Técnicas de Otimização de Código para Placas de Processamento Gráfico

Fernando Magno Quintão Pereira

Abstract

The low cost and the increasing programmability of graphics processing units, popularly know as GPUs, is contributing to bring parallel programs closer to the reality of the everyday application developer. Presently, we can have access, for the price of an ordinary desktop, to a hardware boosting hundreds of processing elements. This brave new world brings, alongside the many possibilities, also difficulties and challenges. Perhaps, for the first time since the popularization of computers, it makes sense to the ordinary programmer to open the compiler books on the final chapters, which talk about very unusual concepts, such as polytopes, iteration space and Fourier-Motskin transformations. This material covers, in a very condensed way, some good practices and code optimization techniques that can be used, either by the compiler or by the developers themselves, to produce efficient code to graphics processing units. We will discuss a little bit of what are GPUs, which applications should target them, and which transformations we can apply on GPU programs to take more benefit from the SIMD execution model and the complex memory hiearchy that characterizes this hardware.

Resumo

O baixo custo e a crescente programabilidade das placas gráficas, normalmente conhecidas como GPUs, têm contribuído sobremaneira para trazer o processamento paralelo mais próximo da realidade de programadores comuns. Atualmente pode-se ter acesso, pelo preço de um computador popular, a um hardware que provê algumas centenas de processadores. Este admirável mundo novo, junto com as muitas possibilidades, traz também dificuldades e desafios. Talvez, pela

primeira vez desde a popularização dos computadores, faça sentido a programadores comuns abrir os livros de compiladores nos capítulos finais, onde conceitos até agora obscuros, como politopos e transformações de Fourier-Motskin vêm aguardando um momento de serem úteis fora do círculo dos especialistas em computação paralela. Este material, que explica de forma condensada alguns destes conceitos, tem o propósito de disseminar um pouco da ciência que suporta às placas de processamento gráfico. Muitas das técnicas descritas neste texto podem ser usadas tanto em compiladores, quanto por programadores que buscam melhorar o desempenho de suas aplicações paralelas. Embora algumas das otimizações descritas sejam específicas para o ambiente de desenvolvimento CUDA, vários métodos abordados neste texto podem ser aplicados sobre outras arquiteturas paralelas que seguem o modelo de execução SIMD.

1.1. A Ascensão das Placas Gráficas Programáveis

Placas gráficas programáveis são um conceito relativamente novo, por outro lado, elas se sustentam em uma longa cadeia de desenvolvimentos em Ciência da Computação. Nesta seção tentaremos contar um pouco desta história.

A Roda da Reencarnação. Os computadores de propósito geral que compramos nas lojas normalmente vêem 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 que podemos ver nos video-games. 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 tem um enorme poder computacional, mesmo porque as imagens que vemos brotar nos monitores de nossos computadores são cada vez mais exuberantes.

Mas nem sempre foi assim. Até meados da década de 90, pouco poder-se-ia dizer sobre hardware gráfico. Havia então a noção de um “vetor de imagens gráficas", uma possível tradução para *video graphics array*, or VGA. VGAs são exatamente isto: uma área contígua de dados 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, contendo poucas cores e resolução muito baixa.

Esta época saudosa, contudo, não durou muito. Novas aplicações foram surgindo. Jogos digitais e aplicações de CAD passaram a exigir processamento de imagens tridimensionais; usuários tornavam-se sempre mais exigentes. Esta contínua pressão sobre os aplicativos gráficos fez com que engenheiros de hardware movessem algumas das atividades de processamento para o que seriam as primeiras placas gráficas. Muitas destas atividades são computacionalmente intensivas. Rasterização, isto é, a conversão de vetores, pontos e outros elementos geométricos em pixels, é um exemplo deste tipo de programa. Rasterizadores foram os primeiros processadores gráficos implementados diretamente em hardware.

Shaders, ou processadores de sombras, vieram logo a seguir. O sombreamento é necessário para que 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 engenheiros da época foi criar sombreadores configuráveis. Isto é quase um hardware programável. Existem, inclusive, bibliotecas para a programação destes sombreadores. OpenGL é o exemplo mais notório, embora não seja o único.

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 de recursos computacionais. Um hardware configurável pede um conjunto de instruções. Neste caso, porque não adicionar a este processador super especializado a capacidade de realizar operações aritméticas e lógicas normalmente realizadas na CPU? E se estamos neste nível, pouco custa que queiramos também desvios condicionais. Temos então, um processador

Turing completo, que, embora projetado para lidar com gráficos, pode realizar outras tarefas, não necessariamente relacionadas à renderização de imagens em uma tela. Chegamos aqui às GPUs, do inglês *graphics processing units*, ou unidades de processamento gráficos.

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 estória é que, em nossa incessante busca por desempenho, tenhamos adicionado novos módulos, especializados e não programáveis, às GPUs. Os mais recentes modelos, por exemplo, possuem funções de rasterização implementadas em hardware, como era o caso das antigas CPUs. Aparentemente a *roda da reencarnação* deu uma volta e meia na história das placas gráficas.

Arquiteturas Heterogêneas. O conjunto CPU-GPU forma uma arquitetura de computadores heterogênea bem diferente do modelo CPU-VGA tão comum no passado. A figura 1.1 compara estas duas arquiteturas. Enquanto o vetor de processamento gráfico não possui qualquer componente programável, a GPU é um processador de propósito geral. De fato, temos dois processadores que trabalham juntos, ainda que em essência eles sejam substancialmente diferentes. Arquiteturas heterogêneas não são um modismo novo em ciência da computação. Vários computadores massiçamente paralelos utilizam aceleradores para determinados tipos de tarefa. Entretanto, tais computadores são muito caros e portanto, bem menos acessíveis que as nossas GPUs.

A GPU possui mais unidades de processamento que a CPU – muito mais, na verdade. Porém, os processadores presentes na GPU são mais simples e geralmente mais lentos que os processadores das CPUs. Outra diferença entre estes dois mundos se dá em termos de entrada e saída de dados. As GPUs possuem mecanismos de comunicação muito simples. Normalmente elas se comunicam somente com a CPU, via áreas de memória pré-definidas. Desta forma, aplicações muito iterativas, tais como editores de texto, não se beneficiariam muito do poder de processamento das GPUs. Por outro lado, aplicações em que exista muito paralelismo de dados brilham neste am-

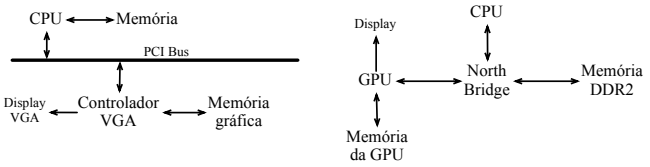

Figura 1.1. Um hardware gráfico composto de CPU e VGA comparado com um arranjo formado por CPU e GPU.

biente. Seria possível pensar que a GPU é um processador "escravo" da CPU, pois é ao nível da CPU que decide-se quais tarefas serão executadas na placa gráfica. Como veremos mais adiante, o desenvolvedor cria suas aplicações programando explicitamente o envio de dados para a GPU e a leitura dos dados que ela produz.

Preparar, Apontar, Fogo! A GPU, em seu núcleo, implementa um modelo computacional chamado SIMD, do inglês *Single Instruction Multiple Data*, ou "instrução única, dados múltiplos". No modelo SIMD temos vários processadores, mais somente uma unidade de processamento. Ou seja, uma máquina SIMD faz muitas coisas em paralelo, mas sempre as mesmas coisas!

E qual o benefício disto? O benefício reside no paralelismo de dados, o qual ilustraremos com uma analogia, não menos mórbida que educativa. Em um pelotão de fuzilamento temos uma parelha de soldados, todos armados com fuzis e um capitão, que lhos comanda via uma sequência de instruções bem conhecidas: "preparar, apontar, fogo". Cada uma destas instruções possui o mesmo efeito sobre os soldados, porém cada soldado manipula uma arma diferente. O capitão consegue, portanto, ordenar o disparo de quatro tiros em paralelo. Nesta metáfora, o capitão corresponde à unidade de controle e cada soldado representa um processador. As armas são os dados.

A GPU é um magnífico exemplo de hardware paralelo. Pa-

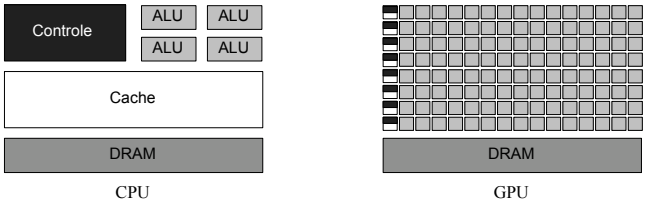

Figura 1.2. Uma comparação entre a GPU e a CPU, conforme figura obtida do manual de programação CUDA [4].

rafraseando o Manual de Boas Práticas da Nvidia [5]: “As GPUs da Nvidia suportam até 768 threads ativas por multiprocessador; algumas GPUs elevam este número a 1.024. Em dispositivos que possuem 30 multiprocessadores, tais como a GeForce GTX 280, isto faz com que 30,000 threads possam estar ativas simultaneamente". Este hardware poderosíssimo tem permitido que algumas aplicações pudessem executar até 100 vezes mais rapidamente que suas rivais restritas à CPU [18]. A figura 1.2 mostra uma típica representação da GPU, comparada com a CPU. A GPU separa uma porção muito grande de sua área útil para tarefas de processamento, ao passo que a CPU usa bastante desta área para implementar a sua memória de cache.

A GPU necessita de todo este paralelismo porque aplicações gráficas tendem a ser embaraçosamente paralelas. Isto quer dizer que tais aplicações são formadas por tarefas em geral independentes umas das outras. Por exemplo, podemos escurecer ou clarear os pixels em uma imagem atendo-nos a cada pixel em separado. Dada a natureza minimalista destas tarefas, a GPU pode se dar ao luxo de possuir núcleos de processamento simples, porém estes devem existir em grande número.

Alguns autores, principalmente aqueles mais ligados à Nvidia, costumam usar a expressão STMD (*Single Thread Multiple Data*) para caracterizar as GPUs. Este contudo, é um termo re-

lativamente novo e ainda pouco utilizado. Muitos outros autores referem-se às GPUs como arquiteturas MSIMD, isto é, máquinas formadas por múltiplas unidades SIMD. Em outras palavras, a GPU possui vários processadores, os quais executam instruções independentemente uns dos outros. Cada um destes processadores é, contudo, uma máquina SIMD.

Notem que ao denominar as placas gráficas de máquinas SIMD não estamos as comparando com as extensões vetoriais normalmente utilizadas em processadores de propósito geral, como as instruções SSE, SSE2 e MMX presentes em processadores Intel. Alguns autores chamam tais extensões de SIMD, uma vez que as instruções vetoriais permitem a realização da mesma operação sobre dados diferentes. Este modelo, contudo, possui uma diferença fundamental quando comparado ao modelo adotado em placas gráficas: a ausência de *execução condicional*. A GPU pode "desligar" algumas das threads que integram uma unidade SIMD. Tal capacidade é necessária para modelar desvios condicionais.

A fim de reforçar tal conceito com uma analogia, podemos voltar ao nosso exemplo do pelotão de fuzilamento. Em alguns círculos militares é tradição que um dos soldados do pelotão receba uma bala de festim, incapaz de causar qualquer ferimento. Tal engodo é realizado com o propósito de diluir a culpa da execução entre os atiradores. Assim, ao efetuar os disparos, um dos soldados, embora execute a mesma ação que os outros, não produzirá qualquer efeito. As placas gráficas utilizam um mecanismo similar para desativar threads, quando necessário: estas podem continuar executando a mesma ação que as demais threads, porém nenhum resultado de computação é escrito de volta na memória. Tal mecanismo, quando implementado em nível de software, é muitas vezes chamado *predicação*.

Uma linguagem de programação heterogênia. É natural que um hardware heterogêneo venha pedir por uma linguagem de programação heterogênea, em que o programador possa especificar quais partes do programa devem executar sobre a CPU e quais partes devem executar sobre a GPU. Existem linguagens assim. Entretanto, várias destas linguagens estão bastante voltadas para o domínio das aplicações gráficas e muitas delas

são, na verdade, APIs construídas sobre linguagens já conhecidas. Esta categoria inclui Cg (C para gráficos) e HLSL (High Level Shading Language). Cg, por exemplo, consiste em uma coleção de funções que podem ser invocadas via uma sintaxe que difere muito pouco da sintaxe da linguagem C. Existem, contudo, linguagens muito mais gerais – os exemplos mais notórios sendo OpenCL e C para CUDA.

Neste material nós utilizaremos C para CUDA para ilustrar a programação de GPUs. Esta linguagem é bastante diferente, em termos de semântica, de outras linguagens, tais como C, C++ ou Java, que também permitem o desenvolvimento de aplicações paralelas. C, C++, Java e muitas outras linguagens de programação foram projetas para atender as necessidades do hardware de propósito geral. Na verdade, estas linguagens tendem a abstrair o hardware, tornando-o quase invisível para o desenvolvedor de aplicações. Este pode, assim, concentrar-se em detalhes algorítmicos dos problemas de programação, não se preocupando com minúcias da arquitetura de computadores subjacente. Entretanto, não é uma tarefa simples abstrair a heterogeneidade de CUDA. Isto pode, obviamente, ser feito em algum futuro não muito distante; porém, tal ainda não foi feito em C para CUDA. Enquanto desenvolvendo aplicações para este ambiente, o programador decide quais partes de sua aplicação irão executar na CPU e quais serão enviadas para a GPU. Além disto, C para CUDA precisa prover ao programador mecanismos com que este possa usufruir do grande poder computacional presente em uma placa gráfica. Com tal intuito, a linguagem disponibiliza três abstrações principais:

- uma hierarquia de threads;
- memórias compartilhadas;
- barreiras de sincronização.

Um programa CUDA contém um corpo principal, que será executado sobre a CPU, e várias funções especiais, normalmente chamadas *kernels*, que serão executadas na GPU. Ao observar um kernel, temos a impressão de que seu texto trata-se de um programa sequencial. Entretanto, este mesmo código será simultaneamente executado por centenas de threads. O

ambiente de execução CUDA organiza estas threads em uma hierarquia formada por blocos e grades:

- Threads pertencentes ao mesmo bloco seguem o modelo SPMD (do inglês *Single Program Multiple Data*) de execução. Este é o modelo de paralelismo mais conhecido: temos um programa que será executado por múltiplos processadores, não necessariamente em passo único, como dar-se-ia em arquiteturas SIMD. Tais threads podem comunicar-se via memórias compartilhas e podem sincronizar-se via barreiras.
- Threads pertencentes à mesma grade executam o mesmo programa de forma independente, podendo comunicar-se somente via memória global.

Esta hierarquia existe para que o mesmo programa escrito em C para CUDA possa ser executados em placas gráficas de diferentes capacidades. Em geral uma grade de threads é executada de cada vez, porém, como o hardware pode não possuir processadores suficientes para cada thread, a grade é particionada em blocos e a uma quantidade maximal de blocos é dada a chance de realizar trabalho.

C para CUDA provê sintaxe para a identificação de threads. Cada thread criada neste ambiente de execução possui um nome, que é dado por um vetor tridimensional. Embora threads possam ser identificadas por três inteiros, muitas vezes utilizaremos somente um ou dois deles. Na verdade, a forma como o programador organiza as threads em um programa depende muito dos dados que este programa manipula. Normalmente aplicações que lidam com arranjos unidimensionais de dados irão utilizar somente o primeiro identificador de cada thread. Aplicações que lidam com matrizes irão utilizar dois identificadores, e aplicações que lidam com volumes irão utilizar todos os três identificadores. Ilustraremos estas noções mostrando como re-escrever o programa C na figura 1.3 em C para CUDA. Dados dois vetores x e y, este programa computa o produto vetorial $y = \alpha x + y$, para uma constante α qualquer. O programa correspondente em C para CUDA é mostrado na figura 1.4.

```
void saxpy_serial(int n, float alpha, float *x, float *y) {
  for (int i = 0; i < n; i++)
    y[i] = alpha*x[i] + y[i];
}
// Invoke the serial function:
saxpy_serial(n, 2.0, x, y);
```

Figura 1.3. Um programa em C que calcula o produto vetorial $y = \alpha x + y$. Este exemplo foi criado por Nickolls e Kirk [20].

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) y[i] = alpha * x[i] + y[i];
}
// Invoke the parallel kernel:
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Figura 1.4. Um programa em C para CUDA que calcula o produto vetorial $y = \alpha x + y$. Exemplo retirado de Nickolls e Kirk [20].

O Assembly de CUDA. Um programa escrito em uma linguagem de alto nível normalmente é traduzido para código binário a fim de ser executado. Em CUDA isto não é diferente. Existem compiladores que traduzem CUDA para instruções Direct3D, por exemplo. Neste material nós utilizaremos uma linguagem assembly conhecida por PTX, do inglês *Parallel Thread Execution*. O programa PTX que corresponde ao nosso primeiro kernel, visto na figura 1.4, é dado na figura 1.5. PTX, sendo uma linguagem assembly, manipula tipos de dados muito simples, como inteiros de 8, 16, 32 e 64 bits e números de ponto flutuante de 16, 32 e 64 bits. PTX descreve um típico código de três endereços, contendo instruções aritméticas (`add`, `sub`, `mul`, etc), lógicas (`eq`, `leq`, etc), de fluxo de controle (`bra`, `ret`, `call`, etc) e de transmissão de dados (`load`, `store`). PTX também

```
.entry saxpy (.param .s32 n, .param .f32 alpha,
              .param .u64 x, .param .u64 y) {
        .reg .u16 %rh<4>; .reg .u32 %r<6>; .reg .u64 %rd<8>;
        .reg .f32 %f<6>;  .reg .pred %p<3>;
$LBB1__Z9saxpy_parallelifPfS_:
        mov.u16         %rh1, %ctaid.x;
        mov.u16         %rh2, %ntid.x;
        mul.wide.u16    %r1, %rh1, %rh2;
        cvt.u32.u16     %r2, %tid.x;
        add.u32         %r3, %r2, %r1;
        ld.param.s32    %r4, [n];
        setp.le.s32     %p1, %r4, %r3;
       @%p1 bra         $Lt_0_770;
        cvt.u64.s32     %rd1, %r3;
        mul.lo.u64      %rd2, %rd1, 4;
        ld.param.u64    %rd3, [y];
        add.u64         %rd4, %rd3, %rd2;
        ld.global.f32   %f1, [%rd4+0];
        ld.param.u64    %rd5, [x];
        add.u64         %rd6, %rd5, %rd2;
        ld.global.f32   %f2, [%rd6+0];
        ld.param.f32    %f3, [alpha];
        mad.f32         %f4, %f2, %f3, %f1;
        st.global.f32   [%rd4+0], %f4;
$Lt_0_770:
        exit;
}
```

Figura 1.5. A versão PTX do programa da figura 1.4.

contém muitas instruções para calcular funções transcendentais como senos, cossenos e arco-tangentes. Além destas instruções, também presentes em linguagens assembly mais tradicionais, tais como x86 ou PowerPC, PTX contém instruções que lidam com as particularidades da execução paralela, incluindo aquelas que manipulam barreiras de sincronização, leitura e escrita atômicas e acesso à memórias compartilhadas.

E a partir daqui? Existem duas fontes principais de otimizações em programas CUDA: a memória e o fluxo de controle dos programas. Na próxima seção daremos uma olhada no primeiro destes itens, ficando o segundo para a nossa última seção.

1.2. Otimizações de memória

Nesta seção descreveremos a hierarquia de memória que caracteriza as GPUs, dando ênfase às boas práticas de programação que possibilitam um melhor uso desta hierarquia.

A hierarquia de memória. As arquiteturas de computadores

normalmente organizam a memória em uma hierarquia. Em direção ao topo desta hierarquia temos os dispositivos de armazenamento mais rápidos e também mais caros e menores, tais como os registradores. Na direção oposta temos a memória mais abundante, mais barata e, infelizmente, de acesso mais lento, tal como o disco rígido. E entre registradores e discos rígidos temos vários níveis de cache e a memória RAM. A GPU também possui uma hierarquia de memória e o conhecimento sobre a organização desta hierarquia é uma informação fundamental para o desenvolvimento de aplicações eficientes.

Uma GPU tende a manipular quantidades muito grandes de dados. Cada imagem que deve ser renderizada em uma tela de alta resolução contém milhões de pixels. A título de exemplo, a placa GeForce 8800 é capaz de manipular 32 pixels por ciclo de execução. Cada pixel contém uma cor, representada por três bytes e uma profundidade, representada por quatro bytes. Além disto, o processamento de um único pixel, em média, leva à leitura de outros 16 bytes de informação. Isto quer dizer que 736 bytes são manipulados por ciclo de execução – uma quantidade considerável, quando comparada a uma típica instrução de CPU, que manipula cerca de 12 bytes por ciclo de execução.

Diferentes GPUs utilizam diferentes tipos de hierarquias de memória, mas um arranjo típico consiste em separar a memória nos grupos abaixo relacionados:

Registradores: registradores são rápidos, porém poucos. Cada thread possui um conjunto privado de registradores.

Compartilhada: threads no mesmo bloco compartilham uma área de memória, a qual funciona como um cache manipulado explicitamente pelo programa.

Local: cada thread possui acesso a um espaço de memória local, além de seus registradores. Observe que a palavra “local" não implica em acesso rápido: esta área de memória está fora do micro-chip de processamento, junto à memória global, e portanto, ambas estas memórias possuem o mesmo tempo de acesso.

Global: esta memória está disponível para todas as threads em cada bloco e em todas as grades. Na verdade, a memó-

ria global é a única maneira de threads em uma grade conversarem com threads em outra.

Os registradores e as memórias compartilhadas, locais e globais estão dentro da placa gráfica, apesar de somente a memória compartilhada estar dentro do micro-chip de processamento. Existe, contudo, uma outra memória, que também é importante, mas que está fora da placa: trata-se da memória DRAM utilizada para que a GPU e a CPU possam comunicar-se. Esta memória não é parte da GPU; ao contrário, ela está para a GPU assim como o disco rígido está para a CPU. Logo, ela encontra-se no limite inferior de nossa hierarquia de memória, sendo extremamente abundante, porém apresentando um tempo de acesso muito lento. A figura 1.6 coloca a hierarquia de memória em perspectiva com a hierarquia de threads.

Existem algumas leis que devem ser obedecidas para que a memória possa ser utilizada de forma eficiente em GPUs:

1. os registradores são as unidades de armazenamento mais rápidas e também as mais limitadas. A GPU provê um número fixo de registradores para o bloco de threads. Assim, quanto mais registradores uma thread usar, menos threads poderão existir em um bloco;

2. o acesso à memória local ou global é muito lento. Threads deveriam ser capazes de compartilhar a maior quantidade possível de dados em memória compartilhada.

3. a comunicação entre dispositivos, isto é, entre a CPU e a GPU, é muito cara. Este canal é algumas ordens de magnitude mais lento que a memória compartilhada, e mesmo as memórias globais e locais. Logo, este tipo de comunicação deve ser diminuído tanto quanto possível. Esta é uma das principais razões porque GPUs não são adequadas para aplicações iterativas.

A viagem inter-planetária. Comparemos o tempo que uma thread gasta para ler o valor armazenado em um registrador com o tempo que a CPU leva para enviar esta mesma informação para a GPU. Valendo-me de uma analogia, poder-se-ia dizer que a primeira leitura equivale a atravessar uma rua deserta e sem

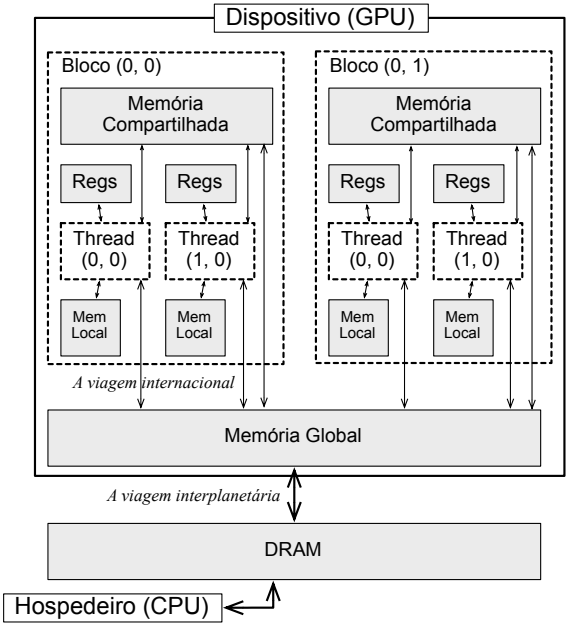

Figura 1.6. A organização de memórias dentro da GPU.

semáforos, ao passo que a segunda leitura equivale a uma viagem à lua – caminhando. Em outras palavras, para usar a GPU, a CPU precisa lhe enviar dados, o que é feito por um barramento PCI (*Peripheral Component Interconnect*). Esta tranferência é muito lenta, comparada com o tempo de acesso de dados dentro do micro-chip CUDA.

A transferência de dados entre CPU e GPU é ilustrada pelo programa na figura 1.7. As funções `cudaMalloc`, `cudaFree` e `cudaMemcpy` são parte da biblioteca de desenvolvimento CUDA. A primeira função reserva uma área na memória da GPU, ao

```
void Mul(const float* A, const float* B, int width, float* C)  {
    int size = width * width * sizeof(float);

    // Load B and C to the device
    float* Bd;
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
    float* Cd;
    cudaMalloc((void**)&Cd, size);
    cudaMemcpy(Cd, C, size, cudaMemcpyHostToDevice);

    // Allocate A on the device
    float* Ad;
    cudaMalloc((void**)&Ad, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(width / dimBlock.x, width / dimBlock.y);

    // Launch the device computation
    matMul<<<dimGrid, dimBlock>>>(Bd, Cd, Ad, width);

    // Read A from the device
    cudaMemcpy(A, Ad, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
```

Figura 1.7. Código escrito para a CPU, que invoca um kernel da GPU, passando-lhe matrizes de dados e lendo de volta uma matriz de dados.

passo que a segunda função libera esta área para outros usos. Finalmente, `cudaMemcpy` copia dados da CPU para a GPU, ou vice-versa, dependendo de seu último argumento.

Existem importantes aspectos de desempenho que precisam ser considerados, quando escrevemos programas CUDA, no que toca a comunicação entre GPU e CPU. Em primeiro lugar, é justificável o envio de dados para a GPU somente quando o trabalho que será executado sobre estes dados possui alta complexidade. Por exemplo, é possível paralelizar a soma de duas matrizes de mesma dimensão atribuindo a cada thread a tarefa de produzir um elemento desta adição, o que será feito via

a soma de dois elementos em posições equivalentes das matrizes parcela. Neste caso, cada thread recebe uma quantidade de trabalho constante, isto é, cuja complexidade assimptótica é $O(1)$, além disto, esta constante, neste caso particular, é bem pequena. Logo, o trabalho de enviar os dados para a GPU, o que é feito de forma sequencial, já é maior que o trabalho realizado dentro da placa gráfica. Em vista destas observações, é fácil concluir que a razão entre o trabalho para transferir dados e o trabalho realizado sobre estes dados não é boa. A fim de transmitir as duas matrizes parcela e recuperar a matriz resultado são realizadas $3N^2$ acessos à memória. O kernel realiza N^2 adições. Portanto, a razão entre transferência e trabalho que encontramos é de $1/3$. Por outro lado, no caso de multiplicação de matrizes temos um cenário muito melhor. Este algoritmo, implementado nas figuras 1.7 e 1.8, transfere a mesma quantidade de dados que o algoritmo de adição de matrizes, porém realiza N^3 adições e multiplicações. Temos então uma razão entre transferência e trabalho de $N^3/3N^2 = O(N)$.

Em segundo lugar, é importante manter dados na memória da GPU tanto quanto possível e necessário. Uma vez que dados são enviados à GPU, eles permanecem na memória da placa gráfica, mesmo que o kernel que os processou já tenha terminado. Assim, programas que usam múltiplos kernels podem evitar a "viagem interplanetária" simplesmente invocando novos kernels sobre dados já transferidos. Esta abordagem pode ser benéfica mesmo que o trabalho intermediário a ser realizado sobre tais dados pudesse ser mais rapidamente feito na CPU – um algoritmo sequencial executado na GPU pode ser mais rápido que o uso do barramento externo de memória.

Acesso agrupado à memória global. Os dados passados à GPU pela CPU ficam armazenados em uma área de dados conhecida como memória global. Nesta seção discutiremos alguns fatores que influenciam o bom uso desta memória. Nós utilizaremos o programa de multiplicação matricial usado por Ryoo *et al.* [22] a fim de demonstrar o uso eficiente tanto da memória global quanto de outros níveis de memória, os quais serão apresentados em seções posteriores. Este kernel é obtido da paralelização do programa visto na figura 1.8.

```
// Computes the matrix product using line matrices:
void mulMatCPU(float* B, float* C, float* A, unsigned W) {
  for (unsigned int i = 0; i < W; ++i) {
    for (unsigned int j = 0; j < W; ++j) {
      A[i * W + j] = 0.0;
      for (unsigned int k = 0; k < W; ++k) {
        A[i * W + j] += B[i * W + k] * C[k * W + j];
      }
    }
  }
}
```

Figura 1.8. Programa escrito em C que realiza o produto matricial $A = B \times C$, usando matrizes linearizadas.

```
__global__ void matMul(float* B, float* C, float* A, int W) {
  float Pvalue = 0.0;

  int tx = blockIdx.x * blockDim.x + threadIdx.x;
  int ty = blockIdx.y * blockDim.y + threadIdx.y;

  for (int k = 0; k < W; ++k) {
    Pvalue += B[tx * W + k] * C[k * W + ty];
  }

  A[ty + tx * W] = Pvalue;
}
```

Figura 1.9. Kernel que realiza o produto matricial $A = B \times C$. Esta é uma paralelização do algoritmo visto na figura 1.8.

Em um modelo PRAM de processamento é possível realizar a multiplicação matricial em tempo $O(\ln N)$. Esta solução, contudo, não é exatamente trivial e nós iremos utilizar, em vez dela, uma solução $O(N)$ no modelo PRAM, em que N é a largura da matriz. O algoritmo escrito em C para CUDA é dado na figura 1.9. Cada thread é reponsável pela produção de um único elemento A_{ij} na matriz final A. Para produzir tal elemento, a thread deve realizar um produto escalar entre a i-ésima linha da matriz B e a j-ésima linha da matriz C.

Placas gráficas diferentes agrupam os dados em memória

global de várias maneiras; porém, suporemos que esta memória está particionada em segmentos de 16 palavras cada. Caso 16 threads que formam a metade de um warp precisem ler dados localizados em um mesmo segmento, então podemos efetuar esta transação via somente uma viagem à memória global. Dizemos, neste caso, que o acesso à memória é *agrupado*. Por outro lado, se estas threads buscarem dados em segmentos diferentes, então será feita uma viagem para casa segmento distinto. A fim de garantir que a leitura ou escrita de memória global seja agrupada, podemos utilizar os seguintes passos de verificação:

1. Considere a thread t, cujo identificador seja um múltiplo de 16, isto é, $t = 16 \times n$.
2. determine o segmento de dados s que t está usando.
3. para cada thread $t + i, 1 \leq i \leq 15$, verifique se $t + i$ está também usando s.

A fim de entender o impacto de acessos agrupados sobre o desempenho de aplicações CUDA, consideremos o kernel `mat Mul` da figura 1.9. Lembrando que o índice the uma thread em um arranjo bidimensional de threads, é dado por $tid = x + y \times Dx$, sendo Dx o número de threads no eixo x, temos que o índice x de uma thread varia mais rapidamente. Logo, para que tenhamos o acesso agrupado, é preciso que as threads $(0,0),(1,0),\ldots,(15,0)$ leiam dados em um mesmo segmento. Este alinhamento não acontece nas leituras da matriz `B`, pois, supondo $t_x = 0,1,2,\ldots,15$ e `W = 600`, estaríamos lendo as posições `B[0]`, `B[600]`, ..., `B[9000]`. O acesso agrupado também não ocorre na escrita da matriz `A`, pois o índice t_x, que varia mais rapidamente que o índice t_y, está multiplicando a variável `W`.

Uma forma de agrupar os acessos à memória global, neste caso, é via uma simples re-ordenação de índices, conforme feito na figura 1.10. Vê-se que este programa usa acessos agrupados em todas as leituras e escritas de dados. Por exemplo, novamente supondo $(t_x,t_y) = (0,0),(1,0),\ldots,(15,0)$, e `W = 600`, as posições lidas da matrix `B` serão sempre as mesmas para todas as threads em meio warp e as posições lidas da matriz `C` serão consecutivas e portanto contidas em um segmento de dados.

```
__global__ void matMulCoalesced(float* B, float* C, float* A, int W) {
  float Pvalue = 0.0;

  int tx = blockIdx.x * blockDim.x + threadIdx.x;
  int ty = blockIdx.y * blockDim.y + threadIdx.y;

  for (int k = 0; k < W; ++k) {
    Pvalue += B[ty * W + k] * C[k * W + tx];
  }

  A[tx + ty * W] = Pvalue;
}
```

Figura 1.10. Modificação do programa da figura 1.9 para usar acessos agrupados à memória global.

Esta simples modificação de código nos dá um programa que é quase 10 vezes mais rápido que aquele programa visto na figura 1.9, quando executado em uma placa GPU 9400M!

A memória global e a memória compartilhada. Uma boa maneira de medirmos o desempenho de uma função em C para CUDA é contarmos quantas operações de ponto-flutuante esta função realiza por segundo [22]. Este número é conhecido por (FLOPS) e para obtê-lo, faz-se conveniente observarmos o programa PTX produzido pelo laço mais interno do programa na figura 1.10, o qual é mostrado na figura 1.11. Aproximadamente $1/8$ das instruções dentro do laço deste programa são operações de ponto flutuante: existem 16 instruções, uma delas, *multiply-add*, equivale a duas operações. Considerando-se que uma placa GTX 8800 realiza 172.8 bilhões de operações de ponto-flutuante por segundo, então temos, neste hardware, uma taxa de trabalho de 21.6 GFLOPS. Entretanto, ao medir-se o desempenho deste programa em tal placa, percebe-se que a taxa de trabalho real é bem menor: 10.58 GFLOPS [22]. O gargalo, neste caso, é a forma de utilização da memória.

O programa da figura 1.10 realiza muitos acessos redundante à memória global. Cada thread lê dois vetores de tamanho N e escreve um vetor de tamanho N. Ou seja, dado que temos N^2 threads, uma para cada posição da matriz produto,

```
        mov.f32         %f1, 0f00000000;   // 0.0F
        mov.s32         %r10, %r5;
$Lt_0_1282:
        cvt.u64.u32     %rd3, %r7;
        mul.lo.u64      %rd4, %rd3, 4;
        ld.param.u64    %rd2, [B];
        add.u64         %rd5, %rd2, %rd4;
        ld.global.f32   %f2, [%rd5+0];
        cvt.u64.u32     %rd6, %r9;
        mul.lo.u64      %rd7, %rd6, 4;
        ld.param.u64    %rd1, [C];
        add.u64         %rd8, %rd1, %rd7;
        ld.global.f32   %f3, [%rd8+0];
        mad.f32         %f1, %f2, %f3, %f1;
        add.u32         %r7, %r7, 1;
        ld.param.s32    %r3, [Width];
        add.u32         %r9, %r3, %r9;
        setp.ne.s32     %p2, %r7, %r8;
        @%p2 bra        $Lt_0_1282;
        bra.uni         $Lt_0_770;
```

Figura 1.11. A versão PTX do programa visto na figura 1.9.

então temos $3N^3$ acessos à memória global. Poderíamos diminuir esta redundância via um *cache*. Poucas GPUs provêem um cache para a memória global; entretanto, o desenvolvedor pode programar um cache explicitamente usando a memória compartilhada. A memória compartilhada é cerca de 100 vezes mais rápida que a memória global. Esta memória pode ser entendida como um cache manipulado diretamente pelo desenvolvedor CUDA. Isto é, arquiteturas tradicionais tendem a controlar o cache, de forma tal que o programador não precise se preocupar em definir quais dados devem ser alocados lá. O hardware é, atualmente, muito bom em controlar caches. GPUs mais antigas, contudo, não possuem este tipo de gerência automática de cache – fica a cargo do programador decidir quais dados devem ser armazenados na memória de rápido acesso.

A solução vista na figura 1.10 é relativamente ingênua, pois as threads não compartilham dados, ainda que elas usem uma grande quantidade de informação redundante. Um quarto das instruções na figura 1.11 lêem ou escrevem dados em memória global. A fim de utilizar-se completamente a GPU, seria necessário uma banda de transmissão de dados de 173GB/s (128 th-

```
__global__ void matMulTiled(float* B, float* C, float* A, int Width) {
  __shared__ float Bs[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Cs[TILE_WIDTH][TILE_WIDTH];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Identify the row and column of the A element to work on
  int Row = blockIdx.x * TILE_WIDTH + tx;
  int Col = blockIdx.y * TILE_WIDTH + ty;

  float Pvalue = 0;
  // Loop over the B and C tiles required to compute the A element
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Coolaborative loading of B and C tiles into shared memory
    Bs[ty][tx] = B[Col * Width + (m * TILE_WIDTH + tx)];
    Cs[ty][tx] = C[Row + (m * TILE_WIDTH + ty) * Width];
    __syncthreads();

    #pragma unroll 1
    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Bs[ty][k] * Cs[k][tx];
    __syncthreads();
  }
  A[Col * Width + Row] = Pvalue;
}
```

Figura 1.12. Versão ladrilhada do kernel de multiplicação de matrizes.

reads $\times$ $1/4$ $\times$ 4 Bytes $\times$ 1.35GHz), a qual é consideravelmente maior que a banda fornecida pela placa: 86.4GB/s. O uso eficiente da memória compartilhada pode reduzir este problema.

Ladrilhamento. Uma conhecida forma de otimização de código, usada para aumentar a proximidade entre os dados manipulados dentro de laços de execução é o *ladrilhamento* [16]. Nesta otimização, os dados a serem manipulados são divididos em blocos e dados nos mesmos blocos, supostamente próximos no cache, são usados em conjunto. O ladrilhamento é particularmente útil em nosso caso, pois podemos dividir a tarefa de multiplicar duas grandes matrizes em várias tarefas de multiplicar matrizes menores, com um mínimo de interferência. A figura 1.12 mostra o nosso exemplo ladrilhado.

O ladrilhamento pode levar a programas de código bastante complicado e o programa da figura 1.12 não foge à regra. Normalmente nós calcularíamos uma célula da matriz A percor-

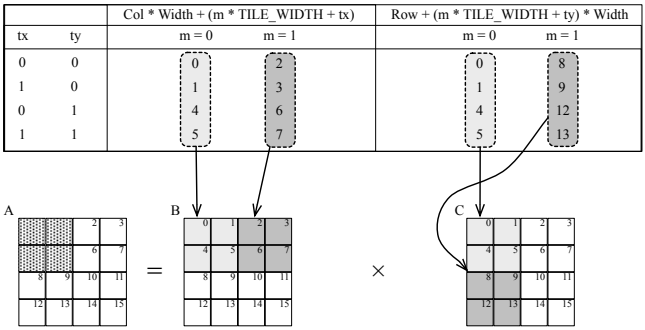

Figura 1.13. Duas iterações do bloco de threads $(0,0)$ do programa da figura 1.12.

rendo uma linha da matriz B e uma coluna da matriz C. No programa ladrilhado, contudo, estamos fazendo isto em partes. Copiamos um ladrilho de dados da matriz B e outro da matriz C para a memória compartilhada. Então operamos sobre estes ladrilhos, efetuando um "mini-produto" matricial, computando, assim, o resultado parcial de várias células de A. A fim de obter o valor final destas células, precisamos processar uma linha completa de ladrilhos de B e uma coluna completa de ladrilhos de C, conforme ilustramos na figura 1.13, em que estamos supondo uma matriz de tamanho 4×4 e ladrilhos de tamanho 2×2. A figura mostra as duas iterações necessárias para produzir o ladrilho do canto superior esquerdo de A.

O programa ladrilhado ainda contém leituras repetidas de dados da memória global, porém, a quantidade de redundâncias é bem menor quando comparada com as redundâncias do programa visto na figura 1.10. A fim de demonstrar este fato, seja $N^2/2$ o tamanho de cada ladrilho. Temos então, um total de $2(N/(N/2))^2$ ladrilhos. Cada ladrilho é lido $N/(N/2)$ vezes. Portanto, temos $4N^2$ leituras da memória global. Dadas as $2N^3$ leituras do programa original, observamos uma redução subs-

```
__global__ void matMulUnroll(float* B, float* C, float* Pd, int Width) {
  __shared__ float Bs[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Cs[TILE_WIDTH][TILE_WIDTH];
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int Row = blockIdx.x * TILE_WIDTH + tx;
  int Col = blockIdx.y * TILE_WIDTH + ty;
  float Pvalue = 0;
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    Bs[ty][tx] = B[Col * Width + (m * TILE_WIDTH + tx)];
    Cs[ty][tx] = C[Row + (m * TILE_WIDTH + ty) * Width];
    __syncthreads();
    Pvalue += Bs[ty][ 0] * Cs[ 0][tx];
    Pvalue += Bs[ty][ 1] * Cs[ 1][tx];
    Pvalue += Bs[ty][ 2] * Cs[ 2][tx];
    Pvalue += Bs[ty][ 3] * Cs[ 3][tx];
    Pvalue += Bs[ty][ 4] * Cs[ 4][tx];
    Pvalue += Bs[ty][ 5] * Cs[ 5][tx];
    Pvalue += Bs[ty][ 6] * Cs[ 6][tx];
    Pvalue += Bs[ty][ 7] * Cs[ 7][tx];
    Pvalue += Bs[ty][ 8] * Cs[ 8][tx];
    Pvalue += Bs[ty][ 9] * Cs[ 9][tx];
    Pvalue += Bs[ty][10] * Cs[10][tx];
    Pvalue += Bs[ty][11] * Cs[11][tx];
    Pvalue += Bs[ty][12] * Cs[12][tx];
    Pvalue += Bs[ty][13] * Cs[13][tx];
    Pvalue += Bs[ty][14] * Cs[14][tx];
    Pvalue += Bs[ty][15] * Cs[15][tx];
    __syncthreads();
  }
  Pd[Col * Width + Row] = Pvalue;
}
```

Figura 1.14. Multiplicação de matrizes após a aplicação do desenrolamento.

tancial destes acesso, com consequâncias não menos notáveis para o desempenho da aplicação ladrilhada: o programa da figura 1.12 é aproximadamente 25% mais rápido que o programa da figura 1.10, quando executado em uma placa GPU 9400M.

Desenlaço. Aumentamos a taxa de trabalho útil realizado por um kernel diminuindo a quantidade de instruções que não realizam operações de ponto-flutuante – no caso de um programa que produz saída neste formato. Entre estas instruções, destacam-se as operações de acesso à memória e as operações que controlam o fluxo do programa: desvios condicionais e incondicionais. Tendo lidado com o primeiro grupo de operações na seção anterior, voltamos agora nossa atenção para o segundo. Laços normalmente contêm um desvio que testa uma condição sobre uma variável de indução. O *desenlaço*, uma otimização

Kernel	Figura	Speed-up	Registradores	Ocupação
`matMul`	1.9	1	9	1
`matMulCoalesced`	1.10	9.54	9	1
`matMulTiled`	1.12	11.75	11	2/3
`matMulUnroll`	1.14	26.28	9	1

Tabela 1.1. Resultados obtidos durante a otimização do kernel de multiplicação matricial (GPU 9400M).

de código clássica, diminui o número de execuções destes testes, podendo, inclusive, diminuir a necessidade de registradores no interior do laço. Continuando com o exemplo de Ryoo *et al.* [22], vemos, na figura 1.14, o resultado do desenlaço aplicado ao programa da figura 1.12. Nesta versão do programa estamos supondo ladrilhos de tamanho 16×16, e, indo contra todos os bons princípios de programação, mas justificados pela simplicidade da exposição, nós utilizamos esta quantidade explicitamente na figura 1.14.

O código PTX do programa visto na figura 1.14 revela que o laço mais interno deste algoritmo contém 59 instruções e destas, 16 são de ponto-flutuante. Temos, assim, uma taxa de trabalho de 93.72 GFLOPS. Dado que nós desenrolamos completamente o laço mais interno da figura 1.12, o registrador utilizado para armazenar a variável de indução `m` não é mais necessário. Desta forma, o desenlaço não somente aumentou a taxa de trabalho útil realizado pelo programa, mas também reduziu a pressão por registradores. Conforme veremos a seguir, esta redução é importante para aumentar a quantidade de threads que podem ser executadas simultaneamente na placa gráfica. O programa desenlaçado é 2.25x mais rápido que o programa da figura 1.12. A tabela 1.1 contém um sumário destes resultados. Conforme nos mostra esta tabela, o kernel `matMulTiled`, primeiro a utilizar a memória compartilhada, não é capaz de aproveitar a máxima quantidade de threads disponíveis, pois ele necessita de mais registradores que os outros kernels.

Registradores. Tanto em CPUs quanto em GPUs, registrado-

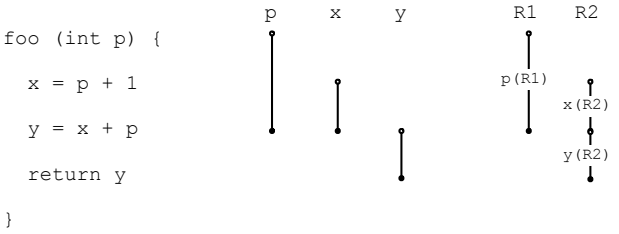

Figura 1.15. Pressão de registradores em uma sequência linear de código.

res são as unidades de armazenamento que permitem a leitura e escrita mais rápidas. E, assim como em CPUs, GPUs também fornecem uma quantidade limitada de registradores para a execução de programas. Por exemplo, a placa GeForce 8800 possui 16 multiprocessadores e cada um deles possui 8.192 registradores. Este número pode parecer grande a princípio, porém ele será compartilhado entre até 768 threads. Logo, a fim de obtermos a máxima ocupação do hardware, cada thread poderia utilizar no máximo 10 registradores.

Desta forma, existe uma tensão entre o número de registradores disponíveis para cada thread e a quantidade de threads que podem ser executadas simultaneamente. Por exemplo, o programa da figura 1.9 usa 9 registradores. Portanto, ele pode executar a quantidade máxima de 768 threads, uma vez que $9 \times 768 < 8,192$. Ou seja, uma placa GeForce 8800 pode escalonar três blocos, cada um com 256 threads, para executarem juntos. Todavia, caso a pressão de registradores aumente em duas unidades, como deu-se no programa da figura 1.12, então teríamos $256 \times 11 \times 3 = 8,488 > 8,192$. Neste caso apenas dois blocos de threads poderiam ser executados em conjunto, resultando em uma taxa de ocupação do hardware de 2/3.

O cálculo da *pressão de registradores*, isto é, o número máximo de registradores demandado por um programa é um

problema difícil e muitas de suas variações são problemas NP-completos. Duas variáveis podem ser alocadas no mesmo registrador se elas não estão “vivas" no mesmo ponto do programa. Dizemos que uma variável v está viva em um determinado ponto de programa p se existe um caminho no fluxo de execução do programa que vai de p até um outro ponto de programa p' onde v é usada. Este caminho não deve passar por nenhuma definição da variável v. A figura 1.15 ilustra estas noções. Nesta figura temos uma função que possui um parâmetro de entrada e duas variáveis. Este parâmetro e as variáveis precisam ser armazenados em alguma área de memória. Idealmente gostaríamos de usar registradores para tal armazenamento. As linhas de vida das variáveis, isto é, os pontos de programa onde estas variáveis estão vivas, são mostradas como segmentos de linha na figura. É fácil notar que a linha de vida de `y` não se sobrepõe à linha de vida de nenhuma das outras variáveis. Logo, `y` pode compartilhar um registrador com qualquer destas variáveis. A direita da figura mostramos como seria uma possível alocação usando dois registradores, em que às variáveis `y` e `x` foi dado o mesmo registrador.

A pressão de registradores na figura 1.15 é dois – o número máximo de linhas de vida que se sobrepõem. Não seria possível, por exemplo, armazenar `p` e `x` no mesmo registrador: estas variáveis estão vivas no mesmo ponto de programa. O cálculo deste valor neste exemplo, contudo, é bastante simples, pois o programa não possui desvios condicionais. O grafo de fluxo de controle do programa é uma linha e as linhas de vida formam um grafo de intervalos, o qual pode ser colorido em tempo polinomial. O problema naturalmente torna-se mais complicado uma vez que os programas passem a ter desvios condicionais e laços. Appel e Palsberg fornecem uma explicação detalhada sobre um algoritmo de alocação de registradores baseado na coloração de grafos [3].

1.3. Análises e Otimizações de Divergência

Muitas das otimizações que vimos na Seção 1.2, tais como ladrilhamento e alocação de registradores, são técnicas clássicas de compilação [1], podendo ser usadas tanto em programas

sequenciais quanto em programas paralelos. Existem, contudo, otimizações de código que fazem sentido apenas quando usadas sobre máquinas SIMD, como os processadores de warps em CUDA. Esta peculiaridade deve-se a um fenômeno denominado *execução divergente*, o qual descreveremos nesta seção.

Warps de threads. Em uma GPU, threads são organizadas em grupos que executam em *passo único*, chamados *warps* no jargão da Nvidia e *frentes de onda* no jargão da ATI. Para entender as regras que governam o funcionamento das threads em um mesmo warp, imaginemos que cada warp pode usar várias unidades lógicas e aritméticas; porém, existe somente um buscador de instruções por warp. Por exemplo, a placa GeForce 8800 é capaz de executar 16 warps simultaneamente. Cada warp agrupa 32 threads e pode usar 8 unidades lógicas e aritméticas. Como existe somente um buscador de instruções, cada warp pode executar 32 instâncias da mesma instrução em quatro ciclos de pipeline da GPU. Cada uma destas instruções, embora designando a mesma operação, não precisa usar os mesmos dados. Trazendo de volta nossa analogia do pelotão de fuzilamento, cada soldado seria uma unidade de processamento, o capitão que lhes grita as ordens faz o papel do buscador de instruções e cada fuzil representa os dados sendo processados. Evidentemente este modelo de execução é mais eficiente quando usado sobre aplicações regulares em que todas as threads podem executar as mesmas tarefas. Infelizmente muitas das aplicações que gostaríamos de paralelizar não são tão regulares e divergências podem acontecer entre as threads. Dizemos que um programa é divergente se durante sua execução threads agrupadas no mesmo warp seguem caminhos diferentes logo após terem processado um desvio condicional.

Divergências ocorrem, então, devido à desvios condicionais. A condição de desvio pode ser verdadeira para algumas threads e falsa para outras. Dado que cada warp possui acesso no máximo a uma instrução a cada ciclo de execução, em vista de divergências algumas threads terão de esperar ociosas enquanto as outras perfazem trabalho. Desta forma, divergências podem ser uma das grandes fontes de degradação de desempenho em aplicações GPU. A título de ilustração, Baghsorkhi

et al [7] mostraram, analiticamente, que aproximadamente um terço do tempo de execução do soma paralela de prefixos [14], disponível no pacote de desenvolvimento da Nvidia, perde-se devido às divergências.

A transformação de um programa com o intuito de diminuir os efeitos de divergências não é uma tarefa imediata por várias razões. Em primeiro lugar, alguns algoritmos são inerentemente divergentes, isto é, seu comportamento, divergente ou não, depende de dados de entrada. Em segundo lugar, o problema de encontrar os pontos de programa onde ocorrem as mais sérias divergências coloca uma carga relativamente grande sobre o programador, que precisa escanear manualmente códigos muitas vezes longos e complicados.

Ordenação bitônica. Ilustraremos os conceitos de divergência, bem como as técnicas de otimização que podem ser usadas em programas divergentes, usando como exemplo o algoritmo de ordenação bitônica. A ordenação vetorial não é tão regular quanto, por exemplo, a multiplicação matricial que vimos na seção anterior. A ordenação exige uma quantidade razoável de coordenação entre as threads, e divergências acontecem naturalmente, devido ao vetor a ser ordenado. Uma boa abordagem para a ordenação paralela são as *redes de ordenação*.

Uma das redes de ordenação mais conhecidas é o algoritmo de ordenação bitônica [8]. Este algoritmo apresenta uma complexidade assimptótica de $O(n \ln^2 n)$, quando restrito a um hardware sequencial. Ainda assim, a habilidade de realizar $n/2$ comparações em paralelo, onde n é o tamanho do vetor de entrada, torna este algoritmo muito atrativo para o ambiente SIMD de uma GPU. Nesta seção utilizaremos uma popular implementação da ordenação bitônica [1]. Este kernel é frequentemente utilizado como um módulo auxiliar em outras implementações CUDA. Em particular, a ordenação bitônica é parte da implementação do quicksort paralelo de Cederman e Tsigas [9]. Neste caso, a ordenação bitônica é invocada para ordenar os pequenos arranjos que o quicksort produz via a contínua aplicação do algoritmo de partição pivotal. A implementação da orde-

[1] `http://www.cs.chalmers.se/~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];
}
```

Figura 1.16. Implementação da ordenação bitônica.

nação bitônica, conforme obtida no pacote de desenvolvimento da Nvidia, é dada na figura 1.16.

Dado que a GPU está presa ao modelo de processamento SIMD, é provável que divergências venham a acontecer durante a execução da ordenação bitônica. Ou seja, invariavelmente a condição `(tid & k) == 0`, em que `tid` é o identificador da thread, será verdadeira para algumas threads e falsa para outras. A fim de observarmos o impacto de divergências sobre o desempenho da aplicação CUDA, faz-se necessário deixarmos o alto nível da linguagem fonte, para nos atermos aos detalhes do código PTX, cujo assembly é mostrado na figura 1.17.

A figura 1.17 usa uma notação que omitimos até agora: trata-se do *grafo de fluxo de controle* (CFG). Este CFG descreve as instruções PTX geradas para o bloco de código da figura 1.16 que existe no escopo do teste `ixj > tid`. O CFG possui um nodo para cada *bloco básico* do programa assembly, sendo um bloco básico uma sequência maximal de instruções sem desvios, sejam estes condicionais ou não. Os possíveis fluxos de execução entre os blocos básicos são descrito pelas arestas do

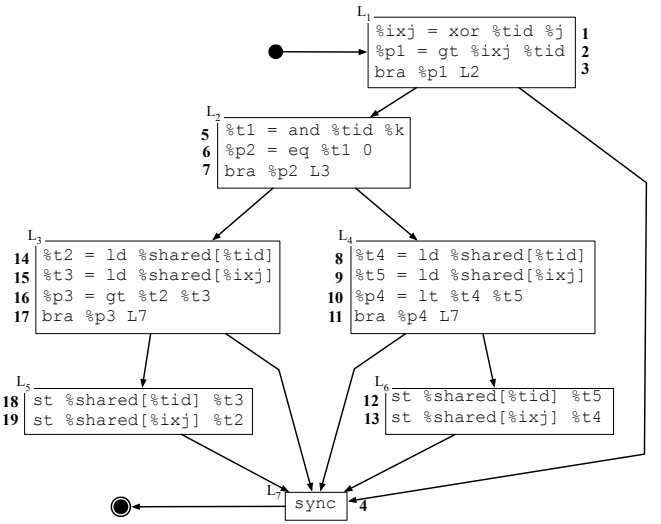

Figura 1.17. O grafo de fluxo de controle, em PTX simplificado, que representa o kernel da figura 1.16.

grafo. Esta descrição é estática: o programa pode nunca vir a seguir um determinado fluxo de execução, ainda que esta possibilidade exista, conforme descrito pelas arestas do CFG.

A figura 1.18 mostra um traço de execução do programa da figura 1.17, em um cenário em que temos quadro unidades funcionais e quatro threads por warp: t_0, t_1, t_2 e t_3. Supondo um vetor de entrada igual a $[4,3,2,1]$, quando $k = 2$ e $j = 1$, temos duas divergências. A primeira divisão acontece no ciclo $i = 3$, devido ao desvio `bra %p1, L2` e esta separa as threads t_0 e t_2 das threads t_1 e t_3. Esta divergência acontece porque a condição `%ixj > %tid` é verdadeira apenas para as threads t_0 e t_2. A segunda divisão acontece no ciclo $i = 6$, devido ao desvio `bra`

ciclo	rót.	Instrução	t_0	t_1	t_2	t_3
1	L_1	`xor %ixj %tid %j`	ι_1	ι_1	ι_1	ι_1
2		`gt %p1 %ixj %tid`	ι_2	ι_2	ι_2	ι_2
3		`bra %p1` L_2	ι_3	ι_3	ι_3	ι_3
4	L_2	`and %t1 %tid %k`	ι_5	•	ι_5	•
5		`eq %p2 %t1` 0	ι_6	•	ι_6	•
6		`bra %p2` L_3	ι_7	•	ι_7	•
7	L_3	`load %t2 %shared %tid`	ι_{14}	•	•	•
8		`load %t3 %shared %ixj`	ι_{15}	•	•	•
9		`gt %p3 %t2 %t3`	ι_{16}	•	•	•
10		`bra %p3` L_7	ι_{17}	•	•	•
11	L_4	`load %t4 %shared %tid`	•	•	ι_8	•
12		`load %t5 %shared %ixj`	•	•	ι_9	•
13		`lt %p4 %t4 %t5`	•	•	ι_{10}	•
14		`bra %p3` L_7	•	•	ι_{11}	•
15	L_5	`store %tid %t3`	ι_{18}	•	•	•
16		`store %tid %t2`	ι_{19}	•	•	•
17	L_6	`store %tid %t5`	•	•	ι_{12}	•
18		`store %tid %t4`	•	•	ι_{13}	•
19	L_7	`sync`	ι_4	ι_4	ι_4	ι_4

Figura 1.18. Uma curta trilha de execução do programa 1.17, mostrando as primeiras iterações deste código, supondo quatro threads por warps e $v = \{4,3,2,1\}$.

`%p2, L3` e ela separa as threads t_0 e t_2.

Medindo divergências via profiling. 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: profiling e análise estática. Profiling é um método dinâmico: o programa deve ser executado em um ambiente monitorado. A análise estática é justamente o contrário: a ferramenta de análise estuda o código do programa tentando provar quais desvios condicionais são passíveis de cau-

sar divergências e quais nunca o farão. Falaremos de profiling primeiro, deixando a análise estática para o final desta seção.

Existem pelo menos dois profilers que capturam o comportamento divergente de programas CUDA. Um deles é “profiler visual" da Nvidia (CVP) [2]. 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.* [10].

Usando informação de profiling para otimizar aplicações. Nesta seção mostraremos como obter mais desempenho de aplicações CUDA usando para isto o resultado da informação de profiling. Otimizaremos o algoritmo de ordenação bitônica mostrado na figura 1.16, porém mostraremos resultados obtidos para a implementação de quicksort de Cederman *et al.* [9]. Este algoritmo usa o kernel da figura 1.16 para ordenar arranjos de tamanho abaixo de um certo limiar. Os ganhos de desempenho que reportaremos nesta seção referem-se, portanto, ao algoritmo quicksort inteiro, embora nós mudaremos somente a ordenação bitônica, que é um sub-módulo daquele algoritmo. Veremos, assim, que é possível obter entre 6-10% de melhorias de desempenho modificando entre 10-12 instruções assembly de um programa de 895 instruções!

A figura 1.19 mostra resultados de profiling que obtemos via a ferramenta desenvolvida por Coutinho *et al.* [10] para três desvios condicionais da ordenação bitônica. Informação de profiling é dada como um quociente: *número de divergências / número de visitas*. Esta figura nos mostra que os condicionais aninhados no laço mais interno do kernel padecem de uma grande quantidade de divergências. Mais especificamente, a condição `(tid & k) == 0` é visitada por 28 milhões de warps e um terço destas visitas apresentam divergências. O mapa também mostra que as divergências são comuns em ambas as sub-

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

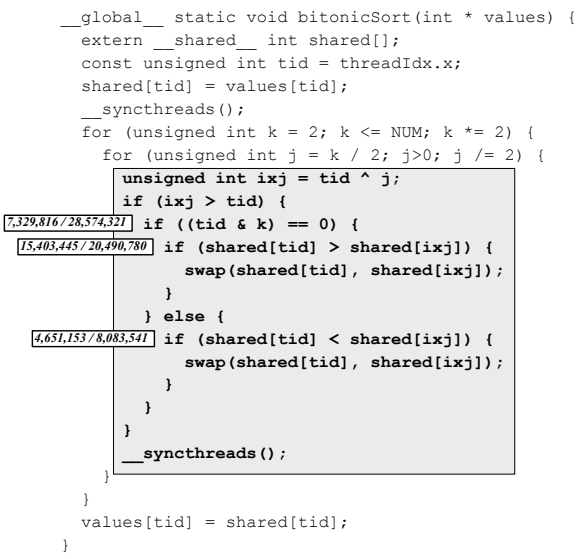

Figura 1.19. Informações produzidas pelo profiler usando a entrada padrão do algoritmo de ordenação quicksort de Cederman *et al.* [9].

cláusulas do bloco condicional, a saber, o teste `shared[tid] > shared[ixj]` e o teste `shared[tid] < shared[ixj]`.

A fim de melhorar este kernel, atentemos para o fato de que o caminho formado pelo bloco L_3 seguido pelo bloco L_5 é muito parecido com o caminho formado pelos blocos L_4 e L_6. A única diferença deve-se aos condicionais nos pontos de programa 9 e 13. De posse desta observação, costuramos os blocos L_5 e L_6 juntos, usando um pequeno truque de manipulação de índices, obtendo assim o programa da figura 1.20 (a). Divergências podem, obviamente, ainda acontecer, contudo, onde o maior cami-

(a)

```
unsigned int a, b;
if ((tid & k) == 0){
  b = tid;
  a = ixj;
} else {
  b = ixj;
  a = tid;
}
if (sh[b] > sh[a]){
  swap(sh[b],sh[a]);
}
```

(b)

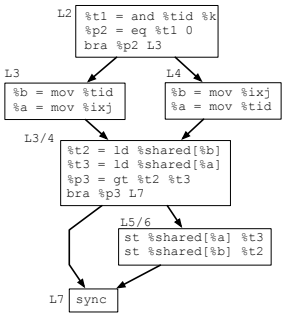

Figura 1.20. Programa otimizado após a junção dos blocos L_5 e L_6 da figura 1.17.

nho divergente da figura 1.17 contém oito instruções, o pior caso da figura 1.20 (b) contém apenas seis. Esta otimização dá-nos um ganho de desempenho de 6.75% em uma placa GTX 8800.

O código da figura 1.20 ainda nos fornece oportunidades de otimização. Podemos usar os operadores ternários disponíveis em CUDA para casar os blocos básicos L_3 e L_4, assim removendo a divergência ao final do bloco L_2. O programa fonte modificado é mostrado na figura 1.21 (a), e o código PTX equivalente é dado pela figura 1.21 (b). Uma instrução como `%a = sel %tid %ixj %p` atribui `%tid` a `%a` se `%p` for diferente de zero. O valor `%ixj` é usado caso contrário. Neste novo programa, o mais longo caminho divergente possui somente duas instruções. Esta última otimização dá-nos cerca de 9.2% de ganho de desempenho.

Análise estática de programas. Um profiler, ainda que muito útil, possui algumas desvantagens:

1. a informação produzida pelo profiler é dependente dos dados de entrada. Caso o programa instrumentado seja alimentado com dados que não causem qualquer divergência, então obteremos um histograma de divergências

(a)

```
int p = (tid & k) == 0;
unsigned b = p?tid:ixj;
unsigned a = p?ixj:tid;

if (sh[b] > sh[a]) {
  swap(sh[b], sh[a]);
}
```

(b)

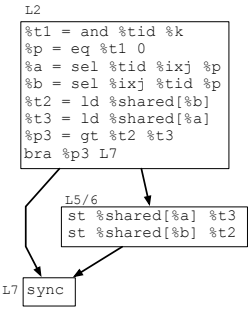

Figura 1.21. Código que resulta do uso de operadores ternários no programa visto na figura 1.20.

incondizente com a realidade do programa.

2. profilers baseados em instrumentação podem aumentar o tempo de execução do programa sob análise em até 1000x, conforme demonstrado anteriormente [10]!
3. um profiler não prova que uma divergência nunca acontecerá em um certo teste condicional.

O último item desta lista é particularmente problemático: a fim de implementar otimizações automaticamente, o compilador precisa de garantias concretas acerca do comportamento do programa fonte. Uma forma de contornar estas deficiências é via análise estática de programas. A análise estática inclui uma gama muito grande de técnicas que nos permitem inferir, a partir do texto to programa, informações acerca de seu funcionamento. Nesta seção voltaremos nossa atenção para um tipo de análise, o qual chamaremos de *análise de divergências*.

Faz-se necessário salientar que a análise estática apresenta desvantagens também. O maior problema é, provavelmente, o fato de que, a fim de fornecer resultados sempre corretos a aná-

lise estática precisa ser bastante conservadora. Em nosso contexto, é preciso marcar como divergente todos os testes condicionais que não podemos provar que são não divergentes. Ainda assim, estas duas técnicas: profiling e análise estática, complementam uma a outra, frequentemente fornecendo mais informação em conjunto que a soma de cada parte em separado. O profiler, por exemplo, é útil para calibrar a análise estática. Esta, por seu turno, permite reduzir a quantidade de instrumentação inserida nos programas, uma vez que não é necessário instrumentar desvios que nunca serão divergentes.

Dado um programa CUDA P, estamos interessados em determinar uma aproximação conservadora, porém não trivial, de seu conjunto de *variáveis divergentes*. Uma variável v é divergente se existem duas threads no mesmo warp tal que cada thread enxerga v com um valor diferente em um mesmo momento da execução do programa. Na próxima seção nós descreveremos uma análise estática que encontra um conjunto de variáveis divergentes. Esta análise é semelhante à inferência de barreiras de Aiken and Gay [2]; todavia, enquanto a inferência de barreiras supõe um modelo de execução SPMD, a análise de divergências supõe uma arquitetura SIMD. A fim de explicar como se dá a análise de divergências, usaremos alguns conceitos de teoria de compiladores, os quais são introduzidos a seguir.

Atribuição estática única. Compiladores frequentemente utilizam representações intermediárias para facilitar a tarefa de compreender automaticamente os programas. Uma das mais conhecidas representações intermediárias é o chamado formato de atribuição estática única, uma possível tradução da nomenclatura original – *Static Single Assignment* (SSA) [11]. Dada a popularidade da sigla SSA, nós a utilizaremos para referir-monos a esta representação intermediária. Programas neste formato possuem uma propriedade fundamental: cada variável é definida somente uma vez no texto do programa. Dada que esta propriedade simplifica enormemente a manipulação de programas, no que se segue iremos supor que todos os nossos códigos PTX estão neste formato.

A figura 1.22 ilustra a conversão de parte do código visto na figura 1.20 para o formato SSA. Esta conversão é feita se-

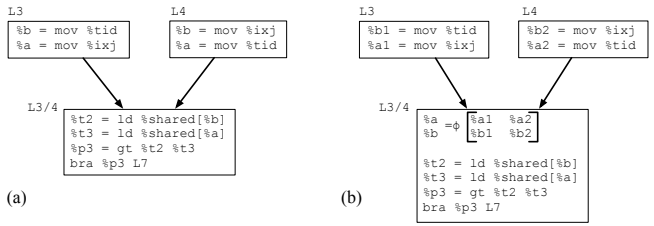

Figura 1.22. A representação de atribuição estática única (SSA). (a) programa original. (b) programa em formato SSA.

gundo duas ações: cria-se novos nomes para as diversas definições da mesma variável e inserem-se funções ϕ no texto do programa, a fim de mesclar as linhas de vidas destas variáveis em um mesmo nome. Por exemplo o nome de variável `a` possui duas definições na figura 1.22 (a); logo, este programa não está no formato SSA. A fim de fazer a conversão, renomeamos estas definições para `a1` e `a2`. Existe, entretanto, um novo problema: a variável `a` é usada no bloco básico $L_{3/4}$; fato este que nos impõe a questão de qual nome deveria ser usado naquele bloco: `a1` ou `a2`? A resposta, neste caso, é ambos os nomes. Se o fluxo de execução do programa alcança $L_{3/4}$ vindo de L_3, então o nome correto é `a1`, doutro modo o fluxo de execução pode somente provir de L_4 e o nome que precisa ser usado é `a2`.

Esta ligação entre a semântica dinâmica do programa e seu texto estático é feita por uma abstração denominada função ϕ. Estas instruções especiais funcionam como multiplexadores. A função ϕ presente no bloco $L_{3/4}$, isto é, `a = phi(a1, a2)` atribuirá à variável `a` ou o valor de `a1` ou o valor de `a2`, dependendo de onde vem a execução que alcança $L_{3/4}$. Funções ϕ são uma abstração notacional, não existindo realmente em programas assembly. Compiladores que utilizam o formato SSA requerem um passo adicional, que consiste na implementação destas ins-

truções usando-se instruções concretas, normalmente cópias, logo antes da geração de código de máquina.

Além do formato SSA, nós lançaremos mão também do conceito de pós-dominância. Vamos supor que cada instrução em um programa PTX é representada por um rótulo l. Um rótulo l_p pós-domina outro rótulo l se e somente se, todo caminho de l até a saída do programa passa necessariamente por l_p. Mais ainda, dizemos que l_p é o *pós-dominador imediato* de l se $l_p \neq l$ e qualquer outro rótulo que pós-domina l também pós-domina l_p. Fung *et al.* [13] mostraram que o melhor ponto para reconvergir threads divergentes é o pós-dominador do desvio que causou as divergências.

O conceito de pós-dominância usa a noção de caminho de programa, que definimos da seguinte forma: Dado um programa P, nós dizemos que um rótulo l vai para um rótulo l', o que indicamos por $l \rightarrow l'$, se uma das três condições é verdadeira:

- l' é a próxima instrução após l e l não rotula um desvio incondicional;
- l é um desvio condicional sendo l' um de seus alvos;
- l é um desvio incondicional cujo destino é l'.

Dizemos que $l_1 \xrightarrow{*} l_n$ se $l_1 = l_n$, ou $l_1 \rightarrow l_2$ e $l_2 \xrightarrow{*} l_n$. Se l_p é o pós-dominador imediato de l, então nós definimos a região de influência de l, que denotamos por $IR(l)$, como a união de todos os caminhos $l \xrightarrow{*} l_p$ que contêm l_p exatamente uma vez – e por conseguinte ao final. Dizemos que uma variável v pertencente a um programa P em formato SSA *alcança* um rótulo l se P contêm dois rótulos l_d e l_u tais que:

- v é definida por uma instrução em l_d;
- v é usada por uma instrução em l_u;
- P contém um caminho $l_d \xrightarrow{*} l$;
- P contém um caminho $l \xrightarrow{*} l_u$.

Usaremos o programa da figura 1.23 para ilustrar os conceitos expostos nesta seção. O pós-dominador imediato da instrução `branch %p0` B_2, ao final de B_1, é a instrução `sync` no início

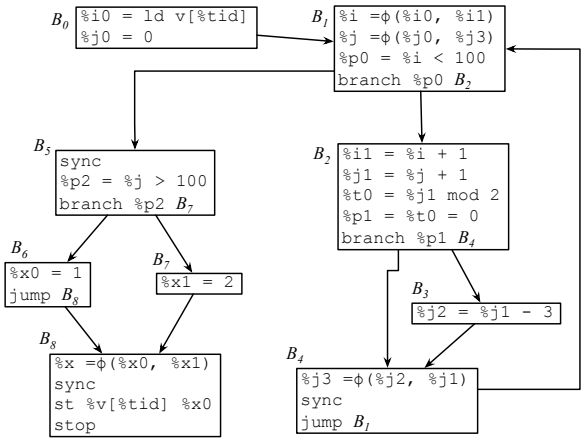

Figura 1.23. Programa exemplo para ilustrar a análise de divergências.

de B_5. A região de influência deste desvio condicional compraz as instruções nos blocos básicos B_1, B_2, B_3 e B_4.

A busca pelas variáveis divergentes. Dados os conceitos sobre os quais falamos previamente, o Teorema 1.3.1, cuja prova está disponível on-line [3], nos dá os subsídios necessários para encontrar as variáveis divergentes em um programa.

Teorema 1.3.1 *(*VARIÁVEL DIVERGENTE*) Uma variável $v \in P$ é divergente se e somente se, uma das condições abaixo for verdadeira:*

1. *$v =$ `tid` ou qualquer outra variável que possui um valor pré-definido e particular para cada thread, como por exemplo `laneId`, a posição da thread no warp.*

[3] `http://divmap.wordpress.com`

2. *v é definida por uma instrução atômica do tipo leia-modifique-escreva, como por exemplo* `atomic { v = *a; *a = *a + b }`.
3. *v possui uma dependência de dados de alguma variável divergente.*
4. *v possui uma dependência de sincronização de alguma variável divergente.*

Antes de adentrar os pormenores de dependências de dados e de sincronização, procuremos entender os dois primeiros itens do Teorema 1.3.1. A variável `tid` é divergente por definição, uma vez que ela contém um valor diferente para cada thread. Já uma incremento atômico como `atomic { v = *a; *a = *a + 1 }` é executado exclusivamente por cada thread. Esta serialização faz com que cada thread obtenha um valor diferente para v, pois todas elas lêem – e alteram – o mesmo endereço de memória e esta leitura não é simultânea.

Dado um programa P, a variável $v \in P$ possui uma *dependência de dados* com relação à variável $u \in P$ se P contém uma instrução que cria v e usa u. Por exemplo, na figura 1.23 nós temos que `%i` depende de `%i1`, devido à função ϕ no bloco B_1. Temos também que `%i1` depende de `%i`, devido à atribuição `%i1 = %i + 1` no bloco B_2. O problema de determinar o fecho transitivo das dependências de dados é um tipo de *fatiamento de programas* [27] e ele pode ser resolvido por uma simples busca em grafo. Este grafo, convenientemente chamado grafo de dependência de dados, é definido 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 adicionamos uma aresta entre n_u e n_v.

A fim de encontrar o conjunto de variáveis divergentes em P, nós começamos a travessia do grafo de dependências a partir de n_{tid}, mais os vértices que representam variáveis definidas por instruções atômicas. Marcamos então todos os outros nodos alcançáveis a partir deste conjunto inicial como divergentes. Existem, contudo, variáveis divergentes que não seriam capturadas

por este algoritmo. Tratam-se das *dependências de sincronização*, que definimos abaixo:

Definição 1.3.2 *Dada uma instrução* `branch %p` B*, dizemos que* v *possui uma dependência de sincronização de* `%p` *se e somente se, o valor de* v *no pós-dominador imediato deste desvio condicional depende do resultado do predicado* `%p`*.*

Para ilustrar este novo conceito, retornemos à figura 1.23. A variável `%x` é atribuída por uma função ϕ, cujo resultado depende de qual caminho o fluxo de execução usa para alcançar o bloco B_8. Este caminho, por sua vez, depende do resultado de `branch %p2` B_7. Dizemos, assim, que `%x` possui uma dependência de sincronização com relação ao predicado `%p2`. Existe um algoritmo para encontrar tais dependências que usa o seguinte teorema provado por Aiken e Gay [2]: "Se `branch %p` B é um desvio condicional e l_p é seu pós-dominador imediato, então a variável v é dependente por sincronização de p se e somente se, v é definida por uma instrução localizada dentro da região de influência do desvio condicional e v alcança l_p."

É possível transformar todas as dependências de sincronização em dependências de dados. Para tanto utilizamos uma antiga e até certo ponto obscura, representação intermediária chamada *formato SSA chaveado*, (tradução livre de *Gated SSA form* – GSA) [21]. Neste formato algumas funções ϕ são aumentadas com predicados. Não mostraremos este resultado aqui, mas somente variáveis definidas por funções ϕ possuem dependências de sincronização. Logo, aumentando estas funções ϕ com os predicados que as controlam, estamos efetivamente adicionando uma dependência de dados entre a variável definida pela função ϕ e estes predicados. Dizemos que uma função ϕ aumentada com um predicado p é *chaveada* por p. Por exemplo, abaixo temos uma função ϕ chaveada pelos predicados p_1 e p_2. Usando a nossa noção de dependências de dados, pode-se dizer que v é dependente destes dois predicados.

$$v = \phi(v_1, \ldots, v_n), p_1, p_2$$

O algoritmo na figura 1.24 converte um programa em formato SSA para um programa no formato GSA. Para um algo-

Algoritmo 1.24: chaveamento de programas em formato SSA – para cada instrução `branch` p B cujo pós-dominador imediato é l_p faça:

1. para cada variável v, definida em $IR(p)$ e que alcança l_p faça:
   (a) se v é usada em l_p como o parâmetro de uma função ϕ, isto é, $v' = \phi(\ldots, v, \ldots)$, chaveada ou não, então substitua esta instrução por uma nova função ϕ chaveada por p;
   (b) se v é usada em uma instrução de atribuição $x = f(\ldots, v, \ldots)$, em l_p ou em algum rótulo l_x que é dominado por l_p, então faça:
      i. divida a linha de vida de v, inserindo uma nova instrução $v' = \phi(v, \ldots, v), p$ em l_p, com um parâmetro para cada predecessor de l_p;
      ii. renomeie cada cópia de v para v' em l_p, ou em qualquer rótulo dominado por l_p;
      iii. reconverta P para o formato SSA. Esta ação é necessária devido à nova definição de v.

Figura 1.24. Algoritmo que converte um programa em formato SSA em um programa em formato GSA.

ritmo mais eficiente, recomendamos o trabalho de Tu e Padua [26]. A figura 1.25 mostra o resultado de executar o algoritmo da figura 1.24 no programa da figura 1.23. As funções ϕ em B_4 e B_8 foram marcadas no passo (1.a) de nosso algoritmo. As variáveis `%j3` e `%x` são dependentes, por sincronização, das variáveis `%p1` e `%p2`, respectivamente. A função ϕ de um parâmetro em B_5 foi criada pelo passo (1.b.i) do Algoritmo 1.24.

Tendo chaveado as funções ϕ, as dependências de dados passam a incorporar também as dependências de sincronização. Isto é, ao criar uma instrução como $v = \phi(v_1, \ldots, v_n), p_1, \ldots, p_k$, nós estamos afirmando que v possui dependências de dados não somente devido às variáveis $v_i, 1 \leq i \leq n$, mas também devido aos predicados $p_j, 1 \leq j \leq k$. Dando sequência ao nosso exemplo, a figura 1.26 mostra o grafo de dependências criado para o programa chaveado da figura 1.25.

Surpreendentemente, notamos que a instrução `branch %p1` B_4 não pode causar uma divergência, ainda que o predicado `%p1` possua uma dependência de dados pela variável `%j1`, que

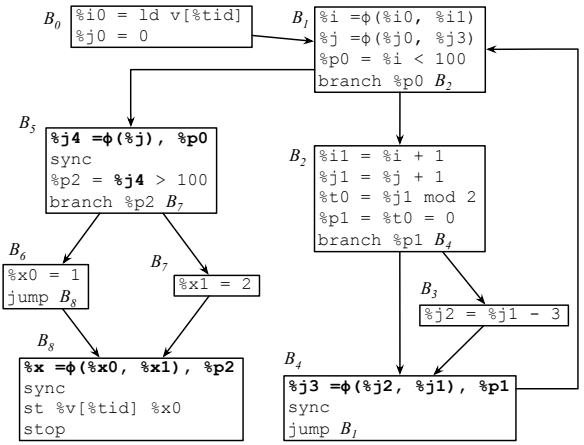

Figura 1.25. Resultado de aplicar o algoritmo 1.24 ao programa da Figura 1.23.

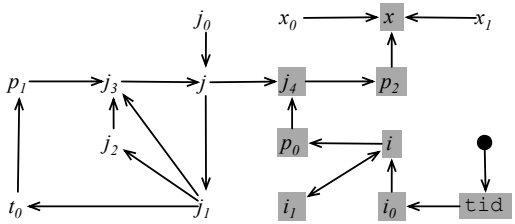

Figura 1.26. O grafo de dependências criado para o programa chaveado visto na figura 1.25. Variáveis divergentes estão coloridas com cinza.

por sua vez é criada dentro de um laço controlado por uma variável divergente. Isto é, a variável `%j1` não é divergente, embora o seja a variável `%p0` que controla o laço em que `%j1` é criada. A variável `%j4`, por outro lado, criada pelo passo (1.b.i) do algoritmo 1.24, é divergente, pois ela depende deste mesmo predicado `%p0`. O valor de `%j4`, na verdade, depende de quantas vezes as diferentes threads irão iterar sobre o laço controlado por `%p0`. Ainda que este fato não possa causar uma divergência dentro do laço, conforme mostra o Algoritmo 1.24, a possibilidade de diferentes threads iterando sobre o laço quantidades diferentes de vezes pode levar a divergências fora dele. Por exemplo, devido à natureza divergente de `%j4`, o desvio condicional `branch, %p2`, B_7, é também divergente, isto é, threads que visitam este desvio podem tomar caminhos diferentes.

Otimizações de caminhos divergentes. Existem algumas técnicas de otimização de código que usam as informações produzidas pela análise de divergência para melhorar o desempenho de programas em face às divergências. Descreveremos em maiores detalhes três destas otimizações: otimizações peephole, compartilhamento de variáveis e fusão de caminhos divergentes. Existe uma quarta otimização, bastante conhecida, que omitiremos. Trata-se da realocação de threads. Em presença de divergências, pode-se reagrupar as threads entre diferentes warps, de modo que a maior parte dos warps contenha threads que tomem decisões similares. Zhang *et al.* [28] implementaram a realocação de threads manualmente, obtendo ganhos de desempenho de até 37% em algumas aplicações CUDA. De forma similar, Fung *et al.* [13] propuseram uma nova forma de realizar a realocação de threads ao nível de hardware, inferindo, via simulação, ganhos de até 20% de desempenho.

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 assembly 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 [4]:

> "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 ser aumentadas com o prefixo `uni` ou que possuem implementação mais eficiente para código não divergente:

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

call.uni uma chamada de função não divergente. Chamadas de função podem ser acrescidas de um predicado, sendo a função chamada somente para aquelas threads cujo predicado seja verdadeiro. Esta instrução supõe 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, esta variável

[4] PTX programmer's manual, 2008-10-17, SP-03483-001_v1.3, ISA 1.3

```
__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 1.27. Um kernel CUDA em que a pressão de registradores é 11, levando à um terço da ocupação de threads ativas.

pode ser alocada na memória compartilhada da GPU. 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. Recordando a discussão acerca de alocação de registradores na Seção 1.2, 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.

Explicaremos o compartilhamento de dados via o e exemplo da figura 1.27, que, embora artificial, têm a vantagem de ilustrar em poucas linhas nossas idéias. Este kernel 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

```
__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 1.28. Compartilhamento interno de variáveis. Este código é 10% mais eficiente que o kernel da figura 1.27 (GPU 9400M).

`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 1.28. 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 corretude do programa; porém, tal controle diminui também a utilização do barramento de memória, aumentando a eficiência deste kernel.

Fusão de caminhos divergentes. Desvios condicionais do tipo "if-then-else" podem ter muitas instruções em comum. A otimização de código chamada fusão de caminhos divergentes (tradução livre de *branch fusion*) consiste em mover estas instruções comuns para caminhos compartilhados do grafo de fluxo de controle. A fusão de caminhos divergentes é um tipo mais ex-

tensivo de eliminação parcial de redundâncias [19]. Usaremos o exemplo da figura 1.29 para explicar esta otimização. Podemos ver que as sequências de instruções que começam nos rótulos l_4 e l_{13} possuem muitas operações – e alguns operandos – em comum. Representemos uma instrução *store* por $\uparrow$ e uma instrução *load* por $\downarrow$. Podemos assim representar as duas sequências de instruções nos caminhos divergentes da figura 1.29 (a) por $T = \{\bot, \downarrow, *, *, *, /, /, *, +, \uparrow\}$ e $F = \{\bot, \downarrow, *, *, /, *, *, +, \uparrow\}$. Este exemplo levanta três questões, as quais enunciamos abaixo.

1. Seria vantajoso unificar partes destes caminhos divergentes em termos de tempo de execução?
2. Quais os trechos de código que mais benefício trariam em caso destes serem unificados? Caso estejamos apenas procurando as sequências mais longas, então a figura 1.29 (b) mostra uma possível resposta.
3. Existe alguma forma sistemática de unificarmos estes caminhos, produzindo assim um programa equivalente ao código original? Em nosso exemplo, a figura 1.29 (c) mostra uma possível versão do programa unificado. Usamos operadores ternários (`sel`) para escolher os operandos fontes de instruções unificadas. Em arquiteturas em que tais operadores não existam, pode-se recorrer a conversão de desvios (*if-conversion*) [15, 23].

No restante desta seção tentaremos responder a cada uma destas questões.

Smith-Waterman. Atendo-nos à primeira questão levantada, precisamos encontrar quais os trechos de código cuja unificação maior benefício trariam ao desempenho do programa otimizado. Em geral existem muitos diferentes programas que realizam a mesma computação, e encontrar a mais eficiente dentre tantas versões é um problema muito difícil. Por exemplo, se pudéssemos re-ordenar as instruções em um caminho divergente talvez fosse possível produzir uma sequência de unificações mais eficiente. Todavia, o espaço de buscas neste caso é enorme. Iremos supor então que não podemos transformar o programa para obter melhores unificações. Em vez disto, trabalharemos sobre um

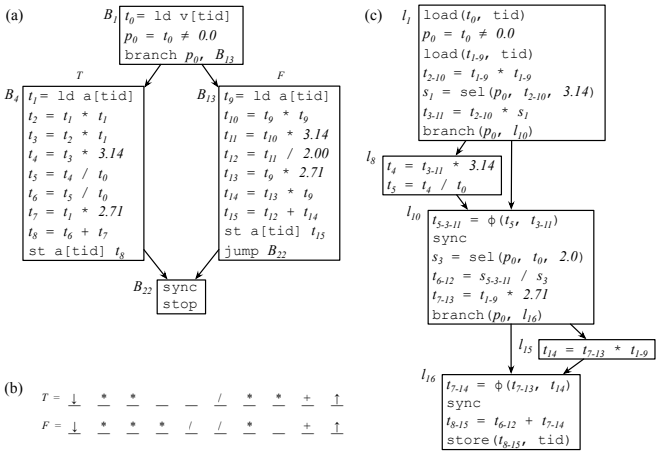

Figura 1.29. (a) Exemplo que será utilizado para explicar a fusão de caminhos divergentes. (b) Um possível alinhamento de instruções. (c) Código após a fusão.

problema mais simples, chamado *alinhamento bi-dimensional de instruções*, o qual definimos abaixo:

Definição 1.3.3 *(*ALINHAMENTO 2D DE INSTRUÇÕES*)*
***Instância**: dados os seguintes parâmetros:*

- *dois arranjos de instruções,* $T = \{i_1, \ldots, i_n\}$ *e* $F = \{j_1, \ldots, j_m\}$*;*
- *uma função de lucro* s*, tal que* $s(i, j)$ *seja o lucro de unificar as instruções* i *e* j*;*
- *uma função de custo* c*, tal que* $c(i, j)$ *seja o custo de unificar as instruções* i *e* j*;*
- b*, o custo constante que advêm de quebras entre sequências unificadas. Isto é,* b *deve ser pago sempre que houver discontinuidades entre trechos de código a serem uni-*

ficados. Em geral b é o custo de se inserir desvios condicionais dentro das sequências unificadas.

***Problema**: encontre 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 $y_r \geq y_s$*
- *$\sum(s(x,y) - c(x,y)) - b \times G$ é máximo, sendo $(x,y) \in A$, e G o número de descontinuidades nas sequências unificadas.*

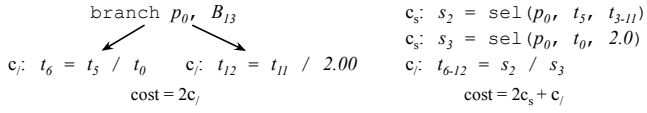

Figura 1.30. Comparação entre o custo de unificação e execução divergente.

Existe um lucro e um custo, em termos de ciclos de execução, para unificar um par de instruções, o que normalmente derivamos do manual do programador. Por exemplo, se uma instrução de *load* custa 100 ciclos, então o casamento de duas delas nos economizaria este custo em um caminho divergente. Por outro lado, a fim de efetuar tal unificação pode ser necessário que tenhamos de inserir seletores no programa, os quais adicionam tempo de processamento ao código modificado. A figura 1.30 ilustra as operações necessárias para unificar duas instruções de divisão localizadas em caminhos divergentes do programa visto na figura 1.29 (a). À esquerda temos parte do programa original e à direita temos o programa após a fusão. O custo que acabamos de mencionar inclui a seleção dos parâmetros da nova instrução ($2c_s$) e a execução da operação de divisão ($c_/$). Quando o custo dos seletores é menor que o lucro da unificação, dizemos que tal transformação é *lucrativa*. No exemplo

da figura 1.30, temos um lucro $s = 2c_{/} - 2c_s - c_{/} = c_{/} - 2c_s$. Observe que se um dos operandos nas duas instruções unificadas for o mesmo, então este operando não demanda um seletor e seu custo é abatido do lucro da unificação.

Podemos casar instruções que não são necessariamente as mesmas. Por exemplo, é possível unificar uma adição e uma multiplicação em arquiteturas que provêem uma instrução do tipo *multiply-add*, que perfaz ambas as operações. De modo semelhante, podemos unificar instruções de comparação tais como “maior-que" e “menor-que" via transformações simples de seus operandos. Na verdade, é possível levar este abuso de identidades entre instruções ao extremo, usando técnicas mais avançadas, como a saturação de igualdades [25], que normaliza todas as instruções em um trecho de código.

O problema de encontrar sequências similares de instruções é muito parecido ao problema de sequenciamento genético: dadas duas sequências de genes, pede-se pelas sequências mais longas de genes em comum. Tal problema, já muito estudado em biologia, possui uma elegante solução: o algoritmo de sequencialmento de Smith e Waterman [24]. Nós utilizaremos este mesmo algoritmo para determinar as sequências mais lucrativas de unificações. Este algoritmo possui dois passos: primeiro, constrói-se uma matriz de ganhos que mostra o lucro de unificar cada possível par de instruções. Segundo, encontra-se nesta matriz a sequência de unificações mais lucrativa.

A fim de construir a matriz de ganhos H nós escrevemos uma das sequências de instruções ao longo da linha superior da matriz e escrevemos a outra ao longo de sua coluna mais à esquerda. Cada posição da matriz é associada a um valor g, isto é: $H[i, j] = g$, onde g é o máximo lucro obtido a partir de qualquer forma possível de unificar instruções até os índices i e j. Para calcular $H[i, j]$ nós usamos a fórmula abaixo, sendo o significado dos parâmetros s, c e b os mesmos da Definição 1.3.3:

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

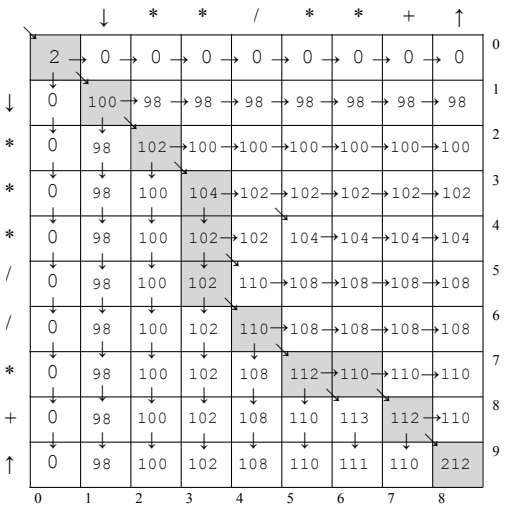

Figura 1.31. A matriz de ganhos produzida para o programa da figura 1.29 (a).

Continuando com nosso exemplo, a figura 1.31 mostra a matriz de ganhos construída para o programa da figura 1.29 (a). Estamos usando a seguinte função de lucratividade: $s(\downarrow,\downarrow) = s(\uparrow,\uparrow) = 100$, $s(*,*) = 2$, $s(/,/) = 8$, $s(+,+) = 2$. Supomos o custo de descontinuidade $b = 2$. Note que este custo é pago somente uma vez por descontinuidade, quando deixamos uma sequência diagonal de células mais lucrativas e caminhamos para uma posição vertical ou horizontal na matriz. Por simplicidade estamos fazendo $c = 0$.

Tendo construído a matriz de ganhos, encontramos uma solução para o problema do alinhamento de instruções em dois passos. Primeiro visitamos as células da matriz para encontrar a posição $H[x,y]$ com o maior lucro. Depois atravessamos a ma-

triz, começando por $H[x,y]$ e indo em direção à posição $H[0,0]$, seguindo o sentido em que cada célula foi atualizada durante a construção da matriz. Isto é, se $H[i,j]$ foi atualizada a partir de $H[i-1,j-1]$, então nós continuamos nosso passeio a partir desta última célula. Em nosso exemplo, a célula de maior ganho é $H[9,8]$ e a sequência mais lucrativa é $A = \langle(1,1),(2,2),(3,3),(4,3),(5,4),(6,4),(7,5),(7,6),(8,7),(9,8)\rangle$.

Na figura 1.31 nós aumentamos cada célula da matriz de ganho com a direção de sua atualização. As posições coloridas em cinza marcam o caminho mais lucrativo dentro desta matriz. Este caminho denota exatamente o alinhamento visto na figura 1.29 (c). A partir de caminhos lucrativos em diagonais nós criamos blocos básicos de instruções unificadas. Os caminhos horizontais ou verticais representam caminhos divergentes. Damos às atualizações horizontais e verticais preferência sobre atualizações diagonais. Em outras palavras, quando o ganho de unificar duas instruções for o mesmo de manter estas instruções em caminhos divergentes, escolhemos a segunda opção. Desta forma tendemos a diminuir o número de descontinuidades na sequência alinhada, evitando a inserção de desvios condicionais no código alvo. Por exemplo, na figura 1.31, poderíamos ter atualizado $H[2,4]$ a partir de $H[1,3]$. Entretanto, a unificação das duas próximas instruções – multiplicação e divisão – possui um lucro muito pequeno. Assim, seríamos forçados a inserir um desvio condicional no código. Por outro lado, dado que a atualização de $H[2,4]$ deu-se por um caminho vertical, o custo da inserção deste desvio já foi pago anteriormente.

1.4. O Caldeirão no Final do Arco-Íris

A otimização de código não é uma área nova de pesquisa. Ao contrário, desde os primeiros passos neste campo, dados por pioneiros como John Backus [6], muito já foi feito, tanto pela indústria quanto pela academia. Ainda assim, as GPUs, com seu modelo de execução ainda pouco conhecido, trazem novos desafios para os projetistas de compiladores e para os desenvolvedores de aplicações de alto desempenho. Existem muitas maneiras de manter-mo-nos atualizados sobre esta rápida evolução tecnológica. Conferências e simpósios, por exemplo,

são fonte de informação atual e valiosa sobre este assunto. Entre os vários anais importantes, vale citar PLDI, POPL, PPoPP, PACT, MICRO, CGO e CC. Além das conferências existem muitas listas de discussão, revistas eletrônicas e blogs que discorrem acerca de GPUs e CUDA e muitos internautas sentir-se-ão felizes em poder sanar dúvidas de outrem. A comunidade de software livre, em particular, parece lançar-se com entusiasmo à tarefa de desenvolver novas ferramentas e técnicas para o desenvolvimento de aplicações em placas gráficas. Geradores de código PTX, por exemplo, já existem em compiladores como Ocelot [12], Open64 e LLVM [17]. Certamente será muito interessante descobrir onde esta nova tecnologia nos levará. Como tudo em Ciência da Computação, o limite de nossos avanços é a nossa própria criatividade!

Referências

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[2] Alexander Aiken and David Gay. Barrier inference. In *POPL*, pages 342–354. ACM Press, 1998.

[3] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.

[4] Vários autores. *NVIDIA CUDA Compute Unfied Device Architecture – Programming Guide*. NVIDIA, 1.0 edition, 2007.

[5] Vários autores. *NVIDIA CUDA C Programming Best Practices Guide – CUDA Toolkit 2.3*. NVIDIA, 1.0 edition, 2009.

[6] John Backus. The history of fortran i, ii, and iii. *SIGPLAN Not.*, 13(8):165–180, 1978.

[7] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for gpu architectures. In *PPoPP*, pages 105–114. ACM, 2010.

[8] Kenneth. E. Batcher. Sorting networks and their applications. In *AFIPS*, pages 307–314. ACM, 1968.

[9] Daniel Cederman and Philippas Tsigas. GPU-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics*, 14(1):4–24, 2009.

[10] Bruno Coutinho, Diogo Sampaio, Fernando Magno Quint ao Pereira, and Wagner Meira Jr. Performance debugging of GPGPU applications with the divergence map. In *SBAC-PAD*, pages 33 – 44. IEEE, 2010.

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *POPL*, pages 25–35, 1989.

[12] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot, a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *PACT*, pages 354–364, 2010.

[13] Wilson Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO*, pages 407–420. IEEE, 2007.

[14] Mark Harris. The parallel prefix sum (scan) with CUDA. Technical Report Initial release on February 14, 2007, NVIDIA, 2008.

[15] Ken Kennedy and Kathryn S. McKinley. Loop distribution with arbitrary control flow. In *Supercomputing*, pages 407–416. IEEE, 1990.

[16] M. S Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS*, pages 63–74. ACM, 1991.

[17] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[18] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA*, pages 451–460. ACM, 2010.

[19] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, 1979.

[20] John Nickolls and David Kirk. *Graphics and Computing GPUs. Computer Organization and Design, (Patterson and Hennessy)*, chapter A, pages A.1 – A.77. Elsevier, 4th edition, 2009.

[21] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI*, pages 257–271. ACM, 1990.

[22] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP*, pages 73–82. ACM, 2008.

[23] Jaewook Shin. Introducing control flow into vectorized code. In *PACT*, pages 280–291. IEEE, 2007.

[24] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[25] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *POPL*, pages 264–276. ACM, 2009.

[26] Peng Tu and David Padua. Efficient building and placing of gating functions. In *PLDI*, pages 47–55. ACM, 1995.

[27] Mark Weiser. Program slicing. In *ICSE*, pages 439–449. IEEE, 1981.

[28] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In *ICS*, pages 115–126. ACM, 2010.