de desvios condicionais presentes nos *kernels* (entre 50% e 60%) gera divergências. Na tabela também indicamos o desvio com maior número de divergências por *kernel*. A partir dessas informações pode-se perceber que alguns pontos do código apresentam uma quantidade não ne-

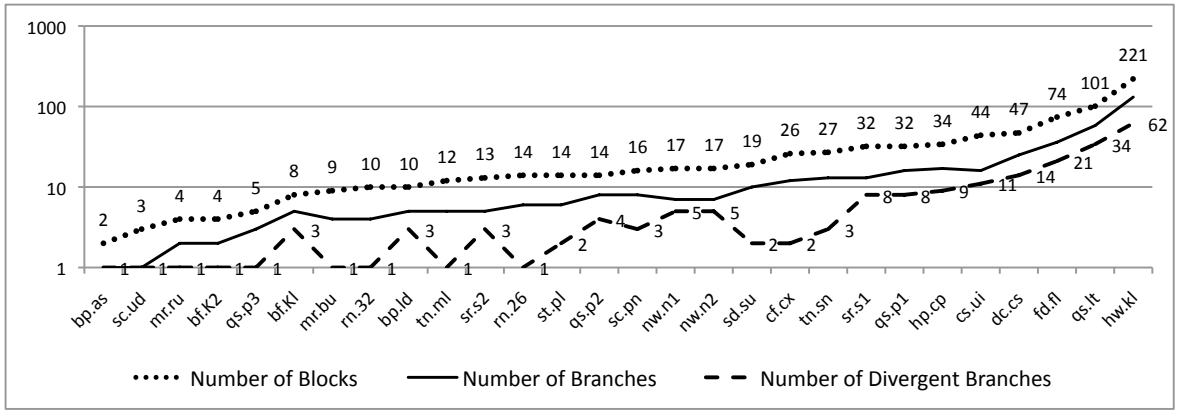

Figura 4.8: Número de blocos básicos, desvios condicionais e desvios divergentes por *kernel*.

gligenciável de divergências. É interessante notar que esses programas passaram por muitos desenvolvedores e são muito otimizados, mas ainda assim apresentam muitas divergências. Pela tabela, também pode-se notar que os desvios mais visitados tendem a ser os mais divergentes, embora existam exceções, como `dc.cs`, `rn.26`, `sd.su`, `tn.sn`, `tn.ml` e `st.pl`. Portanto, técnicas de monitoração tradicionais, como as empregadas na ferramenta `gprof` [Graham et al., 1982b], podem identificar facilmente as regiões mais executadas do código, mas elas não necessariamente serão as maiores fontes de divergências.

Os *kernels* na Tabela 4.2 contêm 829 blocos básicos, dos quais 426 possuem desvios condicionais. Cerca de metade dos desvios condicionais (211 desvios) é divergente. As divergências são bastante bem distribuídos entre os *kernels*. A Figura 4.8 representa estes números, para cada *kernel*. Nela, os *kernels* estão ordenados pelo número de blocos básicos em ordem crescente. Note que o eixo Y está em escala logarítmica. A linha pontilhada representa o número de blocos básicos por *kernel*, já a linha contínua é o número de blocos básicos com desvios condicionais e, finalmente, a linha tracejada representa o número de desvios condicionais que são divergentes. Podemos ver que, em média, cerca de um quarto dos blocos básicos de um *kernel* típico possuem desvios divergentes.

A maioria das divergências ocorre em poucos desvios: Este fato segue-se naturalmente a partir do conhecimento comum que poucas partes do código são responsáveis por maior parte do tempo de execução do programa. Portanto, a maioria das divergências estarão concentrados nos desvios mais executados. A Tabela 4.3 ilustra esse

Kernel	Hot	BR	*Kernel*	Hot	BR
dc.cs	8	0.57	fd.fl	9	0.43
mr.bu	1	1.00	mr.ru	0	0.00
rn.26	1	1.00	rn.32	1	1.00
sd.su	2	1.00	sc.pn	2	0.25
sc.ud	1	1.00	tn.sn	2	0.67
tn.ml	1	1.00	bp.ld	3	0.60
bp.as	1	0.10	bf.K1	3	0.60
bf.K2	1	0.50	cf.cx	1	0.08
hw.kl	1	0.01	hp.cp	4	0.24
nw.n1	4	0.57	nw.n2	4	0.57
sr.s1	6	0.47	sr.s2	2	0.40
st.pl	1	0.17	cs.ui	9	0.56
qs.Lt	5	0.09	qs.P1	1	0.06
qs.P2	1	0.12	qs.P3	1	0.33

Tabela 4.3: **Concentração de divergências** Hot: número de desvios responsáveis por pelo menos 90% das divergências. BR: desvios "Hot" divididos pelo total de desvios do *kernel*.

padrão inerente a todos os *benchmarks*. Notamos que 13 *kernels* têm mais de 90% de suas divergências em um único desvio e que em apenas quatro *kernels* – `dc.cs`, `sr.s1`, `fd.fl` e `cs.ui` – contêm mais de seis desvios altamente divergentes. Este padrão de concentração parece apontar que o desenvolvedor deve focar em poucas regiões do programa quando implementa otimizações que reduzam o efeito das divergências sobre aplicações SIMD. Por exemplo, o *kernel* `partition1` (`qs.P1`) do quicksort paralelo contém 16 desvios, mas apenas um concentra mais de 90% de todas as divergências que ocorreram durante a execução.

Desvios executados frequentemente e raramente têm chances similares de causar divergências. Como já discutimos antes, a maioria das divergências é causada por poucos desvios. No entanto, a chance de um desvio ser divergente, pelo menos em nossos testes, parece ser independente do número de vezes que o desvio é executado. Ou seja, desvios que são executadas raramente, tais como declarações "if-then-else" fora de laços e desvios frequentemente executados, tais como laços aninhados profundamente, parecem ter chances semelhantes de causar divergências. Para ilustrar isso, a Figura 4.9 contém histogramas dos quatro maiores *kernels* utilizados nos nossos testes. Em todos os histogramas, a linha mais grossa é o número de vezes que o desvio foi visitado (os desvios estão ordenados em ordem crescente do número de visitas) e a área cinza representa o número de divergências. Note que o eixo Y está em escala logarítmica. Na figura, o *kernel* principal do programa Heartwall do Rodinia, `hw.kl`, tem um total de

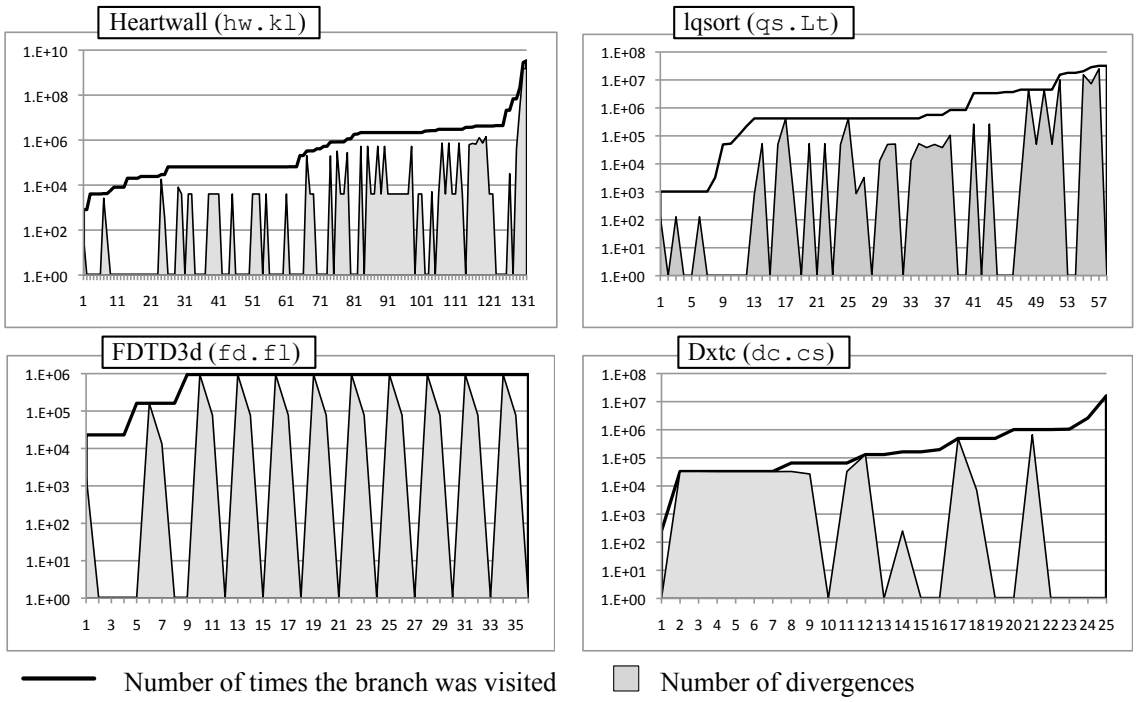

Figura 4.9: Histograma das divergências dos quatro maiores *kernels* em nossos *benchmarks*.

1.537.782.862 divergências, de 3.372.281.211 desvios visitados. Embora a maioria das divergências (1.496.652.630, no último desvio do histograma) sejam causadas por um único desvio, que foi visitado 2.839.397.460 vezes, o histograma indica que divergências podem acontecer mesmo nos desvios raramente executados. Temos notado um padrão similar em outros programas. Como exemplo, mostramos também histogramas para `qs.Lt`, `fd.fl` e `dc.cs`, nossos outros três maiores *kernels*.

Parece não existir correlação entre o número de vezes que um desvio é visitado e a proporção dessas visitas que divergem. A Figura 4.10 apresenta a fração das divergências em relação ao total de visitas, para todos os desvios dos programas testados que receberam pelo menos uma visita (385 desvios). Cada ponto representa um desvio e os desvios são ordenados ao longo do eixo X em ordem crescente do número de visitas. O ponto mais à esquerda representa um desvio em `qs.P3`, que foi visitado 6 vezes e não causou divergências. O último desvio vem de `hw.kl`, que, como já dissemos antes, foi visitado 2.839.397.460 vezes e causou 1.496.652.630 divergências. Em termos de números absolutos, 22,11% de visitas resultaram em divergências. A média geométrica de divergências foi de 0,16 divergências por visita. O coeficiente de correlação entre

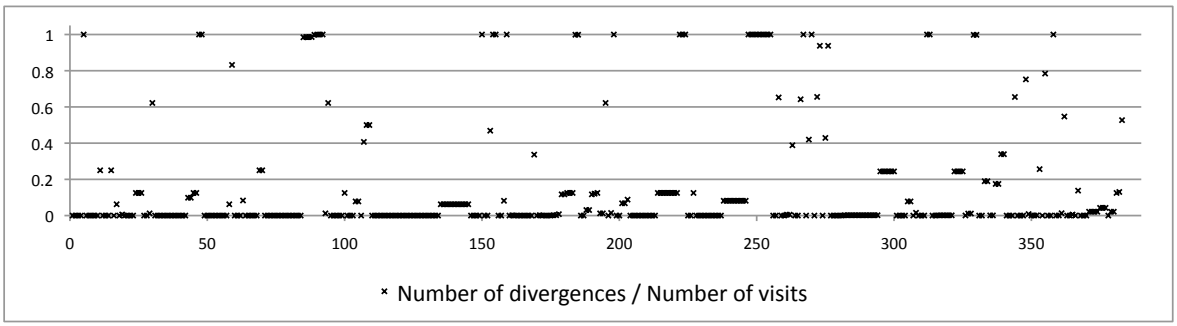

Figura 4.10: Proporção de divergências. O gráfico contém um ponto para cada desvio em nossos *benchmarks*. O eixo X está ordenado pelo número de vezes que o desvio foi visitado, do menos visitado para o mais visitado.

as visitas e divergências é de 0,35, o que indica correlação fraca [Jain, 1991]. Este coeficiente denota a dependência entre duas variáveis A e B. Um coeficiente de 1 indica que A e B ocorrem juntos, 0 indica que A e B são estatisticamente independentes e -1 indica que B tende a ocorrer quando A não ocorre.

4.5 Conclusão

Neste capítulo apresentamos uma técnica de monitoração dinâmica para análise de divergências. Essa técnica permite que desenvolvedores implementem programas que executem de forma mais eficiente arquiteturas SIMD, como as GPUs da Nvidia que executam código CUDA. Ela mostra os desvios que causaram divergências durante uma execução, degradando o desempenho, e informa o número de divergências ocorridas em cada desvio, permitindo que os programadores priorizem os desvios mais divergentes. Porém essa técnica tem uma limitação, seus resultados valem só para a entrada utilizada durante a execução, ou seja, não podem ser generalizados para outras entradas. No próximo capítulo apresentaremos uma técnica de análise estática de divergências que complementa a monitoração dinâmica com resultados menos detalhados sobre o comportamento do programa, mas que valem para todas entradas.

Capítulo 5

Análise Estática de Divergências

Neste capítulo vamos mostrar nossa análise estática para detecção de desvios divergentes. Inicialmente apresentaremos essa técnica de forma simplificada e informal e nas Seções 5.1 e 5.2 a descreveremos detalhadamente e com mais formalismo.

O objetivo da análise é verificar se um desvio é divergente ou não sem ter que executar o programa. Como essa análise é feita em tempo de compilação, não se conhece a entrada do programa, o que pode afetar o resultado da análise em alguns casos. Assim, o algoritmo é um aproximação conservadora: alguns desvios não divergentes podem ser considerados como divergentes, mas o contrário nunca irá acontecer.

```
if (y > 0) {
  x = 1;
} else {
  x = 0;
}
a = x + 1;
```

Figura 5.1: Desvio Original.

```
pred = (y > 0);
if (pred) {
  x = 1;
} else {
  x = 0;
}
a = x + 1;
```

Figura 5.2: Desvio usando predicado.

A linguagem PTX [PTX, 2010] utiliza predicados, variáveis booleanas que controlam se uma instrução vai ser executada ou não. Podemos adicionar predicados a qualquer instrução e quando o fazemos, a instrução só é executada se o predicado for verdadeiro. Dessa forma, desvios condicionais são implementados adicionando-se predicados a instruções de desvio incondicional. Assim um desvio como o da Figura 5.1 é implementado na forma mostrada na Figura 5.2.

Para verificar se um desvio pode ser divergente, precisamos apenas analisar se a variável `pred`, utilizada como condição do desvio, pode ter valores diferentes entre as

threads de um *warp*, o que chamamos de *variável divergente*. Para saber se as variáveis que controlam os desvios são divergentes, temos que montar um grafo de dependência entre variáveis do *kernel* e caso uma variável seja dependente de uma variável divergente ela também será divergente. Caso contrário, ela garantidamente terá o mesmo valor entre todas as *threads* de um *warp*. Existem variáveis que não são divergentes por definição:

Argumentos do *kernel*: obrigatoriamente têm os mesmos valores para todas as *threads* que estão executando o *kernel*.

Dados em qualquer memória da GPU (exceto memória local): se o endereço de leitura for o mesmo, as *threads* receberão o mesmo valor.

Enquanto outras são divergentes por definição:

O identificador da *thread* (`threadIdx`): Cada *thread* tem um valor diferente.

Dados em qualquer memória da GPU lidos usando instruções atômicas: essas instruções garantem que cada *thread* receberá um valor diferente mesmo se estiverem operando sobre o mesmo endereço.

Uma dependência usada para montar o grafo de dependências é a de dados: ela acontece quando uma variável é utilizada no calculo da outra. Por exemplo, se o código tem uma linha como $c = r * 3.14;$, então existe uma dependência de dados de c sobre r.

```
pred = (y > 0);
if (pred) {
  x1 = 1;
} else {
  x2 = 0;
}
x3 = φ(x1, x2);
a = x3 + 1;
```

Figura 5.3: Desvio no formato SSA.

```
pred = (y > 0);
if (pred) {
  x1 = 1;
} else {
  x2 = 0;
}
x3 = φ(pred, x1, x2);
a = x3 + 1;
```

Figura 5.4: Dependência de sincronização transformada em dependência de dados.

A outra dependência usada é a dependência de sincronização. Ela ocorre no final de um trecho de código potencialmente divergente, como um *if* ou um laço que pode influenciar no valor de uma variável, onde a GPU sincroniza as *threads* para voltarem

a executarem juntas. Nossas análises trabalham com o código no formato SSA (Static Single Assignment) onde o compilador insere funções ϕ para expressar as dependências de dados nessas regiões. Assim um trecho de código como o da Figura 5.2 é transformado para o da Figura 5.3. Nossa estratégia para lidar com essas dependências é transformá-las em dependências de dados, inserindo o predicado que controla o desvio como parâmetro da função ϕ, como na Figura 5.4.

5.1 Definições

Dado um programa μ-SIMD P, estamos interessados em determinar uma aproximação conservadora, mas não trivial, do conjunto de *variáveis divergentes*. A variável v é divergente se existem duas *threads* (t_1, σ_1) e (t_2, σ_2), tais que $\sigma_1(v) \neq \sigma_2(v)$, no mesmo momento durante a execução do programa. Para tornar esta definição independente do ponto do programa onde variável é utilizada, doravante trabalharemos com programas μ-SIMD na forma *Static Single Assignment (SSA)* [Cytron et al., 1991], que denotaremos por μ-SIMD$_\phi$.

Programas na forma SSA fazem uso de funções ϕ, que funcionam como seletores. Por exemplo, quando convertemos um trecho de código como o da Figura 5.2 para o formato SSA, adicionamos funções ϕ, como a da Figura 5.3, no início de blocos básicos que podem ser alcançados por mais de um caminho, para selecionarem o valor da variável dependendo do caminho utilizado para se chegar ao bloco básico.

Usaremos a notação padrão para representar funções ϕ SSA, como por exemplo: $v = \phi(v_1, \ldots, v_n)$. Não vamos criar novos rótulos para representar funções ϕ. Ou seja, se a conversão do programa P para SSA levar-nos a inserir uma sequência de funções ϕ antes de a instrução $P[l]$, então, representaremos todas estas funções ϕ com o rótulo l. Essa notação indica o fato de que funções ϕ são apenas uma abstração, não estão presentes no código de máquina e simplificam o Algoritmo 1, que será apresentado em breve. Usando uma análise de caso simples mais uma indução sobre as regras da Figura 3.5 podemos provar que o Teorema 5.1.1 fornece o conjunto de variáveis divergentes.

Theorem 5.1.1 *Uma variável $v \in P$ é divergente se e somente se alguma dessas condições for verdadeira:*

1. $v =$ `tid`.
2. *v é definida por uma operação atômica, ex:* `atomic`(v, v_x).
3. *existe uma* dependência de dados *de v sobre uma variável divergente.*

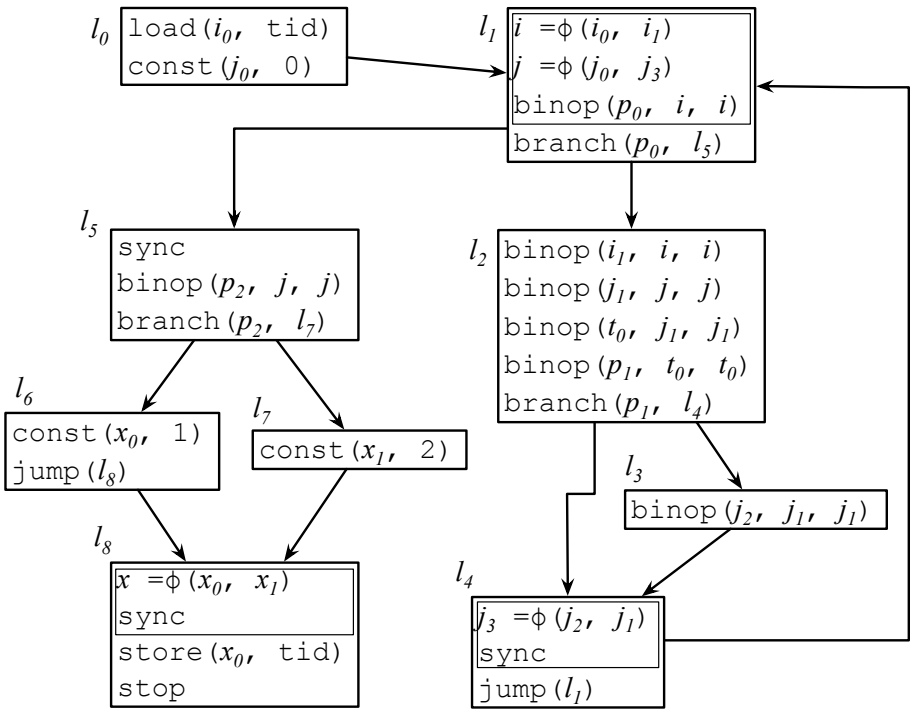

Figura 5.5: Um programa μ-SIMD$_\phi$. Instruções nas caixas (funções ϕ seguidas de uma instrução μ-SIMD) correspondem a apenas um rótulo.

4. existe uma dependência de sincronização *de v sobre uma variável divergente.*

Prova: Para explicar o Teorema 5.1.1 esclareceremos as noções de *dependência de dados* e *dependência de sincronização*. Dado um programa P, uma variável $v \in P$ possui uma dependência de dados sobre uma variável $u \in P$, se P contém alguma instrução de atribuição $P[l]$, seja ela uma função ϕ ou uma operação comum, que define v e usa u, por exemplo, $P[l] = \texttt{binop}(v, u, u)$. Usaremos o programa na Figura 5.5 para ilustrar os conceitos desta seção. Neste exemplo, vemos que a variável i_0 é divergente, pois possui uma dependência de dados sobre `tid`. Similarmente, i é divergente, uma vez que possui uma dependência de dados sobre i_0. O problema de determinar o fecho transitivo de um conjunto de variáveis divergentes (levando em conta apenas propagações de dependências de dados) é um tipo de divisão de programa [Weiser, 1981], que pode ser resolvida pela *Análise de Variância* [Stratton et al., 2010, p.115] de Stratton *et al.*. No entanto, a análise de variância, proposta originalmente para encontrar o conjunto de variáveis divergentes em um programa, pode gerar falsos-negativos, ou seja, esta análise pode reportar que uma variável, que apresenta comportamento divergente, não é divergente. Esta omissão acontece porque a análise de variância não lida com

um fenômeno que chamamos de dependência de sincronização, que será introduzida na Definição 5.1.2.

Para definir dependências de sincronização, precisamos rever alguns termos bem conhecidos dos desenvolvedores de compiladores. Um bloco básico l_p pós-domina outro bloco l, se e somente se todos os caminhos a partir de l até o termino do programa (ou seja, a instrução `stop`) atravessam l_p. Além disso, dizemos que l_p é o *pós-dominador imediato* de l se $\mathrm{l}\, l_p \neq l$ e qualquer outro bloco que pós-domina l também pós-domina l_p. Fung *et al.* [Fung et al., 2007] mostrou que re-convergindo *threads* divergentes no pós-dominador imediato do desvio divergente maximização da utilização do *hardware* de forma quase ótima. Embora o trabalho de Fung também tenha descoberto situações nas quais é melhor fazer essa re-convergência após l_p, elas são muito raras. Assim, assumiremos que o pós-dominador imediato, l_p, de um desvio divergente sempre terá uma instrução `sync`.

Definition 5.1.2 *Dada uma instrução* $P[l] =$ `branch`(p, l')*, dizemos que* v *possui uma* dependência de sincronização *sobre* p *se e somente se o valor de* v *em* l_p *(o pós-dominador de* l*) depende do resultado de* p*.*

prova: ($\Rightarrow$) Suponhamos que v possui uma dependência de sincronização sobre p. Se v não alcança l_p, então por definição, v não pode ter uma dependência de sincronização sobre p. Se v é definido fora de $IR(p)$, então, pela forma SSA sabemos que não pode ser atribuído um valor a v em qualquer caminho dentro de $IR(p)$, portanto, o resultado de `branch(p, l')` não tem qualquer influência sobre o valor de v.

($\Leftarrow$) Sejam l_1 e l_2 os dois sucessores de l (note ou $l_1 = l'$ ou então $l_2 = l'$). Se v é definida no bloco básico l_v dentro de$IR(p)$, então existe um caminho $l_1 \xrightarrow{*} l_p$ que contém l_v, e outro caminho $l_2 \xrightarrow{*} l_p$ que não. Se assumirmos o contrário, é um exercício simples derivar uma contradição em que l_p não imediatamente pós-domina l, enquanto l_v o faz. Suponhamos que nossa máquina abstrata da Figura 3.3 visite l pela primeira vez. Assim, para cada PE (t, σ), temos que inicialmente $\sigma(v) = \perp$, ou seja, possui um valor indefinido. Se assumirmos que temos um PE (t_1, σ_1) que chega a l_p através do caminho $l_1 \xrightarrow{*} l_p$, então $\sigma_1(v) \neq \perp$. Por outro lado, se temos um PE (t_2, σ_2) que chega a l_p através do caminho $l_2 \xrightarrow{*} l_p$, então $\sigma_2(v) = \perp$. Portanto, o valor de v é controlado por p e v possui uma dependência de sincronização sobre p. $\square$

5.2 Calculando variáveis divergentes

A fim de determinar o conjunto de variáveis divergentes em um programa μ-SIMD$_\phi$, usamos uma estrutura de dados de Scholz *et al.* [Scholz et al., 2008], chamada de *grafo de dependências de entrada*. Observe que, apesar de utilizar essa estrutura de dados de Scholz, nossa análise tem um propósito diferente da dele, e produz resultados muito diferentes. Nós definimos o grafo de dependências de entrada da seguinte forma:

- Para cada variável $v \in P$, seja n_v um vértice de G.
- se P contém uma instrução que define a variável v, e usa a variável u, então adiciona-se uma aresta de n_u até n_v.

A fim de encontrar o conjunto de variáveis divergentes de P, começamos a partir de n_{tid}, mais os nós que representam variáveis definidas por instruções atômicas, e marcamos cada variável que pode ser alcançada a partir deste conjunto de nós. Observe que até agora não estamos levando em consideração dependências de sincronização. Lidamos com dependências de sincronização em nosso grafo de acessibilidade convertendo-os em dependências de dados, um "feito" que realizamos, novamente, seguindo a ideia de Scholz *et al.* de *sujar* funções ϕ com os predicados que as controlam. Assim, adicionamos predicados á à sintaxe de funções ϕ de Cytron, ou seja:

$$P[l] \text{ é } v = \phi(v_1, \ldots, v_n), p_1, \ldots, p_k$$

onde cada $p_i, 1 \leq i \leq k$ é um predicado que *controla* l. Aqui usamos a definição de dependência de controle de Appel e Palsberg [Appel & Palsberg, 2003, p.425], ou seja, um predicado p controla uma função ϕ $v = \phi(v_1, \ldots, v_n)$ se a atribuição de um valor em v depende do resultado de p.

Sujando funções ϕ desta forma, adicionamos uma dependência de dados entre p_i e v; efetivamente reduzindo dependências de sincronização em dependências de dados. No entanto, ao contrário do Scholz *et al.* nós não sujamos cada função ϕ do programa, e podemos dividir os *live ranges* de algumas variáveis, via funções ϕ de um parâmetro, para indicar que uma variável pode causar divergências apenas em parte da seu *live range*. Então, quais funções ϕ sujar, e onde dividir *live ranges*, para obter uma representação do programa que nos permita resolver a análise de divergências com a maior precisão possível? Respondemos a esta questão através do Algoritmo 1 que executamos uma vez para cada desvio do programa. Para cada instrução $P[l] = \texttt{branch}(p, l)$, o Algoritmo 1 separa o conjunto de variáveis que são definidas dentro de $IR(p)$. Se $P[l]$ é um desvio à frente, ou seja, criado devido à compilação de uma diretiva "if-then-else", então simplesmente usamos p para sujar cada função ϕ no pós-dominador imediato de

Algoritmo 1 Suja funções ϕ.

Para cada instrução $P[l] = \texttt{branch}(v, l')$ com um pós-dominador imediato l_p, faça:

1. Para cada variável v, definida em $IR(p)$, alcançando l_p faça:
   a) se v é usada em l_p como parâmetro de uma função ϕ $v' = \phi(\ldots, v, \ldots)$, suja ou não, essa instrução é substituída por uma nova função ϕ suja por p.
   b) se v é usada por ima instrução de atribuição $x = f(\ldots, v, \ldots)$, em l_p ou em um rótulo l_x, dominado por l_p, então são realizadas as ações seguintes:
      i. divide-se o *live range* de v, inserindo uma instrução $v' = \phi(v, \ldots, v), p$ em l_p, com um parâmetro para cada predecessor de l_p;
      ii. renomeia-se todo uso de v para v' em l_p, ou em qualquer bloco dominado por l_p;
      iii. reconverte-se o programa para a forma SSA, uma ação necessária devido à mudança de nome realizada no passo anterior.

$P[l]$, como apresentado no passo (1.a). No entanto, se $P[l]$ implementa um laço, então dividimos o *live range* de qualquer variável que é usada fora desse laço, de acordo com o passo (1.b) do algoritmo[1]. Nós usamos funções ϕ de um parâmetro para realizar esta divisão de *live range*, e apenas as variáveis definidas por essas funções ϕ (não necessariamente os seus parâmetros) possuem dependência de sincronização sobre p.

A Figura 5.6 apresenta o resultado da execução do Algoritmo 1 sobre o programa da Figura 5.5. As funções ϕ em l_4 e l_8 foram marcadas pelo passo (1.a) deste algoritmo. A variável j_3 possui uma dependência de sincronização sobre a variável p_1 e a variável x possui uma dependência de sincronização sobre a variável p_2. A aridade uma função ϕ em l_5 foi criada pela etapa (1.b.i) do Algoritmo 1.

Uma vez usado Algoritmo 1 para marcar funções ϕ no programa, extraímos o grafo de dependências de entrada. Continuando com nosso exemplo, a Figura 5.7 apresenta o grafo criado para o programa da Figura 5.6. Surpreendentemente, observa-se que a instrução $\texttt{branch}(p_1, l_4)$ não pode causar uma divergência, embora o predicado p_1 possua uma dependência de dados sobre a variável j_1, que é criada dentro de um laço divergente. Ainda assim, a variável j_1 não é divergente, embora a variável p_0 que controla o laço o seja. Podemos provar a não-divergência de j_1 por indução sobre número de iterações do laço. Na primeira iteração, cada *thread* vê $j_1 = j_0 \otimes j_0$, como se infere da regra Bp da linguagem μ-Simd (Figura 3.5). Assumindo que na n-ésima iteração toda *thread* ainda no laço veja o mesmo valor de j, então, a atribuição $j_1 = j \otimes j$

[1] O ponto onde inserimos uma nova função ϕ corresponde ao ponto onde Ottenstein *et al.* iria inserir uma função η [Ottenstein et al., 1990, p.260].

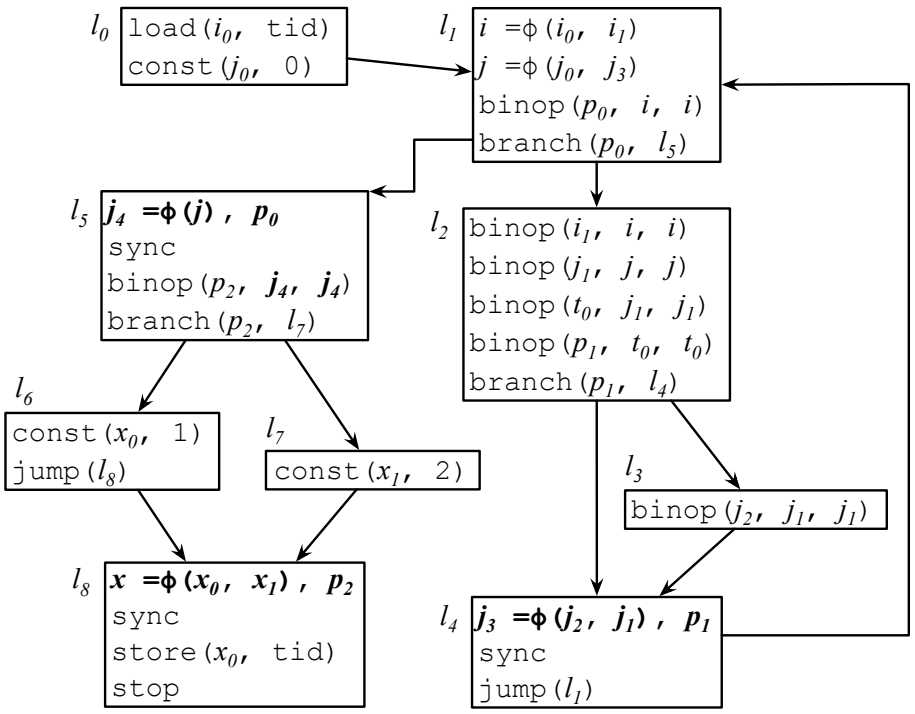

Figura 5.6: O programa da Figura 5.5, após sujar as funções ϕ. As mudanças estão marcadas em negrito.

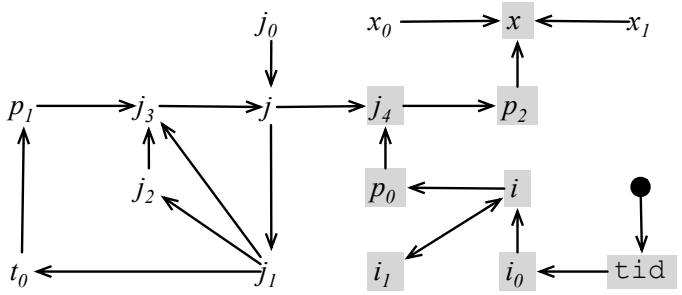

Figura 5.7: O grafo de dependências de entrada criado para o programa da Figura 5.6. Variáveis divergentes estão em cinza.

conclui o passo de indução. No entanto, a variável j pode causar uma divergência em `branch`(p_2, l_7), porque ele é definido por uma função ϕ que usa j_1. Isto é, uma vez que os PEs sincronizam em l_5, eles podem ter re-definido j_1 um número de vezes diferente. Embora esse fato não possa causar uma divergência dentro de $IR(p_0)$, como descrito no Algoritmo 1, podem acontecer divergências fora do laço controlado por p_0. Assim, dividimos o *live range* de j fora do laço, como apresentado pela instrução no rótulo l_5

na Figura 5.6.

Análise de complexidade O algoritmo é executado em duas etapas: primeiro altera-se a representação intermediária do programa e então precorre-se o grafo para identificar as variáveis divergentes. A alteração do programa envolve, no pior caso, percorrer seu grafo de fluxo de controle uma vez a cada desvio. Se tivermos $|B|$ desvios em um programa μ-SIMD, então a travessia é $O(|B|)$. Para cada desvio, podemos ter que renomear variáveis e reconverter o programa para a forma SSA, O que é $O(\alpha|V|)$ [Appel & Palsberg, 2003], onde α é a função de Auckerman inversa, e $|V|$ é o número de variáveis do programa. Assim, o Algoritmo 1 executa em $O(\alpha \times |B|^2 \times |V|)$. A segunda fase é executado em $O(|V|^2)$ no pior caso, mas em média é $O(|V|)$. Por isso, resolvemos o problema de análise de divergências em $O(\alpha \times |B|^2 \times |V| + |V|^2)$.

5.3 Experimentos

Nesta seção, descreveremos os experimentos para validar a análise de divergências. As aplicações e a configuração da máquina utilizada nos experimentos serão descritas na Seção 6.2.

Precisão da análise de divergências: A fim de verificar a precisão da análise de divergências da Seção 5.2, comparamos seus resultados com os da instrumentação descrita no Capítulo 4. O programa com a instrumentação é executado e ela verifica quais desvios causaram divergências durante a execução do programa. Os resultados desta comparação são apresentados na Figura 5.8. Na figura, cada ponto (barra na parte do fundo) é um *kernel* e em todas as partes os *kernels* estão ordenados pela taxa de acerto (apresentada no fundo). No topo da figura mostramos o número de desvios condicionais por *kernel* (742 desvios no total). Só foram incluídos na contagem de desvios do *kernel*, os desvios que foram visitados durante a execução do programa instrumentado. No meio mostramos o número de predições corretas por *kernel* (490 dos 742 desvios foram previstos corretamente). Finalmente, no fundo está a taxa de acertos para cada *kernel*. Os valores estão entre 0 e 1, assim um valor de 1.0 significa uma taxa de acerto de 100% para um determinado *kernel*. Foi obtida uma taxa de erros de média por *kernel* de 34%, devido a falsos positivos. No entanto, ressaltamos que executamos cada um dos programas do Nvidia SDK [SDK, 2011] com uma única entrada, obtendo uma taxa de falsos positivos de 39%, já nos códigos do Rodínia [Che et al., 2009] usamos dois tipos diferentes de entrada de dados, obtendo uma taxa de falsos positivos menor, de 31%. Os caminhos do programa exercitados pela instrumentação são dependentes

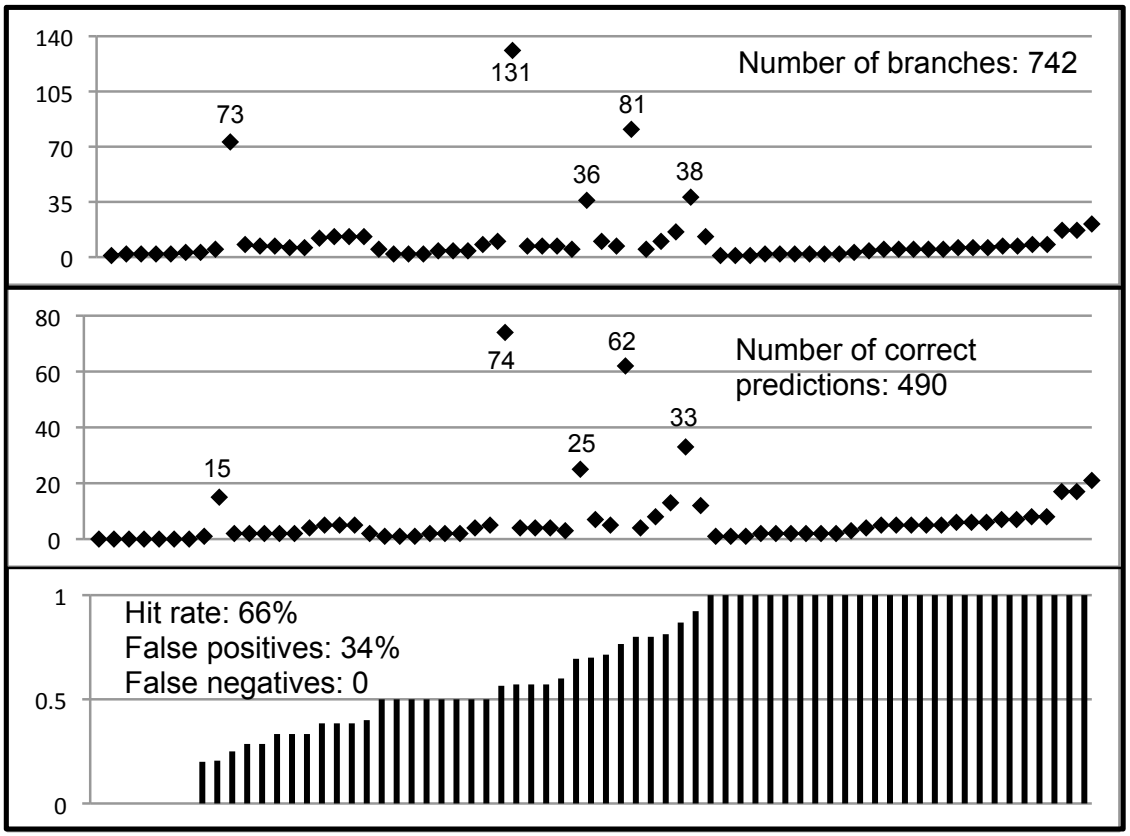

Figura 5.8: Precisão da análise de divergências. Topo: número de desvios por *kernel*. Meio: número de falsos positivos. Fundo: porcentagem de acertos. *Kernels* no eixo X estão ordenados por precisão em todas as partes.

de dados, e assim especula-se que os resultados da análise de divergências estarão mais próxima dos resultados da instrumentação assim que testarmos os programas com uma quantidade de entradas maior. Outra fonte de imprecisão é o fato de que a maioria das amostras de entrada que vêm com os *benchmarks* são ajustadas ao tamanho do *warp* da GPU (são múltiplos de 32). Neste cenário, não encontramos divergências no código que testa especificamente por situações em que o tamanho de entrada não encaixa exatamente no tamanho do *warp*.

Número de variáveis divergentes: A Figura 5.9 apresenta as variáveis divergentes por *kernel*. Cada ponto ou barra no eixo X corresponde a um *kernel* e em ambas as partes, os *kernels* estão ordenados pelo número total de variáveis. Na parte de cima está o número total de variáveis e o número de variáveis divergentes por *kernel*. Como

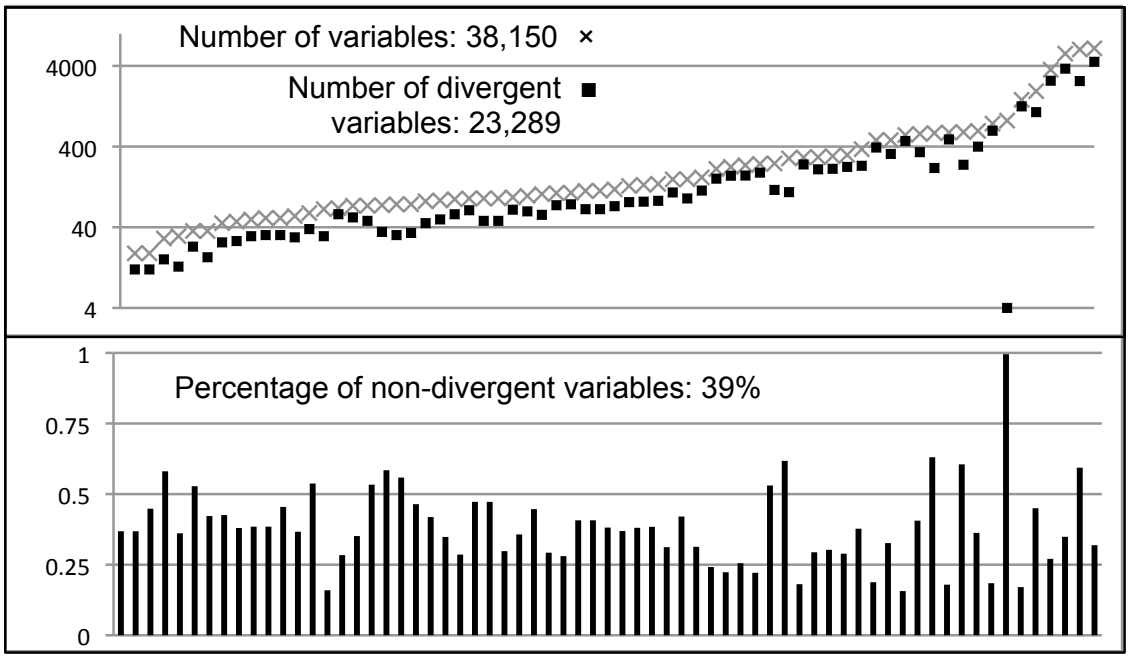

Figura 5.9: Topo: Número de variáveis divergentes (Teorema 5.1.1). Fundo: Proporção de variáveis não divergentes. *Kernels* no eixo X estão ordenadas pelo número de variáveis.

eixo Y está em escala logarítmica, parece que quase todas as variáveis são divergentes, mas, em média, 61% das variáveis é divergente. A parte de baixo mostra a proporção de variáveis não divergentes. Os valores estão entre 0 e 1 (1.0 é equivalente a 100%) e em média 39% das variáveis não são divergentes. Os 67 *kernels*, quando convertidos em formato SSA, contêm um total de 38.150 variáveis. Este número é consideravelmente maior que o número de variáveis nos programas originais, porque a representação SSA renomeia cada redefinição da mesma variável. A análise de divergências concluiu que do total de variáveis, 23.289 variáveis (61%) podem ser divergentes e 14.861 (39%) não são divergentes. Isso significa que podemos mover 39% de todas as variáveis do programa para memória partilhada; possivelmente abrindo espaço para mais *threads* na GPU. Um exceção notável neste gráfico foi o `concurrentKernels::mykernel` do SDK. Nossa análise mostrou que apenas 4 das 839 variáveis do *kernel*, podem ser divergentes.

Tempo de execução da análise: A análise de divergências tem um desempenho, que no pior caso, é quadrático em relação ao número de variáveis do programa, porque o grafo de dependências de entrada contém um vértice para cada variável, e pode ser

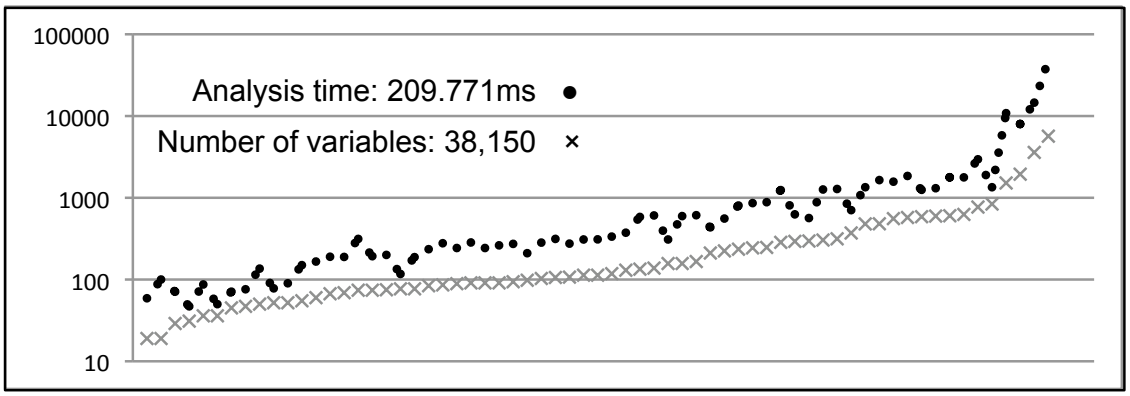

Figura 5.10: Tempo de execução (ms) da análise de divergências, comparado com o número de variáveis do programa. *Kernels* no eixo X estão ordenados pelo número de variáveis.

denso. No entanto, na prática, o tamanho desse grafo é linear no número de variáveis do programa, e os nossos experimentos demonstram este fato. A Figura 5.10 compara o tempo de execução da nossa análise (em milissegundos) com o número de variáveis por *kernel*. Cada ponto no eixo X é um *kernel* e os *kernels* no estão ordenados pelo número de variáveis. O eixo Y é o número de variáveis no *kernel* (×) ou o tempo de execução em milissegundos (•) e está em escala logarítmica. No total os *kernels* têm 38.150 variáveis e a soma do tempo das análises foi de 209.771 milissegundos (209,771 segundos). Observando a Figura 5.10, podemos perceber que existe uma relação linear entre o número de variáveis e o tempo de execução da nossa análise.

5.4 Conclusão

Neste capítulo apresentamos uma técnica de análise estática de divergências. Essa técnica tem as vantagens de que seus resultados valem para todas as execuções do programa analisado, além de não precisar de que o código analisado seja executado, permitindo que seja utilizada pelo próprio compilador. Porém, não tendo acesso aos dados de entrada do programa, nossa análise estática é mais conservadora que a técnica de *profiling* vista no capítulo anterior. Assim, é possível que a análise de divergências marque algumas variáveis como divergentes, ainda que essas variáveis nunca apresentem tal comportamento durante a execução do programa. No próximo capítulo apresentaremos uma técnica de otimização de código divergente que usa a técnica de análise estática

apresentada nesse capítulo para encontrar os trechos de código que deve otimizar.

Capítulo 6

Otimizações de código divergente

A análise de divergências é útil por duas razões. Primeiro, ela mostra ao programador da aplicação as regiões do código que causam degradação de desempenho devido a divergências. Segundo, a identificação de variáveis divergentes fornece informações úteis a otimizações feitas pelo compilador. Chamamos as técnicas que utilizam informações obtidas de análises de divergências para melhorar código SIMD de *otimizações de divergências*. Apesar das análises de divergências serem novas, a literatura contém exemplos de otimizações de divergências. Nesses casos, trabalhos anteriores usaram instrumentação ou inspeção visual para identificar oportunidades de otimização. Dois exemplos de otimizações são *compartilhamento de variáveis* e *otimizações peephole*:

Compartilhamento de variáveis: consiste em guardar variáveis não divergentes na memória compartilhada. Voltando ao modelo μ-SIMD, uma vez que sabemos que uma variável v não é divergente, podemos removê-la da dos registradores de cada *thread* (memória local σ do elemento de processamento no μ-SIMD), e armazená-la na memória compartilhada (vetor compartilhado Σ no μ-SIMD) Algumas arquiteturas de GPU, como as da Nvidia e AMD, dividem um número fixo de registradores do multiprocessador (*stream multiprocessor*) entre todas as *threads*; criando a possibilidade de a demanda por registradores limitar o número de *threads* disponível, aumentando a chance do multiprocessador ficar ocioso por não ter o que fazer enquanto espera uma requisição de acesso à memória ser atendida. Essa otimização tem o efeito de reduzir a pressão de registradores em cada *thread*, permitindo a coexistências de mais *threads*. Collange *et al.* [Collange et al., 2009] propôs um mecanismo de hardware que detecta variáveis não divergentes e as migra para a memória compartilhada. Essa técnica identifica cerca de 19% dos valores nos registradores como não divergentes. Na Seção 5.3, mostramos que

nossa análise de divergências identifica estaticamente mais de 39% das variáveis como não divergentes.

Otimizações Peephole PTX, linguagem de montagem CUDA, possui duas versões para várias de suas instruções. Uma destas versões, que chamaremos *unificada*, considera que a instrução receberá os mesmos dados para todas as *threads*, enquanto a outra, mais genérica, lida com dados divergentes. A utilização de instruções unificadas sempre que possível tende a levar a códigos PTX mais eficientes.

No restante desta sessão descreveremos cada uma dessas otimizações em maiores detalhes.

Otimizações Peephole Otimizações *peephole* são uma categoria de melhorias de código que consiste em substituir pequenas sequências de instruções por outras, mais eficientes. A linguagem de montagem PTX possui várias instruções que admitem uma implementação mais eficiente quando todas as *threads* executam tais instruções com os mesmos dados, isto é, dados não divergentes. Um exemplo típico é a instrução de desvio condicional. A implementação de tais desvios, ao nível de hardware, é bastante complexa, afinal é preciso lidar com a divisão e a sincronização de *threads* divergentes. Entretanto, tal complexidade não deveria ser imposta sobre testes condicionais não divergentes. Por isto, o conjunto de instruções fornecido por PTX contém uma instrução `bra.uni`, a qual pode ser usada na ausência de divergências. O manual de programação PTX contém o seguinte texto sobre tal instrução [PTX, 2010]:

> "All control constructs are assumed to be divergent points unless the control-flow instruction is marked as uniform, using the `uni` suffix. For divergent control flow, the optimizing code generator automatically determines points of re-convergence. Therefore, a compiler or code author targeting PTX can ignore the issue of divergent threads, but has the opportunity to improve performance by marking branch points as uniform when the compiler or author can guarantee that the branch point is non-divergent."

Assim como desvios condicionais, existem outras instruções que podem receber o prefixo `uni`. Existem também instruções que não possuem análogo não divergente:

ldu uma instrução de carregamento uniforme, que assume que o endereço de origem dos dados é não divergente, isto é, todas as *threads* lerão o mesmo endereço.

call.uni uma chamada de função não divergente. Chamadas de função podem ser acrescidas de um predicado, de forma tal que a função será chamada somente

para aquelas *threads* cujo predicado seja verdadeiro. Esta instrução assume que tal predicado possui o mesmo valor para todas as *threads*.

ret.uni instrução de retorno. Um retorno divergente suspende as *threads* até que todas elas tenham retornado ao fluxo normal de execução. Desta forma, diferentes *threads* podem terminar uma função em momentos diferentes. Por outro lado, caso todas as *threads* terminem a função juntas, o retorno unificado pode ser usado para melhorar a eficiência do programa.

Compartilhamento de variáveis Quando a análise de divergências prova que uma variável não é divergente, então esta variável pode ser compartilhada entre todas as *threads*. Existem duas formas básicas de compartilhamento: interno e externo. O compartilhamento interno consiste na alocação de variáveis na área de memória compartilhada da GPU. O compartilhamento externo é uma alteração mais substancial do programa fonte, e consiste na migração de trabalho realizado na GPU para a CPU. A principal vantagem do compartilhamento de dados é a possível diminuição da pressão de registradores no *kernel* otimizado, o que tem o efeito benéfico de aumentar a quantidade de *threads* que podem usar o hardware gráfico simultaneamente. A GPU possui uma quantidade fixa destas unidades de armazenamento; por exemplo 8,192 registradores em uma placa GTX 8800. Para que a placa alcance a ocupação máxima, estes registradores devem ser distribuídos entre o teto de 768 *threads* que podem existir ao mesmo tempo. Assim, uma aplicação que requer mais de 10 registradores por *thread* não será capaz de usar todas as *threads* possíveis.

Iniciaremos esta discussão explicando o compartilhamento interno de dados, o que faremos via o exemplo da Figura 6.1. Nesta seção utilizaremos exemplos que, como este da Figura 6.1, embora bastante artificiais, têm a vantagem de ilustrar em poucas linhas nossas ideias. O *kernel* em questão preenche as células de um vetor `Out` com valores calculados a partir das colunas de uma matriz `In` e mais um conjunto de quatro variáveis. Este *kernel* usa 11 registrados, quando compilado via `nvcc -O3` e, desta forma, utiliza apenas 2/3 das 768 *threads* disponíveis.

A análise de divergências revela que as variáveis `a`, `b`, `c` e `d` possuem sempre os mesmos valores para todas as *threads* ativas. Logo, algumas destas variáveis podem ser mantidas em memória compartilhada, conforme mostra a Figura 6.2. Escolhemos mapear duas destas variáveis, `c` e `d`, para as variáveis compartilhadas `common0` e `common1`. Note que evitamos condições de corrida fazendo com que somente a *thread* zero escreva sobre estas variáveis. Como todas as *threads* escreveriam sempre o mesmo valor na área de memória compartilhada, o acesso exclusivo não é necessário para a correção do pro-

```
__global__ void nonShared1(float* In, float* Out, int Width) {
 int tid = blockIdx.x * blockDim.x + threadIdx.x;
 if (tid < Width) {
   Out[tid] = 0.0;
   float a = 2.0F, b = 3.0F, c = 5.0F, d = 7.0F;
   for (int k = tid; k < Width * Width; k += Width) {
     Out[tid] += In[k] / (a - b);
     Out[tid] -= In[k] / (c - d);
     float aux = a;
     a = b;
     b = c;
     c = d;
     d = aux;
   }
 }
}
```

Figura 6.1: Um *kernel* CUDA em que a pressão de registradores é 11, levando à um terço da ocupação de *threads* ativas.

grama; porém, tal controle diminui também a utilização do barramento de memória, aumentando a eficiência deste *kernel*.

Conforme observamos no programa da Figura 6.2, a *thread* 0 é responsável por realizar as computações que alteram a área de memória compartilhada. Existem situações, contudo, em que é possível migrar tais computações, totalmente ou em parte, para a CPU. Tal não é possível na Figura 6.2, pois os dados compartilhados são usados em estágios intermediários do cálculo de valores não compartilhados. Entretanto, o exemplo da Figura 6.3 ilustra uma situação diferente.

Na Figura 6.3, as variáveis `a`, `b`, `c` e `d` são não divergentes. Mais ainda, valores intermediários destas variáveis não contribuem em nada para o cálculo de variáveis divergentes. Pode-se, portanto, extrair a fatia do programa responsável pelo cálculo destes valores não divergentes, de modo que eles possam ser calculados pela CPU, conforme podemos ver na Figura 6.4. A vantagem, neste caso, é que a CPU pode realizar estas computações mais rapidamente que uma única *thread* da GPU, como foi feito pela *thread* 0 na Figura 6.2. O exemplo final, mostrado o código da GPU bem como o código da CPU, é mostrado na Figura 6.4.

Nós não implementamos compartilhamento de variáveis ou otimizações Peephole contudo, descreveremos a otimização *unificação de caminhos divergentes*, na Seção 6.1.

```
__global__ void regPress2(float* In, float* Out, int Width) {
 int tid = blockIdx.x * blockDim.x + threadIdx.x;
 if (tid < Width) {
   __shared__ float common0, common1;
   float c = 5.0F;
   float d = 7.0F;
   if (threadIdx.x == 0) {
     common0 = 2.0F;
     common1 = 3.0F;
   }
   __syncthreads();
   Out[tid] = 0.0;
   for (int k = tid; k < Width * Width; k += Width) {
     Out[tid] += In[k] / (common0 - common1);
     Out[tid] -= In[k] / (c - d);
     float aux = common0;
     if (threadIdx.x == 0) {
       common0 = common1;
       common1 = c;
     }
     __syncthreads();
     c = d;
     d = aux;
   }
 }
}
```

Figura 6.2: Uso de variáveis compartilhadas para diminuir a pressão de registradores. Este programa é cerca de 10% mais eficiente do que o *kernel* da Figura 6.1 em uma GPU 9400M.

6.1 Unificação de caminhos divergentes

Nesta seção apresentaremos nossa otimização, realizada pelo compilador, de unificação de caminhos divergentes. Ela é um tipo muito extensivo de *eliminação de redundâncias* [Morel & Renvoise, 1979], cujo propósito é extrair código comum de caminhos de execução divergentes.

6.1.1 A ideia geral

No exemplo da Figura 6.5, podemos ver que as sequências de instruções T e F (blocos básicos l_4 e l_{13}) possuem muitas operações (e alguns operandos) em comum. Representaremos uma instrução de leitura da memória como $\downarrow$ e uma instrução de escrita como $\uparrow$. Assim, podemos representar as duas sequências como $T = \{\perp, \downarrow, *, *, *, /, /, *,$

```
__global__ void nonShared2(float* In, float* Out, int Width) {
 int tid = blockIdx.x * blockDim.x + threadIdx.x;
 if (tid < Width) {
   Out[tid] = 0.0;
   float a = 2.0F, b = 3.0F, c = 5.0F, d = 7.0F;
   for (int k = 0; k < Width; k++) {
     int index = tid + (k * Width);
     Out[tid] += In[index] / k;
     Out[tid] -= In[index] / (k + 1);
     float aux = a;
     a = b;
     b = c;
     c = d;
     d = aux;
   }
   Out[tid] /= (a - b) * (c - d);
 }
}
```

Figura 6.3: Este *kernel*, cuja pressão de registradores é 11, possui computações que podem ser compartilhadas externamente à GPU.

$+, \uparrow\}$ e $F = \{\perp, \downarrow, *, *, /, *, *, +, \uparrow\}$. A questão que vem a mente é "posso fundir esses dois caminhos para compartilhar trabalho redundante entre *threads* divergentes?" Em caso afirmativo, podemos perguntar: "qual é a maior sequência de operações comuns entre essas duas sequências?" A Figura 6.5 (b) apresenta uma resposta possível. Dado esse alinhamento, finalmente podemos perguntar "como fundir esses dois caminhos, produzindo um programa novo equivalente ao original?" Nesse caso, uma solução possível é apresentada na Figura 6.5 (c). Note que adicionamos seletores ternários (`sel`) em μ-SIMD para escolher os a origem dos operandos das instruções compartilhadas e isso segue a semântica de C/C++/Java. Uma alternativa para arquiteturas que não oferecem instruções de seleção é a transformação clássica *if-conversion* [Kennedy & McKinley, 1990, Shin, 2007]. No resto desta seção descreveremos os algoritmos para responder as três questões anteriores:

1. Na Seção 6.1.2 explicaremos quais desvios podem ser fundidos para diminuir os caminhos divergentes.

2. Na Seção 6.1.3 apresentaremos como encontrar, dadas duas sequências de instruções disjuntas, as sequências de operações redundantes maiores (ou as mais vantajosas).

```
__global__ void sharedExternally(float* In, float* Out, int Width,
float alpha) {
 int tid = blockIdx.x * blockDim.x + threadIdx.x;
 if (tid < Width) {
   Out[tid] = 0.0;
   for (int k = 0; k < Width; k++) {
     int index = tid + (k * Width);
     Out[tid] += In[index] / k;
     Out[tid] -= In[index] / (k + 1);
   }
   Out[tid] /= alpha;
 }
}
// Parte do programa executada na CPU:
float a = 2.0F, b = 3.0F, c = 5.0F, d = 7.0F;
for (int k = 0; k < matrix_side_size; k++) {
 float aux = a;
 a = b;
 b = c;
 c = d;
 d = aux;
}
float alpha = (a - b) * (c - d);
sharedExternally<<< g, b >>>(In, Out, Width, alpha);
```

Figura 6.4: Compartilhamento externo de valores não divergentes.

3. Na Seção 6.1.4 descreveremos como transformar os caminhos divergentes em uma sequência de instruções onde as operações redundantes estão em um parte unificada.

6.1.2 Caminhos unificáveis

Somente faz sentido unificar os caminhos após desvios em que ambos os caminhos possuam código que não será executado pelas *threads* que tomaram o outro caminho. Chamamos esse desvios de "if-then-else", para distinguir de desvios "if-then" e "while". Por exemplo, a instrução `branch`(p_1, l_4) da Figura 6.6 é um desvio "if-then": se houver uma divergência, algumas *threads* irão executar o bloco básico l_3; contudo as outras não terão um trecho de código exclusivo para processar, elas apenas ficarão esperando as outras em l_4. Nesse caso, nenhuma *thread* realiza trabalho redundante, algumas apenas ficam ociosas durante um tempo. Ainda na Figura 6.6, o desvio possivelmente divergente `branch`(p_0, l_5) é um desvio "while". *Threads* que saírem do laço terão que esperar pelas que ainda têm iterações a realizar. Finalmente, `branch`(p_2, l_7) é um desvio

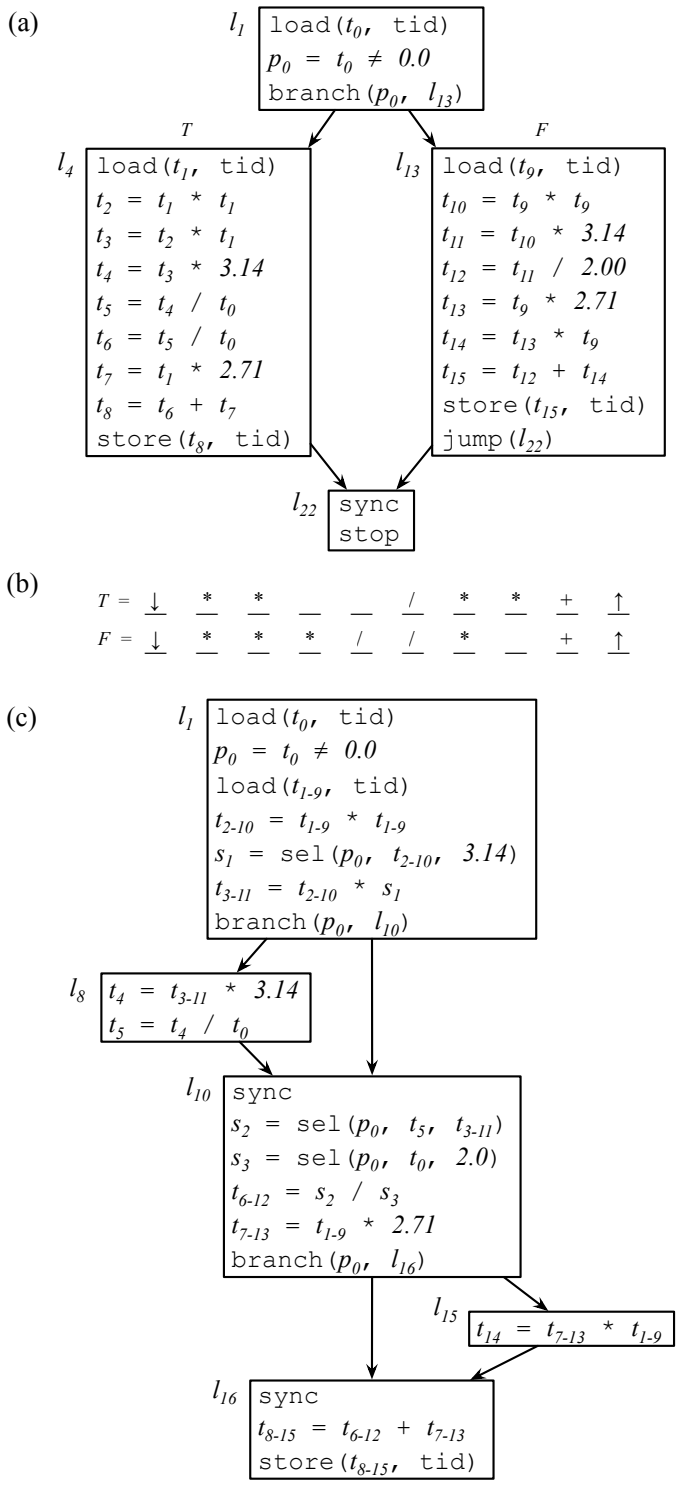

Figura 6.5: (a) Exemplo usado para explicar unificação de caminhos divergentes. (b) Possível alinhamento de instruções. (c) Código após a unificação.

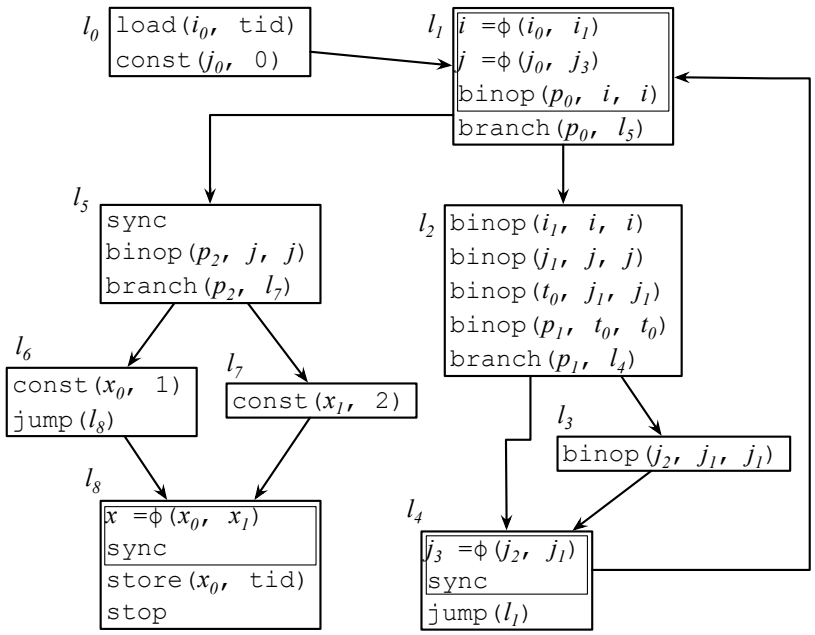

Figura 6.6: Código com vários tipos de desvio.

"if-then-else". Em face de uma divergência – que realmente pode acontecer segundo a análise do Capítulo 5 – algumas *threads* irão executar o rótulo l_6, enquanto outras vão para o rótulo l_7. Nesse caso, cada grupo de *threads* divergentes passa por um caminho exclusivo. Dizemos que os desvios "if-then-else" são *unificáveis*, e os detectamos da seguinte forma:

Definition 6.1.1 DESVIO UNIFICÁVEL *Seja* $l = \texttt{branch}(v, l_1)$ *um desvio condicional com dois rótulos sucessores* l_1 *e* l_2*. Esse desvio é unificável, se e somente se não existe caminho* $l_1 \xrightarrow{*} l_2 \in IR(l)$*, e não existe caminho* $l_2 \xrightarrow{*} l_1 \in IR(l)$*.*

6.1.3 Encontrando o melhor alinhamento de instruções

A efetividade da unificação de caminhos divergentes depende da solução do *Problema de Alinhamento de Instruções*, uma variação do alinhamento de sequências, que definimos como segue:

Definition 6.1.2 ALINHAMENTO DE INSTRUÇÕES BI-DIMENSIONAL ***Definições:*** *dados os seguintes parâmetros de entrada:*

- *dois vetores de instruções,* $T = \{i_1, \ldots, i_n\}$ *e* $F = \{j_1, \ldots, j_m\}$*;*

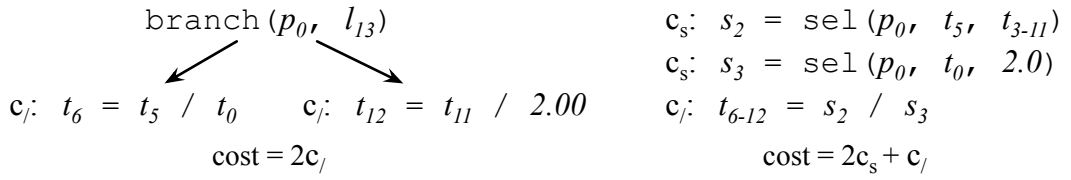

Figura 6.7: Comparando os custos de divergência e unificação.

- *uma função de pontuação 2-aria s, tal que $s(i, j)$ é o ganho de unificar as instruções i e j;*
- *uma função de pontuação 2-aria c, tal que $c(i, j)$ é o custo dos seletores necessários para unificar i e j;*
- *b, a penalidade de intervalo (*gap penalty*), que, no nosso caso, é o custo de inserir um desvio no código.*

***Problema**: encontrar uma sequência ordenada de pares $A = \langle (x_1, y_1), ..., (x_k, y_k) \rangle$, tais que:*

- *se $(x, y) \in A$, então $1 \leq x \leq n, 1 \leq y \leq m$*
- *se $r > s$ então $x_r \geq x_s$*
- *se $r > s$ então $y_r \geq y_s$*
- *$\sum(s(x, y) - c(x, y)) - b \times G$ is máxima, onde $(x, y) \in A$, e G é o número de intervalos no alinhamento.*

Resolvemos o problema de alinhamento de instruções bi-dimensional através do algoritmo clássico de alinhamento de sequências Smith-Waterman [Smith & Waterman, 1981]. Este algoritmo tem duas etapas: primeiro, ele constrói uma *matriz de rentabilidade* que atribui ganhos para cada pareamento de instruções possível. Em seguida, ele percorre essa matriz de trás para frente, a fim de descobrir o melhor alinhamento de instruções geral.

6.1.3.1 Computando a Matriz de Rentabilidade

Existe um custo, medido em termos de ciclos, envolvido na extração de um par de instruções de um caminho divergente. Figura 6.7 ilustra as operações necessárias para

unificar duas operações de divisão localizados em caminhos diferentes da Figura 6.5 (a). À esquerda temos o desvio original e à direita o programa após a unificação do desvio. O custo que acabamos de mencionar inclui selecionar os parâmetros da nova instrução ($2c_s$) e a execução da divisão em si ($c_/$). Dizemos que a fusão é *rentável* se o custo da execução da instrução unificada é inferior ao custo de executar o código divergente. Denotamos essa rentabilidade por uma *função de pontuação* s. No exemplo da Figura 6.7, temos $s = 2c_/ - 2c_s - c_/ = c_/ - 2c_s$. Observe que, se as duas variáveis passadas para o seletor são as mesmas, então o seletor não é necessário, e seu custo não é levado em consideração.

Para unificar instruções divergentes, temos de encontrar a sequência mais rentável de instruções a serem unificadas. Nosso guia nesta busca é a *matriz de rentabilidade*. Para produzir a rentabilidade da matriz, deve-se armazenar uma sequência de instruções ao longo do linha superior e outra sequência ao longo da coluna à esquerda de uma matriz bi-dimensional. Cada célula da matriz está associada a um valor g, ex.: $H[i, j] = g$, onde g é o maior ganho de qualquer modo possível de unificar as instruções até os índices i e j. O ganho da posição $H[i, j]$ é calculado pela seguinte relação de recorrência, onde os parâmetros s, c e b são explicados na Definição 6.1.2:

$$H[i,j] = s(i,j) + \text{MAX} \begin{cases} H[i-1, j-1] - c(i,j) - b, \\ H[i, j-1], \\ H[i-1, j], \\ 0 \end{cases} \tag{6.1}$$

Continuando com nosso exemplo, a Figura 6.8 mostra a matriz de rentabilidade que nós construímos para o programa na Figura 6.5 (a). Nós estamos usando a seguinte função de pontuação: $s(\downarrow, \downarrow) = s(\uparrow, \uparrow) = 100$, $s(*, *) = 2$, $s(/, /) = 8$, $s(+, +) = 2$. Nós assumimos que o custo de inserir um desvio é de $b = 2$. Observe que esse custo é pago apenas uma vez, quando saímos de uma sequência de movimentos diagonais fazermos um movimento horizontal ou vertical. Por razões de simplicidade, consideramos $c = 0$. Normalmente, obtemos esses números do manual de programação CUDA [CUDA, 2011]. Ou seja, $s(\iota, \iota)$ é o número de ciclos que o hardware leva para processar a instrução ι. O custo fixo b é o número de ciclos para executar uma instrução de desvio condicional (tomando o desvio ou não). O custo variável $c(i, j)$ é o número de ciclos para executar um seletor, multiplicado pelo número de seletores necessários para fundir instruções i e j.

Uma vez que nós calculamos a matriz de rentabilidade, para encontrar a solução

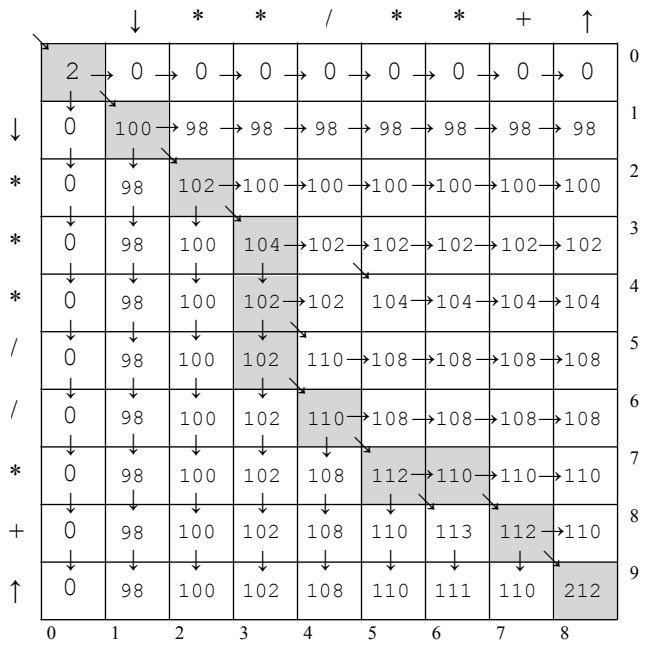

Figura 6.8: Matriz de rentabilidade do programa na Figura 6.5 (a).

do problema de alinhamento de instruções basta percorrer a matriz de trás para frente, a partir da última posição ($H[n, m]$) e indo para trás até a primeira ($H[0, 0]$), seguindo a direção do movimento mais lucrativo que chega até aquela posição. Ou seja, se o melhor modo de chegar à $H[i, j]$ for pela diagonal a partir de $H[i-1, j-1]$, então continuamos a nossa travessia indo para $H[i-1, j-1]$. Para fazer o caminho de volta, armazenamos em cada célula da matriz, a direção (horizonte, vertical ou diagonal) utilizada para chegar a ela. No nosso exemplo, a última posição é $H[9, 8]$, e a sequência mais rentável é $A = \langle(1,1),(2,2),(3,3),(4,3),(5,3),(6,4),(7,5),(7,6),(8,7),(9,8)\rangle$.

No exemplo da Figura 6.8, cada célula da matriz tem a direção do movimento mais rentável para chegar a ela. As caixas cinzas marcam o caminho mais rentável na matriz, o que denota exatamente o alinhamento visto na Seção 6.1.1. Damos preferência a movimentos horizontais e verticais sobre movimentos diagonais. Ou seja, quando o ganho de unificar duas instruções é o mesmo que o ganho de mantê-las divergentes, escolhemos o último pois ele já pagou o custo da inserção uma instrução de desvio. Desta forma, evitamos o custo adicional da inserção de um desvio no código se, mais tarde, encontramos duas instruções que não podem ser alinhadas. Por exemplo, na Figura 6.8, poderíamos ter chegado em $H[4, 3]$ usado uma diagonal a partir $H[3, 2]$, unificando duas multiplicações. No entanto, as próximas duas instruções – multiplicação e divisão – não são compatíveis, tendo uma pontuação muito baixa. Assim,

seríamos obrigados a inserir um desvio no código. Por outro lado, dado que chegamos a $H[4,3]$ verticalmente, o custo do desvio já havia sido pago, portanto, não pagamos nada para chegar à $H[5,3]$ verticalmente.

De forma similar aos algoritmos tradicionais de eliminação de redundância, abusamos do poder de identidades entre as instruções para maximizar a quantidade de código extraída dos desvios. Assim, passamos a unificar instruções com *opcodes* diferentes, caso exista uma identidade entre esses operandos. Por exemplo, o operador de comparação: $p = a > b$ é equivalente a $p = b < a$. Como outro exemplo, na Figura 6.8, célula $H[8,6]$ deixamos o lucro de unificar uma adição com uma multiplicação ser 1. Neste caso, assumimos a existência de uma instrução de "multiplicação-adição": $t = a \times b + c$, que pode ser usada como uma identidade para ambas as instruções. Em nosso trabalho usamos apenas identidades simples, no entanto, é possível levar esta abordagem ao extremo, empregando a técnica de saturação de igualdades [Tate et al., 2009] para encontrar as sequências de código mais semelhantes para cada caminho do desvio.

Complexidade Algorítmica Dadas duas sequências de instruções, T e F, o cálculo da matriz de rentabilidade é de $O(|T| \times |F|)$, em termos de tempo e espaço. Percorrer a matriz da última célula até à origem é $O(|T| + |F|)$.

6.1.4 Geração de Código

Dadas duas sequências de instruções $T = \{i_1, \ldots, i_n\}$ e $F = \{j_1, \ldots, j_m\}$ após um desvio condicional `branch`(p, l'), mais um alinhamento de instruções $A = \{(x_1, y_1), ..., (x_k, y_k)\}$, são utilizadas as seguintes regras para a geração de código:

- $A = \{\ldots, (x-1, y-1), (x, y), \ldots\}$: unifica-se as instruções $T[x]$ e $F[y]$ em uma nova instrução no fim do bloco de instruções unificado;
- $A = \{\ldots, (x-1, y-1), (x, y), (x, y+1), \ldots, (x, y+k), \ldots\}$: cria-se uma sequência de rótulos $l_1, \ldots, l_k$ para as instruções $F[y+1], \ldots, F[y+k]$, e adiciona-se `branch`(p, l) ao fim do bloco de instruções unificado, onde l é o rótulo do próximo bloco unificado;
- $A = \{\ldots, (x-1, y-1), (x, y), (x+1, y), \ldots, (x+k, y), \ldots\}$: cria-se uma sequência de rótulos $l_1, \ldots, l_k$ para as instruções $T[x+1], \ldots, T[x+k]$, e adiciona-se `branch`(p, l) ao fim do bloco de instruções unificado.

A Figura 6.5 (c) apresenta o código que é produzido seguindo-se a matriz de rentabilidade na Figura 6.8.

6.2 Experimentos

Nesta seção, discutiremos os experimentos que usamos para testar e validar a otimização. A otimização foi implementada sobre o compilador de PTX Ocelot [Kerr et al., 2009], versão 1.0.432. PTX [PTX, 2010], é um conjunto de instruções de alto nível, utilizado para representar código de máquina das GPUs da Nvidia. Ocelot é uma ferramenta de código aberto, projetada e implementada na Universidade Georgia Tech, que analisa e otimiza PTX. Os experimentos foram executados em uma GPU NVIDIA GeForce GTX 470.

As aplicações: Não há um consenso sobre o conjunto padrão de aplicações de teste CUDA. Assim, procuramos reunir um conjunto significativo de aplicações que (i) estão disponíveis publicamente, (ii) têm sido utilizados em trabalhos anteriores [Cederman & Tsigas, 2009, Che et al., 2009, Kerr et al., 2009] e (iii) foi concebido e implementado por especialistas na área. Nós escolhemos as seguintes coleções de *benchmarks*: Rodínia [Che et al., 2009], o SDK da Nvidia [SDK, 2011] e o GPU-quicksort [Cederman & Tsigas, 2009]. Estes *benchmarks* contém 30 aplicações, que fornecem mais de 80 *kernels*. Nós encontramos, através de instrumentação, divergências em 67 núcleos, que vamos usar nesta seção. Esses programas, compilados com o nvcc 3.0 da Nvidia, contêm juntos mais de 46.000 instruções PTX.

Oportunidades para unificação de caminhos: Os três gráficos na Figura 6.9 mostram números estáticos que medimos após a aplicação de fusão de caminhos em nossos testes. Cada ponto no eixo X é um *kernel* e eles estão ordenados pelo número de desvios unificáveis. No topo apresentamos o número de desvios por *kernel* (742 no total). Na segunda parte temos o número de desvios unificáveis (196 no total), sendo que o *kernel* com mais desvios unificáveis possui 31 deles. Na terceira parte, está o número de desvios unificáveis que são divergentes e, finalmente, na porte do fundo temos o número de unificações lucrativas. Encontramos 196 desvios unificáveis nas aplicações, daí, cerca de 26 % do ramos são da variedade "if-then-else" que descrevemos no Seção 6.1.2. Quanto à proporção de desvios unificáveis, essa coleção não difere de programas em C tradicionais. Por exemplo, `403.gcc`, o maior programa do SPEC CPU 2006, contém 2.152.106 desvios, dos quais 23,5% são unificáveis. Nós obtivemos esse número, analisando os *bytecodes* que o LLVM 2.8 [Lattner & Adve, 2004] produz após compilar `403.gcc`. A análise de divergências indica que 70% dos desvios unificáveis, ou seja, 137 desvios condicionais, são divergentes. Depois de construir a matriz de rentabilidade de todos estes desvios, descobrimos 30 unificações rentáveis. Em outras palavras, o

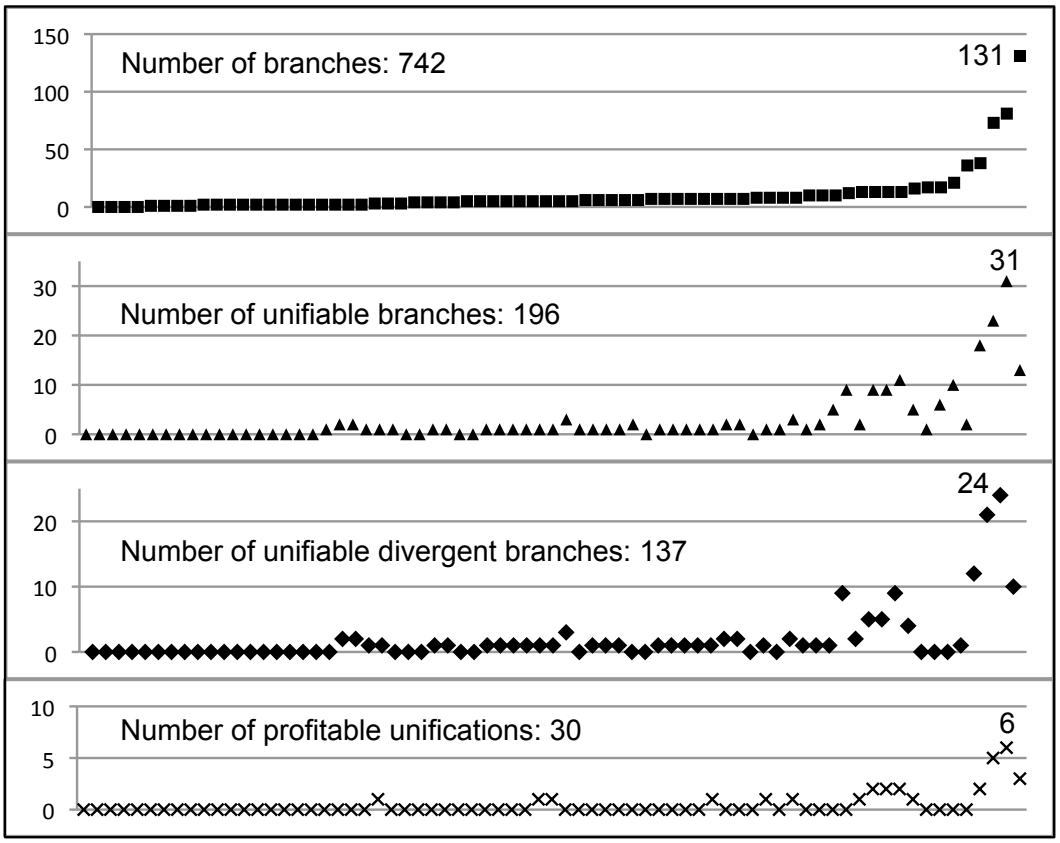

Figura 6.9: Números da unificação de caminhos. *Kernels* no eixo X estão ordenados pelo número de desvios.

algoritmo da Seção 6.1.3 aponta que 4% dos 742 desvios poderiam ser unificados.

Ganhos de desempenho da unificação de caminhos: Escolhemos seis desvios, dos 30, cuja unificação resulta em ganho superior a 100 ciclos. Dois desses desvios estão em `rodinia:lud` (0.37%, -3.35%). Os outros estão no `sdk:mergeSort` (3.59%), `rodinia:heartwall` (0.12%), `rodinia:srad` (0.48%) e GPU-quicksort (3.09%). Os números entre parênteses são as acelerações após a fusão de caminhos. Observamos ganhos em cinco desvios, e uma perda de desempenho em `lud`, que foi devido a um aumento na pressão de registradores no bloco unificado. Mais registradores implica em menos *threads* em execução simultânea na GPU. Nossos ganhos são pequenos porque (i) os códigos são de aplicações bem conhecidas, que já estão altamente otimizadas e (ii) a implementação atual não consegue unificar os desvios no final dos blocos básicos, impedindo-a de unificar conjuntos de vários blocos básicos. A aceleração negativa em `rodinia:lud` é devido a uma divergência falsa: os dados de entrada disponíveis

para esta aplicação estão perfeitamente alinhados com o número de *threads* que ela cria, assim, nunca acontecerão divergências devido a uma condição de fronteira. Este tipo de adaptação manual de dados é um padrão recorrente em alguns dos *benchmarks* CUDA disponíveis publicamente. No entanto, entendemos que colocar o ônus da codificação para um tamanho específico de entrada sobre desenvolvedor é mais exceção do que caso comum nas aplicações para GPU.

6.3 Conclusão

Neste capítulo apresentamos uma técnica de otimização de código divergente para mitigar o impacto das divergências no desempenho de aplicações que executam em arquiteturas SIMD. Essa otimização é totalmente automática, podendo ser realizada durante a compilação, não necessitando de nenhum esforço por parte do programador da aplicação. Infelizmente, essa técnica obteve ganhos de desempenho baixos porque utilizamos códigos de aplicações que já estão altamente otimizadas e ela não consegue unificar instruções de desvio no final dos blocos básicos.

Capítulo 7

Conclusões

Nesta dissertação desenvolvemos técnicas de análise e otimização de código que buscam entender e melhorar automaticamente programas escritos para o modelo SIMD. Mostramos que este modelo de processamento apresenta fenômenos complexos o suficiente cuja análise demanda uma combinação de técnicas de geração e otimização de código estáticas e dinâmicas. Entre tais fenômenos, destaca-se aquele usualmente chamado divergência de execução. Do ponto de vista dinâmico, desenvolvemos um *profiler* de código capaz de mensurar o volume de divergências em código CUDA. Do lado estático, nós projetamos e implementamos uma análise que determina com alta probabilidade os desvios condicionais que irão causar divergências. Ainda do ponto de vista estático desenvolvemos uma otimização de código chamada *fusão de caminhos divergentes*, a qual junta, de forma automática, trechos de código que, doutro modo, seriam executados de forma divergente.

7.1 Análise Dinâmica: O Papel do *Profiler*

Profiling consiste em fazer várias medições de uma aplicação em execução. A medição pode ser feita por código inserido na aplicação ou por uma ferramenta externa. Essa técnica é útil para localizar trechos que precisam ser otimizados e para validar otimizações.

Quando se implementa um profiler, deve-se tomar cuidado para que o processo de medição não contamine os resultados. Bem como é desejável que ele tenha a maior precisão possível. No caso da nossa instrumentação o primeiro foi garantido porque não utilizamos nenhuma variável do programa (o que aumenta a pressão por registradores e o tempo de execução), para obter o segundo tivemos que fazer um laço para selecionar a *thread* escritora e assim garantir que todos desvios sejam contados.

O profiler têm a ventagem de analisar o programa enquanto ele executa, podendo entender fenômenos que dependem da entrada do programa e argumentos para funções, podendo fazer uma análise muito mais profunda sobre o que ocorre durante a execução.

7.2 Análise Estática de Caminhos Divergentes

Ao contrário do profiler, que retorna informação apenas sobre uma determinada execução, a análise estática consegue provar que algumas propriedades são válidas em todas execuções. Além disso não requer a execução da aplicação, permitindo que ela seja invocada automaticamente durante o processo de compilação, ao contrário das análises dinâmicas que necessitam que o programador invoque a aplicação e passe o arquivo de entrada. Infelizmente devido às complexidade de se implementar análises estáticas e devido ao fato que elas não têm informação do que ocorre durante a execução nem sobre os dados de entrada, geralmente as técnicas de análise estática só podem fazer verificações muito conservadoras.

Assim as análises dinâmicas e estática possuem visões complementares sobre o que acontece durante a execução do programa: análises dinâmicas possuem uma visão aprofundada sobre uma ou algumas execuções, enquanto as análises estáticas têm uma visão superficial, mas válida para todas execuções. A análise dinâmica pode ser usada para validar os resultados da análise estática, ou mesmo para guiá-la, informando quais partes do código realmente precisam ser otimizadas.

7.3 O Produto Final: Otimização Automática

A otimização obteve ganhos de desempenho baixos porque utilizamos códigos de aplicações bem conhecidas, que já estão altamente otimizadas e a otimização não consegue unificar as instruções de desvio no final dos blocos básicos, impedindo-a de unificar regiões de código maiores. Em alguns casos a otimização causou perda de desempenho devido a aumento na pressão de registradores no bloco unificado ou à otimização de falsos positivos da análise estática de divergências.

Acreditamos que futuramente surgirão aplicações para GPU mais complexas, às quais o programador não terá tempo de otimizar o código todo (no máximo os trechos mais executados). Nessas aplicações nossas otimizações terão um impacto maior.

7.4 Trabalhos Futuros

Este trabalho abre novas direções para a pesquisa em geração de código para aplicações de alto desempenho:

- Atualmente a instrumentação de divergências só consegue informar o número do bloco básico que contém a instrução onde ocorre o desvio divergente, como descrito na Seção 4.2. Podemos melhorá-la para que seja capaz de informar o número da linha no código fonte onde a divergência ocorreu, aproveitando as diretivas `.loc` que o compilador CUDA (NVCC) insere no código PTX [PTX, 2010].

- Melhorar a precisão da análise de divergências, acoplando-a com a técnica de detecção de vetores afins de Collange *et al.* [Collange et al., 2009]. A análise de variáveis divergentes do Capítulo 5 detecta o que Collange *et al.* chamam de vetores uniformes (o vetor é o conjunto de valores de uma variável para todas as *threads* de um *warp*): o valor da variável v na *thread* x é $v_x = a$, onde a é constante para todas as *threads*. Em um vetor afim, o valor da variável v na *thread* x é $v_x = ax+b$, onde a e b são constantes. Esperamos que com a utilização dessa técnica, a análise de divergências não produza mais falsos positivos para construções como abaixo, onde a análise indica que j possui uma dependência de sincronização em relação a i (ela aponta que, sendo i divergente, as *threads* poderiam executar números diferentes de iterações do laço). Com a detecção de vetores afins, a análise de divergências perceberia que todas as *threads* vão executar o mesmo número de iterações e assim não existe dependência de sincronização entre j e i.

```
// max é multiplo de ntid.x

for (i=tid.x; i<max; i+=ntid.x) {
    int j=0;
    ...
    j++;
}
```

- Tornar unificação de blocos básicos consciente do número de registradores que o código gerado está usando para evitar que o código gerado fique mais lento por usar registradores em excesso.

- Fazer a unificação de blocos básicos tratar as instruções de desvios no final dos blocos unificados, para torná-la capaz de unificar conjuntos de blocos básicos, como fizemos manualmente no algoritmo de ordenação bitônica usado na implementação de *quicksort* de Cederman *et al.* (Seção 4.3).

- Implementar outras otimizações descritas no Capítulo 6, como compartilhamento de variáveis.

Como visto, exitem várias oportunidades de pesquisa, principalmente no aumento do ganho da unificação de blocos básicos. Além disso, a análise estática de divergências pode ter outas aplicações além da otimização de código divergente.

Referências Bibliográficas

[Abel et al., 1969] Abel, N. E.; Budnik, P. P.; Kuck, D. J.; Muraoka, Y.; Northcote, R. S. & Wilhelmson, R. B. (1969). TRANQUIL: a language for an array processing computer. Em *AFIPS 1969 Spring Joint Computer Conference*, volume 34 of *AFIPS Conference Proceedings*, pp. 57--73, Montvale, NJ. AFIPS Press.

[Acharya et al., 1998] Acharya, A.; Uysal, M. & Saltz, J. (1998). Active disks: programming model, algorithms and evaluation. Em *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pp. 81--91, New York, NY, USA. ACM Press.

[Agarwal et al., 1986] Agarwal, A.; Sites, R. L. & Horowitz, M. (1986). ATUM: a new technique for capturing address traces using microcode. Em *ISCA '86: Proceedings of the 13th annual International Symposium on Computer Architecture*, pp. 119--127, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Agarwal et al., 2007] Agarwal, S.; Barik, R.; Sarkar, V. & Shyamasundar, R. K. (2007). May-happen-in-parallel analysis of X10 programs. Em *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pp. 183--193, New York, NY, USA. ACM.

[Aguilera et al., 2003] Aguilera, M. K.; Mogul, J. C.; Wiener, J. L.; Reynolds, P. & Muthitacharoen, A. (2003). Performance debugging for distributed systems of black boxes. Em *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 74--89, New York, NY, USA. ACM.

[Aho et al., 2008] Aho, A. V.; Lam, M. S.; Sethi, R. & Ullman, J. D. (2008). *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley.

[Appel & Palsberg, 2003] Appel, A. W. & Palsberg, J. (2003). *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA. Second edition.

[Apple, 2009] Apple (2009). `http://developer.apple.com/tools/sharkoptimize.html`.

[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. (2006). The Landscape of Parallel Computing Research: A View from Berkeley. Relatório técnico UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[Baghsorkhi et al., 2010] Baghsorkhi, S. S.; Delahaye, M.; Patel, S. J.; Gropp, W. D. & Hwu, W.-m. W. (2010). An adaptive performance modeling tool for GPU architectures. Em Govindarajan, R.; Padua, D. A. & Hall, M. W., editores, *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pp. 105--114, New York, NY, USA. ACM.

[Bakhoda et al., 2009] Bakhoda, A.; Yuan, G. L.; Fung, W. W. L.; Wong, H. & Aamodt, T. M. (2009). Analyzing CUDA workloads using a detailed GPU simulator. Em *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*, pp. 163--174, Los Alamitos, CA, EUA. IEEE Computer Society Press.

[Ball & Larus, 1993] Ball, T. & Larus, J. R. (1993). Branch Prediction For Free. Em *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming Language Design and Implementation*, pp. 300--313, New York, NY, USA. ACM.

[Barnes et al., 1968] Barnes, G. H.; Brown, R. M.; Kato, M.; Kuck, D. J.; Slotnick, D. L. & Stokes, R. A. (1968). The ILLIAC IV Computer. *IEEE Transactions on Computers*, 17(8):746--757.

[Batcher, 1968] Batcher, K. E. (1968). Sorting Networks and Their Applications. Em *AFIPS 1968 Spring Joint Computer Conference*, volume 32 of *AFIPS Conference Proceedings*, pp. 307--314. Thomson Book Company, Washington D.C.

[Bienia et al., 2008] Bienia, C.; Kumar, S.; Singh, J. P. & Li, K. (2008). The PARSEC benchmark suite: characterization and architectural implications. Em *Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques*, PACT '08, pp. 72--81, New York, NY, USA. ACM.

[Bolz et al., 2003] Bolz, J.; Farmer, I.; Grinspun, E. & Schröoder, P. (2003). Sparse matrix solvers on the GPU: conjugate gradients and multigrid. Em *Proceedings*

of the 30th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '03, pp. 917--924, New York, NY, USA. ACM.

[Bougé & Levaire, 1992] Bougé, L. & Levaire, J.-L. (1992). Control structures for data-parallel SIMD languages: semantics and implementation. *Future Generation Computer Systems*, 8(4):363--378.

[Bouknight et al., 1972] Bouknight, W.; Denenberg, S.; McIntyre, D.; Randall, J.; Sameh, A. & Slotnick, D. (1972). The Illiac IV system. *Proceedings of the IEEE*, 60(4):369--388.

[Boyer et al., 2008] Boyer, M.; Skadron, K. & Weimer, W. (2008). Automated Dynamic Analysis of CUDA Programs. Em *Third Workshop on Software Tools for MultiCore Systems - STMCS 2008*.

[Brockmann & Wanka, 1997] Brockmann, K. & Wanka, R. (1997). Efficient Oblivious Parallel Sorting on the MasPar MP-1. Em *Proceedings of the 30th Hawaii International Conference on System Sciences: Software Technology and Architecture*, volume 1 of *HICSS '97*, pp. 200--208, Washington, DC, USA. IEEE Computer Society.

[Brodtkorb et al., 2010] Brodtkorb, A. R.; Dyken, C.; Hagen, T. R.; Hjelmervik, J. M. & Storaasli, O. O. (2010). State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1--33.

[Brown, 2006] Brown, S. (2006). DXT Compression Techniques. `http://www.sjbrown.co.uk/?article=dxt`.

[Buck et al., 2004] Buck, I.; Foley, T.; Horn, D.; Sugerman, J.; Fatahalian, K.; Houston, M. & Hanrahan, P. (2004). Brook for GPUs: stream computing on graphics hardware. Em *Proceedings of the 31th ACM SIGGRAPH Conference and Exhibition 2004*, SIGGRAPH '04, pp. 777--786, New York, NY, USA. ACM.

[Cantrill et al., 2004] Cantrill, B. M.; Shapiro, M. W. & Leventhal, A. H. (2004). Dynamic instrumentation of production systems. Em *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pp. 15--28, Berkeley, CA, USA. USENIX Association.

[Castaño, 2007] Castaño, I. (2007). High Quality DXT Compression using CUDA. Relatório técnico, NVIDIA Corporation, Santa Clara, CA, Estados Unidos. `http://developer.nvidia.com/cuda-cc-sdk-code-samples#dxtc`.

[Cederman & Tsigas, 2009] Cederman, D. & Tsigas, P. (2009). GPU-Quicksort: A practical Quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics*, 14:1.4--1.24.

[Chang et al., 1991] Chang, P. P.; Mahlke, S. A. & Hwu, W.-m. W. (1991). Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301--1321.

[Che et al., 2009] Che, S.; Boyer, M.; Meng, J.; Tarjan, D.; Sheaffer, J. W.; Lee, S.-H. & Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. Em *IISWC '09: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44--54, Washington, DC, USA. IEEE Computer Society.

[Cherem et al., 2008] Cherem, S.; Chilimbi, T. & Gulwani, S. (2008). Inferring locks for atomic sections. Em *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 304--315, New York, NY, USA. ACM.

[Choi et al., 2002] Choi, J.-D.; Lee, K.; Loginov, A.; O'Callahan, R.; Sarkar, V. & Sridharan, M. (2002). Efficient and precise datarace detection for multithreaded object-oriented programs. *ACM SIGPLAN Notices*, 37(5):258--269.

[ClearSpeed, 2011] ClearSpeed (2011). CSX700 Floating Point Processor Datasheet. Relatório técnico 06-PD-1425 Rev 1E, ClearSpeed Technology Ltd, Witney, Oxfordshire, Inglaterra.

[Collange et al., 2009] Collange, S.; Defour, D. & Zhang, Y. (2009). Dynamic detection of uniform and affine vectors in GPGPU computations. Em *Proceedings of the 2009 international conference on Parallel processing*, Euro-Par'09, pp. 46--55, Berlin, Heidelberg. Springer-Verlag.

[Corrigan et al., 2009] Corrigan, A.; Camelli, F.; Löhner, R. & Wallin, J. (2009). Running unstructured grid cfd solvers on modern graphics hardware. Em *19th AIAA Computational Fluid Dynamics Conference*, number AIAA 2009-4001, pp. 1--11.

[Coutinho et al., 2010] Coutinho, B.; Sampaio, D.; Pereira, F. M. Q. & Meira Jr., W. (2010). Performance Debugging of GPGPU Applications with the Divergence Map. Em *Proceedings of the 2010 22nd International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '10, pp. 33--40, Washington, DC, USA. IEEE Computer Society.

[Coutinho et al., 2011] Coutinho, B.; Sampaio, D.; Pereira, F. M. Q. & Meira Jr., W. (2011). Divergence Analysis and Optimizations. Em *Proceedings of the 20th international conference on Parallel Architectures and Compilation Techniques*, PACT '11, New York, NY, USA. ACM.

[Coutinho et al., 2012] Coutinho, B.; Sampaio, D.; Pereira, F. M. Q. & Meira Jr., W. (2012). Profiling divergences in GPU applications. *Concurrency and Computation: Practice & Experience*.

[Crovella & LeBlanc, 1993] Crovella, M. E. & LeBlanc, T. J. (1993). Performance debugging using parallel performance predicates. *ACM SIGPLAN Notices*, 28(12):140--150.

[CUDA, 2011] CUDA (2011). *NVIDIA CUDA Programming Guide*. NVIDIA Corporation. Version 3.2.

[Cytron et al., 1991] Cytron, R.; Ferrante, J.; Rosen, B. K.; Wegman, M. N. & Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451--490.

[Dalheimer & Welsh, 2005] Dalheimer, M. K. & Welsh, M. (2005). *Running Linux*. O'Reilly Media, Inc.

[Davis & Hennessy, 1988] Davis, H. & Hennessy, J. (1988). Characterizing the synchronization behavior of parallel programs. Em *PPoPP '88: Proceedings of the 1st ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pp. 198--211, New York, NY, USA. ACM.

[Evans & Larochelle, 2002] Evans, D. & Larochelle, D. (2002). Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42--51.

[Eyerman & Eeckhout, 2009] Eyerman, S. & Eeckhout, L. (2009). Per-thread cycle accounting in SMT processors. Em *ASPLOS '09: Proceeding of the 14th international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pp. 133--144, New York, NY, USA. ACM.

[Fan et al., 2004] Fan, Z.; Qiu, F.; Kaufman, A. & Yoakum-Stover, S. (2004). GPU Cluster for High Performance Computing. Em *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, Washington, DC, USA. IEEE Computer Society.

[Farrell & Kieronska, 1996] Farrell, C. A. & Kieronska, D. H. (1996). Formal specification of parallel SIMD execution. *Theoretical Computer Science*, 169(1):39--65.

[Flynn, 1972] Flynn, M. J. (1972). Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948--960.

[Foster, 2002] Foster, J. (2002). *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. Tese de doutorado, University of California, Berkeley.

[Fung et al., 2007] Fung, W. W. L.; Sham, I.; Yuan, G. & Aamodt, T. M. (2007). Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. Em *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pp. 407--420, Washington, DC, USA. IEEE Computer Society.

[Graham et al., 1982a] Graham, S. L.; Kessler, P. B. & Mckusick, M. K. (1982a). Gprof: A call graph execution profiler. Em *PLDI '82: Proceedings of the ACM SIGPLAN 1982 conference on Programming Language Design and Implementation*, pp. 120--126, New York, NY, USA. ACM.

[Graham et al., 1982b] Graham, S. L.; Kessler, P. B. & McKusick, M. K. (1982b). gprof: a call graph execution profiler (with retrospective). Em *Best of PLDI*, pp. 49--57.

[Hall et al., 2009] Hall, M.; Padua, D. & Pingali, K. (2009). Compiler research: the next 50 years. *Communications of the ACM*, 52(2):60--67.

[Harish & Narayanan, 2007] Harish, P. & Narayanan, P. J. (2007). Accelerating large graph algorithms on the GPU using CUDA. Em *Proceedings of the 14th international conference on High Performance Computing*, HiPC'07, pp. 197--208, Berlin, Heidelberg. Springer-Verlag.

[Harris et al., 2002] Harris, M. J.; Coombe, G.; Scheuermann, T. & Lastra, A. (2002). Physically-based visual simulation on graphics hardware. Em *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pp. 109--118, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[Hollingsworth & Miller, 1993] Hollingsworth, J. K. & Miller, B. P. (1993). Dynamic control of performance monitoring on large scale parallel systems. Em *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pp. 185--194, New York, NY, USA. ACM.

[Hong & Kim, 2009] Hong, S. & Kim, H. (2009). An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. Em *ISCA '09: Proceedings of the 36th annual International Symposium on Computer Architecture*, pp. 152--163, New York, NY, USA. ACM.

[Hoogvorst et al., 1991] Hoogvorst, P.; Keryell, R.; Matherat, P. & Paris, N. (1991). POMP or How to Design a Massively Parallel Machine with Small Developments. Em *PARLE '91: Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*, pp. 83--100. Springer.

[Huang et al., 2006] Huang, W.; Ghosh, S.; Velusamy, S.; Sankaranarayanan, K.; Skadron, K. & Stan, M. R. (2006). HotSpot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design. *IEEE Transactions VLSI Systems*, 14(5):501--513.

[Huck & Malony, 2005] Huck, K. A. & Malony, A. D. (2005). PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing. Em *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, p. 41, Washington, DC, USA. IEEE Computer Society.

[Jain, 1991] Jain, R. (1991). *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. John Wiley & Sons, Inc.

[Ji et al., 1998] Ji, M.; Felten, E. W. & Li, K. (1998). Performance measurements for multithreaded programs. Em *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pp. 161--170, New York, NY, USA. ACM.

[Kennedy & McKinley, 1990] Kennedy, K. & McKinley, K. S. (1990). Loop distribution with arbitrary control flow. Em *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pp. 407--416, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Kerr et al., 2009] Kerr, A.; Diamos, G. F. & Yalamanchili, S. (2009). A characterization and analysis of PTX kernels. Em *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pp. 3--12, Washington, DC, USA. IEEE Computer Society.

[Khronos OpenCL Working Group, 2008] Khronos OpenCL Working Group (2008). *The OpenCL Specification, version 1.0.29.*

[Krinke, 1998] Krinke, J. (1998). Static slicing of threaded programs. *ACM SIGPLAN Notices*, 33(7):35--42.

[Krüger & Westermann, 2003] Krüger, J. & Westermann, R. (2003). Linear algebra operators for GPU implementation of numerical algorithms. Em *Proceedings of the 30th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '03, pp. 908--916, New York, NY, USA. ACM.

[Lam, 1988] Lam, M. (1988). Software pipelining: an effective scheduling technique for VLIW machines. Em *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*, pp. 318--328, New York, NY, USA. ACM.

[Lattner & Adve, 2004] Lattner, C. & Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Em *Proceedings of the international symposium on Code Generation and Optimization: feedback-directed and runtime optimization*, CGO '04, pp. 75--88, Washington, DC, USA. IEEE Computer Society.

[Lawrie et al., 1975] Lawrie, D. H.; Layman, T.; Baer, D. & Randal, J. M. (1975). Glypnir–a programming language for Illiac IV. *Communications of the ACM*, 18(3):157--164.

[Lee et al., 2010] Lee, V. W.; Kim, C.; Chhugani, J.; Deisher, M.; Kim, D.; Nguyen, A. D.; Satish, N.; Smelyanskiy, M.; Chennupaty, S.; Hammarlund, P.; Singhal, R. & Dubey, P. (2010). Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. Em *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pp. 451--460, New York, NY, USA. ACM.

[Lindholm et al., 2008] Lindholm, E.; Nickolls, J.; Oberman, S. & Montrym, J. (2008). NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28:39--55.

[Magnusson et al., 2002] Magnusson, P. S.; Christensson, M.; Eskilson, J.; Forsgren, D.; Hållberg, G.; Högberg, J.; Larsson, F.; Moestedt, A. & Werner, B. (2002). Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50--58.

[Mark et al., 2003] Mark, W. R.; Glanville, R. S.; Akeley, K. & Kilgard, M. J. (2003). Cg: a system for programming graphics hardware in a C-like language. Em *Proce-*

edings of the 30th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '03, pp. 896--907, New York, NY, USA. ACM.

[Masticola & Ryder, 1991] Masticola, S. P. & Ryder, B. G. (1991). A model of Ada programs for static deadlock detection in polynomial times. Em *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pp. 97--107. ACM.

[Matsumoto & Nishimura, 1998a] Matsumoto, M. & Nishimura, T. (1998a). Dynamic Creation of Pseudorandom Number Generators. Em *Proceedings of the Third International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, pp. 56--69.

[Matsumoto & Nishimura, 1998b] Matsumoto, M. & Nishimura, T. (1998b). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8:3--30.

[Meira, 1997] Meira, W. (1997). *Understanding parallel program performance using cause-effect analysis*. Tese de doutorado, University of Rochester, Rochester, NY, USA. Report Number: UR CSD / TR663.

[Mellor-Crummey et al., 2002] Mellor-Crummey, J.; Fowler, R. J.; Marin, G. & Tallent, N. (2002). HPCVIEW: A Tool for Top-down Analysis of Node Performance. *Journal of Supercomputing*, 23(1):81--104.

[Microsoft, 2011a] Microsoft (2011a). Pixel Shader Differences. `http://msdn.microsoft.com/en-us/library/bb219846.aspx`.

[Microsoft, 2011b] Microsoft (2011b). Vertex Shader Differences. `http://msdn.microsoft.com/en-us/library/bb172931.aspx`.

[Miller et al., 1995] Miller, B. P.; Callaghan, M. D.; Cargille, J. M.; Hollingsworth, J. K.; Irvin, R. B.; Karavanic, K. L.; Kunchithapadam, K. & Newhall, T. (1995). The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37--46.

[Miller et al., 1990] Miller, B. P.; Clark, M.; Hollingsworth, J.; Kierstead, S.; Lim, S. S. & Torzewski, T. (1990). IPS-2: The Second Generation of a Parallel Program Measurement System. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206--217.

[Miller & Yang, 1987] Miller, B. P. & Yang, C.-Q. (1987). IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs. Em *Proceedings of the 7th International Conference on Distributed Computing Systems*, pp. 482--489. IEEE Computer Society Press.

[Mogul, 2003] Mogul, J. C. (2003). TCP offload is a dumb idea whose time has come. Em *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, pp. 25--30, Berkeley, CA, USA. USENIX Association.

[Morel & Renvoise, 1979] Morel, E. & Renvoise, C. (1979). Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22:96--103.

[Mostofi, 2009] Mostofi, H. (2009). Fast Level Set Segmentation of Biomedical Images using Graphics Processing Units. `http://code.google.com/p/cudaseg/`.

[Myer & Sutherland, 1968] Myer, T. H. & Sutherland, I. E. (1968). On the design of display processors. *Communications of the ACM*, 11(6):410--414.

[Mytkowicz et al., 2010] Mytkowicz, T.; Diwan, A.; Hauswirth, M. & Sweeney, P. F. (2010). Evaluating the accuracy of Java profilers. Em *Proceedings of the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '10, pp. 187--197, New York, NY, USA. ACM.

[Naumovich et al., 1999] Naumovich, G.; Avrunin, G. S. & Clarke, L. A. (1999). Data flow analysis for checking properties of concurrent Java programs. Em *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pp. 399--410, New York, NY, USA. ACM.

[Nethercote & Seward, 2007] Nethercote, N. & Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. Em *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 89--100, New York, NY, USA. ACM.

[Nguyen et al., 2007] Nguyen, H.; Sengupta, S. & Owens, J. D. (2007). *GPU Gems 3*, capítulo 39. Parallel Prefix Sum (Scan) with CUDA, pp. 851--876. Addison-Wesley Professional. Online version at `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html`.

[Nickolls & Dally, 2010] Nickolls, J. & Dally, W. J. (2010). The GPU Computing Era. *IEEE Micro*, 30(2):56--69.

[NVP, 2010] NVP (2010). NVIDIA Visual Profiler. `http://developer.nvidia.com/object/visual-profiler.html`.

[Ottenstein et al., 1990] Ottenstein, K. J.; Ballance, R. A. & MacCabe, A. B. (1990). The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. Em *Proceedings of the ACM SIGPLAN 1990 conference on Programming Language Design and Implementation*, PLDI '90, pp. 257--271, New York, NY, USA. ACM.

[Papadopoulou et al., 2008] Papadopoulou, M.-M.; Sadooghi-Alvandi, M. & Wong, H. (2008). Micro-benchmarking the GT200 GPU. `http://www.eecg.toronto.edu/~moshovos/CUDA08/arx/microbenchmark_report.pdf`.

[Patterson & Hennessy, 2008] Patterson, D. A. & Hennessy, J. L. (2008). *Computer Organization and Design: The Hardware/Software Interface*, capítulo A: Graphics and Computing GPUs., pp. A.1--A.77. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edição.

[Pereira & Palsberg, 2008] Pereira, F. M. Q. & Palsberg, J. (2008). Register Allocation by Puzzle Solving. Em *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 216--226, New York, NY, USA. ACM.

[Perrott, 1979] Perrott, R. H. (1979). A Language for Array and Vector Processors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):177--195.

[Petrini et al., 2002] Petrini, F.; Feng, W.-c.; Hoisie, A.; Coll, S. & Frachtenberg, E. (2002). The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22:46--57.

[PowerTop, 2009] PowerTop (2009). `http://www.lesswatts.org/projects/powertop/`.

[prof, 1979] prof (1979). *Unix Programmer's Manual, prof command*. Bell Laboratories, Murray Hill, NJ.

[PTX, 2010] PTX (2010). *NVIDIA Compute PTX: Parallel Thread Execution ISA*. NVIDIA Corporation. Version 2.2.

[rmm3, 2010] rmm3 (2010). *Intel Remote Management Module 3 Technical Product Specification*. Intel Corporation. `http://download.intel.com/support/motherboards/server/sb/e63789002_intel_rmm3_tps_v1_1.pdf`.

[Ryoo et al., 2008a] Ryoo, S.; Rodrigues, C. I.; Baghsorkhi, S. S.; Stone, S. S.; Kirk, D. B. & Hwu, W.-m. W. (2008a). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. Em *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 73--82, New York, NY, USA. ACM.

[Ryoo et al., 2008b] Ryoo, S.; Rodrigues, C. I.; Stone, S. S.; Baghsorkhi, S. S.; Ueng, S.-Z.; Stratton, J. A. & Hwu, W.-m. W. (2008b). Program optimization space pruning for a multithreaded gpu. Em *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code Generation and Optimization*, pp. 195--204, New York, NY, USA. ACM.

[Saha et al., 2009] Saha, B.; Zhou, X.; Chen, H.; Gao, Y.; Yan, S.; Rajagopalan, M.; Fang, J.; Zhang, P.; Ronen, R. & Mendelson, A. (2009). Programming model for a heterogeneous x86 platform. Em *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 431--440, New York, NY, USA. ACM.

[Scholz et al., 2008] Scholz, B.; Zhang, C. & Cifuentes, C. (2008). User-input dependence analysis via graph reachability. Relatório técnico TR-2008-171, Sun Microsystems, Inc., Mountain View, CA, USA.

[SDK, 2011] SDK (2011). NVIDIA GPU Computing SDK Code Samples. `http://developer.nvidia.com/object/cuda_3_2_downloads.html`. Version 3.2.

[Segall & Rudolph, 1985] Segall, Z. & Rudolph, L. (1985). PIE: A Programming and Instrumentation Environment for Parallel Processing. *IEEE Software*, 2(6):22--37.

[Seiler et al., 2008] Seiler, L.; Carmean, D.; Sprangle, E.; Forsyth, T.; Abrash, M.; Dubey, P.; Junkins, S.; Lake, A.; Sugerman, J.; Cavin, R.; Espasa, R.; Grochowski, E.; Juan, T. & Hanrahan, P. (2008). Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Transactions on Graphics (TOG)*, 27(3):1--15.

[Shende & Malony, 2006] Shende, S. S. & Malony, A. D. (2006). The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287--311.

[Shin, 2007] Shin, J. (2007). Introducing Control Flow into Vectorized Code. Em *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pp. 280--291, Washington, DC, USA. IEEE Computer Society.

[Shreiner et al., 2011] Shreiner, D.; Khronos OpenGL ARB Working Group; Licea-Kane, B. & Sellers, G. (2011). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.1*. Addison-Wesley Professional, 8th edição.

[Sites & Agarwal, 1988] Sites, R. L. & Agarwal, A. (1988). Multiprocessor cache analysis using ATUM. Em *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 186--195, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Smith & Waterman, 1981] Smith, T. F. & Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195--197.

[Srivastava & Eustace, 1994] Srivastava, A. & Eustace, A. (1994). ATOM: a system for building customized program analysis tools. Em *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pp. 196--205, New York, NY, USA. ACM.

[Stratton et al., 2010] Stratton, J. A.; Grover, V.; Marathe, J.; Aarts, B.; Murphy, M.; Hu, Z. & Hwu, W.-m. W. (2010). Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. Em *Proceedings of the 8th annual IEEE/ACM international symposium on Code Generation and Optimization*, CGO '10, pp. 111--119, New York, NY, USA. ACM.

[Sun-Yuan Kung et al., 1982] Sun-Yuan Kung; Arun, K. S.; Gal-Ezer, R. J. & Bhaskar Rao, D. V. (1982). Wavefront Array Processor: Language, Architecture, and Applications. *IEEE Transactions on Computers*, 31(11):1054--1066.

[Szafaryn et al., 2009] Szafaryn, L. G.; Skadron, K. & Saucerman, J. J. (2009). Experiences Accelerating MATLAB Systems Biology Applications. Em *Proceedings of the Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits (BiC) 2009*, pp. 1--4, New York, NY, USA. ACM.

[Tallent & Mellor-Crummey, 2009] Tallent, N. R. & Mellor-Crummey, J. M. (2009). Effective performance measurement and analysis of multithreaded applications. Em *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and*

Practice of Parallel Programming, PPoPP '09, pp. 229--240, New York, NY, USA. ACM.

[Tate et al., 2009] Tate, R.; Stepp, M.; Tatlock, Z. & Lerner, S. (2009). Equality saturation: a new approach to optimization. Em *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pp. 264--276, New York, NY, USA. ACM.

[Thompson et al., 2002] Thompson, C. J.; Hahn, S. & Oskin, M. (2002). Using modern graphics architectures for general-purpose computing: a framework and analysis. Em *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pp. 306--317, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Torczon & Cooper, 2007] Torczon, L. & Cooper, K. D. (2007). *Engineering a Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. Second edition.

[Wall, 1992] Wall, D. W. (1992). Experience with a software-defined machine architecture. *ACM Transactions Programming Language Systems*, 14(3):299--338.

[Weiser, 1981] Weiser, M. (1981). Program slicing. Em *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pp. 439--449, Piscataway, NJ, USA. IEEE Press.

[Williams et al., 2009] Williams, S.; Waterman, A. & Patterson, D. (2009). Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65--76.

[Wu & Larus, 1994] Wu, Y. & Larus, J. R. (1994). Static branch frequency and program profile analysis. Em *MICRO 27: Proceedings of the 27th annual International symposium on Microarchitecture*, pp. 1--11, New York, NY, USA. IEEE.

[Xu et al., 1999] Xu, Z.; Miller, B. P. & Naim, O. (1999). Dynamic instrumentation of threaded applications. Em *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pp. 49--59, New York, NY, USA. ACM.

[Yu & Acton, 2002] Yu, Y. & Acton, S. T. (2002). Speckle reducing anisotropic diffusion. *IEEE Transactions on Image Processing*, 11(11):1260--1270.