Paralelização Automática de código com diretivas OpenACC

Kézia C. Andrade Moreira, Gleison S. Diniz Mendonça, Breno Campos Ferreira Guimarães, Péricles Rafael Oliveira Alves e Fernando Magno Quintão Pereira

UFMG – Avenida Antônio Carlos, 6627, 31.270-010, Belo Horizonte, MG, Brazil
{kezia.andrade,gleison.mendonca,brenosfg,periclesrafael,fernando}@dcc.ufmg.br

Resumo Entre as técnicas emergentes para paralelização de código, sistemas de anotação se destacam por sua simplicidade e eficiência. OpenACC é um dos exemplos mais proeminentes de tais sistemas: o programador insere diretivas em uma aplicação C/C++ sequencial e um compilador compatível traduz o código anotado para um formato binário. Este é então executado parte em uma CPU comum, parte em uma Unidade de Processamento Gráfico. Apesar de parecer simples a princípio, sistemas de anotação ainda deixam ao programador tarefas como decidir se uma região de código deve ou não ser paralelizada e, caso positivo, definir como dados devem ser dispostos no hardware paralelo. Neste artigo, apresentamos um conjunto de análises estáticas que auxiliam o programador nestas tarefas. Nossas análises descobrem, sem qualquer intervenção do usuário, (i) quais laços devem ser paralelizados e (ii) quais dados devem ser transferidos entre CPU e GPU. A fim de demonstrar a eficácia das técnicas que propomos, nós as implementamos sobre LLVM e as utilizamos na paralelização automática de benchmarks conhecidos, tais como Polybench, obtendo ganhos de até 121x em tempo de execução. Este resultado se mostra ainda mais significativo ao considerarmos que nossa abordagem não prevê qualquer tipo de encargo ao programador.

1 Introdução

Sistemas de anotação de código voltados para o desenvolvimento de aplicações paralelas em arquiteturas heterogêneas têm sido amplamente utilizados na comunidade científica e na indústria [21]. Esses sistemas foram criados com o intuito de aumentar a produtividade de programadores: eles podem escrever código sem preocupar-se com a sintaxe para criação de *threads* ou a cópia de dados entre diferentes processadores. Ainda assim, eles conseguem obter a partir de programas anotados todos os benefícios do *hardware* paralelo. Exemplos de sistemas de anotação que permitem a colocação de computação em GPUs incluem OpenMP 4.0 [13], OpenSs [20] e OpenACC [25], sendo este último um dos membros mais proeminentes desta família. De forma resumida, OpenACC permite ao programador indicar quais laços devem ser executados em *hardware* paralelo, bem como quais dados devem ser movidos para tal *hardware* a fim de servir como entrada

para as operações a serem executadas. OpenACC é portátil, podendo ser utilizado em plataformas como CPUs *multi-core e* GPUs, e proporciona ganhos significativos de desempenho [27].

Apesar de se mostrar mais simples quando comparada a outras técnicas para paralelização de código, a inserção manual de anotações ainda é uma tarefa difícil e propensa a erros [3]. Do ponto de vista de corretude, é deixada ao programador a tarefa de identificar quais laços são paralelizáveis. Laços são ditos paralelizáveis quando eles não apresentam condições de corrida, uma situação que ocorre quando duas ou mais *threads* usam a mesma posição de memória, e pelo menos um desses usos é uma escrita [12,28]. Em seguida, o desenvolvedor deve avaliar o quão vantajoso seria enviar um determinado laço à GPU: o custo de mover para a GPU os dados necessários à computação somado a um número potencialmente alto de divergências no corpo do laço pode provocar um aumento considerável do tempo de execução de um programa [5]. É fácil perceber então que a complexidade destas tarefas cresce juntamente com a complexidade do código da aplicação. Este trabalho visa automatizar completamente estes passos, facilitando assim a tarefa de escrever código paralelo.

A fim de tornar automática a anotação de laços paralelos, este trabalho provê um conjunto de análises estáticas de código que realiza duas tarefas. A primeira dessas tarefas é inserir anotações para copiar dados entre a CPU e a GPU. A segunda dessas tarefas é estimar quais laços, uma vez marcados como paralelos, possuem maior probabilidade de levar a ganhos de desempenho. Essas tarefas são realizadas sem qualquer intervenção do usuário. As análises estáticas utilizadas para resolver as duas tarefas são detalhadas na Seção 3. A plataforma que ora é apresentada foi implementada sobre dois compiladores. As análises estáticas d Seção 3 foram feitas sobre a infra-estrutura de compilação disponível em LLVM [15]. A geração de código paralelo a partir de programas anotados é feita por PGCC [9]. LLVM é uma infra-estrutura de compilação amplamente utilizada na indústria, possuindo *front-ends* para diversas linguagens, dentre elas C e C++. PGCC, por sua vez, é um compilador fonte-a-fonte, de código fechado, que traduz programas C aumentados com diretivas OpenACC para código escrito em C para CUDA.

O paralelizador de código proposto neste artigo está hoje disponível publicamente via uma interface web: `http://cuda.dcc.ufmg.br/dawn/`. O processo de paralelização é totalmente estático e automático: decidimos quais os dados serão enviados para a GPU e inserimos as diretivas em tempo de compilação e sem nenhuma intervenção do usuário. A validação de sua efetividade deu-se por meio de experimentos com Polybench, um conjunto de *benchmarks* popular na comunidade de computação de alto desempenho. Os experimentos da Seção 4 mostram que nossa ferramenta foi capaz de identificar corretamente quais os dados deveriam ser enviados à GPU e o compilador PGCC paralelizou todos os laços que não possuem dependências entre iterações diferentes. Nesses experimentos conseguimos paralelizar automaticamente 95% dos laços encontrados. Como consequência, observamos *speedups* de até 121x. Resultado este significativo, visto que o uso da ferramenta não prevê encargos ao programador.

2 Visão Geral

Esta seção descreve a linguagem de anotações OpenACC, detalhando a especificação daquelas pragmas da linguagem que são usadas neste trabalho. Em seguida, ilustra-se o mecanismo de paralelização proposto via um exemplo.

2.1 A linguagem de anotação OpenACC

Neste trabalho, utilizamos um sistema de anotações para informar ao compilador em qual unidade de processamento (CPU/GPU) cada laço de um programa deve ser executado. O sistema adotado em nossa implementação foi OpenACC [25]. Existem vários outros sistemas de anotação similares ao OpenACC como, por exemplo, OpenMP [13], criada em 1997 e talvez a mais conhecida dentre tais meta-linguagens. OpenMP 4.0 possui, atualmente, um conjunto de diretivas, rotinas e variáveis de ambiente que fornecem ao compilador informação suficiente para transformar código C para código CUDA. Um terceiro sistema de anotações é OpenSs [20], produto de pesquisa realizada no Centro de Computação de Barcelona.

Este trabalho não definiu um sistema de anotações novo, apenas utilizamos um sistema já disponível. O OpenACC foi usado para indicar como cada computação deveria ser distribuída. A escolha do OpenACC foi pragmática: o autor deste trabalho possui familiaridade com PGCC, um compilador que lê diretivas naquele sistema. Além disso, OpenACC, sendo o mais antigo dentre os sistemas de anotações para aceleração de código – tendo surgido em 2011 – já possui maior suporte por parte da indústria. Tanto OpenMP 4.0 quanto OpenSs poderiam prestar-se aos mesmos propósitos que o OpenACC para este trabalho.

Vários compiladores traduzem código anotado para CUDA-*Compute Unified Device Architecture* [7]. Neste trabalho foi utilizado PGCC, um compilador fechado desenvolvido pelo *Portland Group* (PGI). Existem, contudo alternativas similares hoje publicamente disponíveis. O OpenARC [18] é exemplo de um compilador OpenACC aberto. Ele foi concebido como um framework de pesquisa e inclui transformações do compilador e otimizações para OpenACC. Há alguns compiladores comerciais que reconhecem diretivas OpenACC. Tais compiladores são produzidos por empresas como *Cray* e *CAPS entreprise*. Espera-se que nos anos vindouros, compiladores mais populares, como gcc e LLVM, também sejam capazes de reconhecer diretivas OpenACC. A técnica de compilação descrita neste trabalho não têm por objetivo traduzir código C em código CUDA. Ao contrário, neste trabalho foi usado um compilador que faz tal trabalho: PGCC. O que a técnica proposta neste artigo faz é distribuir diretivas OpenACC em um programa. PGCC não faz tal distribuição: ele já espera um programa anotado. As diretivas que são colocadas automaticamente em programas estão descritas abaixo:

– `pragma acc data pcopy`: copia dados entre CPU e GPU. Exemplos: `#pragma acc data pcopy(n[0:N-1])` copia `N` posições do arranjo `n` entre CPU e GPU.

– `pragma acc kernels`: marca uma região de código que deve ser executada em um acelerador. Em nosso contexto, o único acelerador considerado é a GPU.
– `pragma acc loop independent`: indica ao compilador que as iterações de um laço são independentes umas das outras, e portanto podem ser executadas em paralelo. Essa pragma força a paralelização do código, mesmo que o tradutor de OpenACC para CUDA não consiga provar que as iterações são independentes.

2.2 Este trabalho em um exemplo

O anotador automático proposto neste trabalho utiliza duas análises, conforme já mencionado n Seção 1: uma análise de divergências [24] e uma análise de tamanho de arranjos. Recebe-se do usuário como entrada um código sequencial em C/C++ e retorna-se um código paralelo anotado com diretivas OpenACC. Para demonstrar o funcionamento de nossa ferramenta, iremos utilizar o exemplo da Figura 1-a. Inicialmente, verificamos se os laços a serem paralelizados possuem divergências. Divergências são um fenômeno particular de modelos de execução SIMD (*Single Instruction Multiple Data*). Essas arquiteturas podem ser imaginadas como formadas por um conjunto de processadores que compartilham o mesmo buscador de instruções. Em presença de instruções condicionais, é possível que o fluxo de execução divirja entre processadores. O laço apresentado em nosso exemplo não possui nenhuma instrução condicional e, consequentemente, não apresenta divergências. Garantir uma quantidade pequena de divergências é uma propriedade chave para técnicas de paralelização automática, uma vez que estas podem causar degradação significativa de performance.

```
1 for (int i=0; i<N; i++) {
2   for (int j=1; j<N; j++) {
3     b[i*N+j] = i;
4     a[i*N+j] += b[i*N+(j-1)];
5   }
6 }
```
(a)

```
1 #pragma acc data pcopy(a[1:N*(N-1)+N],
                         b[0:N*(N-1)+N])
2 #pragma acc kernels
4 for (int i=0; i<N; i++) {
5   for (int j=1; j<N; j++) {
6     b[i*N+j] = i;
7     a[i*N+j] += b[i*N+(j-1)];
8   }
9 }
```
(b)

Figura 1: (a) Código sequencial de entrada; (b) Código de saída, automaticamente paralelizado por nossa técnica

Para que uma operação seja executada em um acelerador, seus dados de entrada devem estar presentes em sua memória intern. Dessa forma, após verificada a ausência de divergências, é necessário definir o conjunto de dados que deve ser movido para a GPU. Isto é feito inserindo-se diretivas de cópia de dados para

todas as regiões de memória acessadas em um laço. Linguagens como C e C++, entretanto, não agregam informação de tamanho à memória alocada, tornando esta uma tarefa complexa. Automatizamos esta etapa através de uma análise de inferência de tamanho de regiões de memória. Esta análise verifica quais arranjos são utilizados dentro de um determinado laço, bem como suas expressões de acesso, a fim de definir seus limites simbólicos. Limites simbólicos são dados por variáveis do programa e devem estar disponíveis logo antes da entrada do laço. O funcionamento desta análise é datalhado nSeção 3. A figura 1-b apresenta o programa de exemplo após ser automaticamente anotado por nossa técnica. A primeira diretiva inserida, `pragma data pcopy` (linha 1), informa ao compilador as regiões de dados em memória que devem ser enviados à GPU. A segunda diretiva (linha 2) indica que o laço anotado deve ser transformado em um *kernel* CUDA. Um *kernel* é uma função escrita em CUDA que é inteiramente executada na GPU. Note que em nenhum momento se faz necessário escrever código CUDA. Este passo fica a cargo do compilador de diretivas OpenACC.

3 Solução

Nossa solução é composta por duas análises estáticas de código, executadas a nível de laços do programa, como pode ser vistoa Figura 2. Inicialmente, uma análise de divergências é utilizada para avaliar o quão vantajoso seria enviar um determinado laço para a GPU: um alto número de instruções divergentes pode provocar perda de desempenho. Após selecionarmos os laços a serem enviados para o acelerador, uma técnica de inferência de tamanho de arranjos é utilizada para determinar as regiões de dados em memória que devem ser movidas para o hardware paralelo. As duas análises descritas a seguir foram implementadas sobre o compilador LLVM.

3.1 Análise de Divergencias

Para explicar como divergências podem ocorrer, é necessário a compreensão da hierarquia de funcionamento de uma GPU. Dentro de uma GPU, *threads* são agrupadas em *warps*, blocos e grades ou *grids*. *Threads* comunicam-se apenas com outras *threads* em uma mesma grade, sendo esta dividida em blocos. *Threads* compartilham memória e barreiras com outras *threads* em um mesmo bloco, o qual se separa em *warps*. Dentro de um *warp*, *threads* seguem o modelo de processamento SIMD (*Single Instruction Multiple Data*). Esta organização interna faz com que instruções condicionais se tornem fonte de potencial perda de desempenho, como exemplificado na Figura 3, num fenômeno conhecido como *divergência*. Neste trabalho, utilizamos uma técnica capaz de avaliar o impacto de dois tipos de divergência: de controle e de dados.

Divergência de dados. Uma divergência de dados ocorre sempre que uma mesma variável é mapeada para valores distintos em *threads* diferentes [24]. Sempre que uma variável é alvo de divergência de dados, esta é dita *divergente*. Uma variável não divergente é considerada *uniforme*.

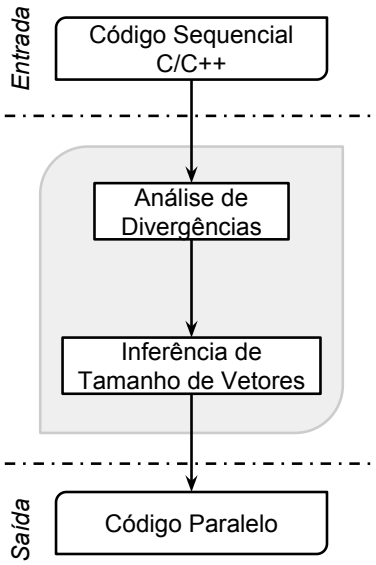

Figura 2: Fluxo de funcionamento das análises aqui propostas.

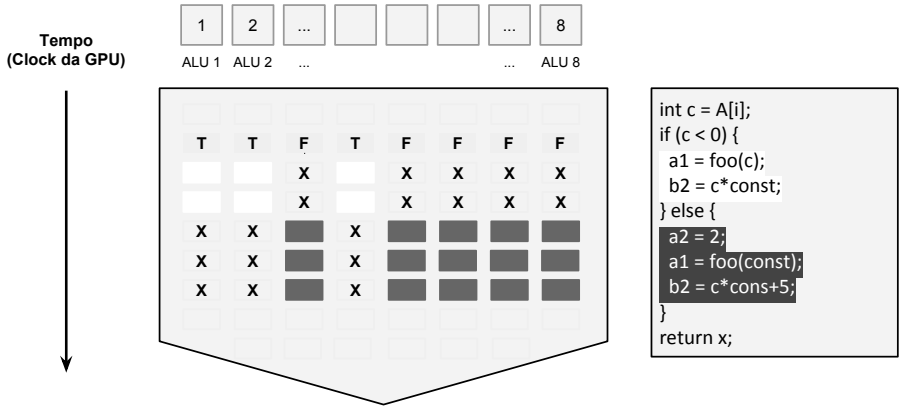

Figura 3: *Warp* com 8 *threads*: um conjunto de ALUs (assinaladas por `X`) deve permanecer ocioso sempre que suas *threads* não cumprirem a condição do bloco *if-else* no programa à direita. Este fato pode causar perda de desempenho.

Divergência de fluxo de controle. Causada pela divergência de dados, a divergência de controle ocorre quando *threads* em um mesmo *warp* seguem fluxos de controle distintos após o processamento de uma instrução de desvio condicional. Se a condição de desvio é controlada por dados divergentes, esta

pode ser verdadeira para algumas *threads*, e falso para outras. Devido ao fato de cada *warp* possuir apenas um buscador de instruções, *threads* que cumprem a condição para um dado fluxo de controle seguem sua execução normalmente, enquanto aquelas que não obedecem à condição permanecem ociosas. A análise que implementamos neste artigo contabiliza a porcentagem de instruções divergentes em laços de um programa. Os resultados desta análise são então combinados a limiares obtidos de forma empírica e então utilizados para identificar laços cuja proporção de instruções divergentes seja superior a uma dada proporção. Assumimos então que estes laços não devem ser enviados à GPU, uma vez que é provável que se tornem mais lentos do que sua versão sequêncial.

Análises de divergência foram já descritas em literatura prévia [5,24]. A análise usada neste artigo é uma adaptação do algoritmo proposto por Sampaio *et al.* [24], e hoje disponível no compilador LLVM[1]. Nota-se, contudo, que este trabalho descreve a primeira tentativa de se utilizar análise de divergências para melhorar modelos de custo que tentam prever o desempenho relativo de computação executando na CPU ou na GPU.

3.2 Inferência de tamanho de arranjos

Sempre que um laço é enviado à GPU, os dados sobre os quais o mesmo opera devem também ser movidos para a memória interna do hardware paralelo. Em OpenACC, estas operações de cópia são definidas pela diretiva `data`, à qual o programador deve fornecer os limites simbólicos de acesso para cada arranjo referenciado no corpo do laço alvo. A fim de automatizar estas operações de cópia, construímos uma análise capaz de inferir tais limites. Inicialmente, derivamos o maior e menor valor simbólico que cada variável de indução em um aninhamento de laços pode assumir. Para tanto, combinamos as expressões condicionais que controlam o termino de cada laço à *Análise de Evolução Escalar* [2] presente em LLVM. Estes valores são então substituídos em cada expressão de indexação de memória encontrada ao longo do aninhamento, definindo assim seus limites. Estes últimos são então combinados em operações de máximo e mínimo, disponíveis logo antes do início de cada laço, definindo assim os intervalos simbólicos de cada arranjo, os quais são utilizados para geração automática de diretivas de cópia de dados entre memória principal e GPU. A Figura 4 mostra o resultado obtido ao se aplicar esta análise a um programa de exemplo.

A Figura 4 nos permite ver que a análise de inferência de tamanhos é simbólica. Isto quer dizer que ela associa regiões de memória com limites que podem ser variáveis, constantes, ou expressões envolvendo máximo, mínimo, soma e multiplicação. Análises de inferência de tamanho de arranjo não são uma novidade. Neste trabalho estamos usando a implementação de Alves *et al.* [1]. Entretanto, o uso desse tipo de análise para permitir a cópia de dados correta e automática entre dispositivos é uma contribuição desse artigo.

[1] `http://llvm.org/docs/doxygen/html/DivergenceAnalysis_8cpp_source.html`

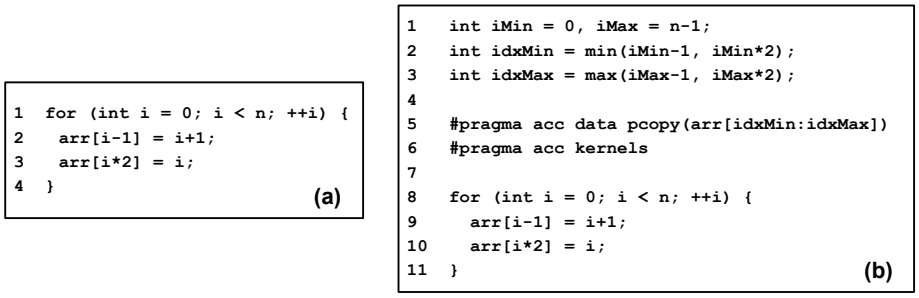

Figura 4: Inferência de tamanho de arranjos. (a) Programa original; (b) Programa automaticamente anotado.

4 Resultados Experimentais

Esta seção demonstra a efetividade do nosso arcabouço de paralelização de código. Com tal intuito, a Seção 4.1 mostra a quantidade de situações diferentes que somos capazes de analisar. A Seção 4.2 demonstra os *speedups* alcançados com a execução da ferramenta proposta neste trabalho. No que se segue, descreve-se o ambiente usado para a realização dos experimentos:

Hardware: O hardware utilizado nesses experimentos é descrito a seguir:

CPU: processador Intel Xeon CPU E5-2620, 6 cores de 2.00GHz e 16 GB de RAM (DDR2). Sistema operacional: Linux Ubuntu 12.04 3.2.0;

GPU: placa gráfica GeForce GTX 670, com 2 GB de memória RAM (CUDA Compute Capability 3.0).

Software: Os testes mostrados nesta seção foram realizados sobre diversos benchmarks, including Parboil, Rodinia e OMPSpec. Dada a grande quantidade de benchmarks disponíveis, mostraremos números condensados para a maior parte deles; entretanto, mostraremos tempo de execução para todos os benchmarks disponíveis em Polybench[2]. Os speedups alcançados para os outros benchmarks são semelhantes. PolyBench [19] é um pacote composto de programas projetados para avaliar o desenvolvimento do modelo poliédrico de paralelismo. Há dezenas de pequenos programas neste conjunto de benchmarks. Escolheu-se trabalhar com os 15 programas usados por Gray *et al* [10]. Para a geração de código paralelo foi utilizado o compilador PGCC (*Portland Group C Compiler*) [3] versão 16.1. As flags usadas em linha de comando foram: `-fast -Mipa=fast,inline -Msmartalloc -acc`. As análises estáticas foram implementadas em LLVM 3.7 [4].

[2] `https://sourceforge.net/projects/polybench/`
[3] `https://www.pgroup.com/support/compile.htm`
[4] `http://llvm.org/`

	Laços		Memória		Instruções	
Nome	**Total**	**Anotáveis**	**Acessos**	**Analízáveis**	**Total**	**Divergentes**
2DConv	8	8	76	74	325	9
2mm	22	22	101	99	406	144
3DConv	12	12	147	145	431	14
3mm	28	28	139	137	496	178
atax	12	12	72	70	290	106
bicg	13	13	80	78	330	133
correlation	22	22	232	230	476	12
covariance	18	18	128	126	367	0
fdtd2d	23	23	151	149	504	95
gemm	14	14	69	67	301	108
gesummv	7	7	84	82	258	96
gramschmidt	16	16	144	142	370	163
mvt	11	11	84	82	327	128
syr2k	13	13	96	94	305	114
syrk_m	13	0	66	64	245	87
syrk	15	15	68	66	297	107
Total	**247**	**234**	**1737**	**1705**	**5728**	**1494**

Figura 5: Dados estáticos coletados com o paralelizador automático.

4.1 Resultados Estáticos

O objetivo da ferramenta é paralelizar o maior número de laços e enviar o mínimo de dados possível para a GPU. O paralelizador é executado de forma automática e suas análises são totalmente estáticas, ou seja, são realizadas em tempo de compilação. Com as análises disponíveis em nossa ferramenta, é possível coletar vários dados que demonstram a eficiência da mesma.

Para inserir diretivas OpenACC, o paralelizador deve ser capaz de analisar todos os acessos à memória presentes em um laço, derivando limites simbólicos para os mesmos. Isto se deve à necessidade de se mover os dados de entrada de um laço para a memória interna do hardware paralelo, como detalhado na Seção 3.2. A Tabela 5 mostra que conseguimos analisar 98% dos acessos a memória presentes em cada um dos benchmarks de Polybench, o que maximiza o número de laços que conseguiremos anotar. Nos *benchmarks* analisados é possívelnotar 234 [5] laços, ou seja, 95% dos laços presente. Note que nossa análise é totalmente automática. Em outras palavras, ela remove do programador o ônus de inserir diretivas de cópias de dados em todos os laços encontrados em Polybench.

Além da memória analisada e laços anotados, coletamos a quantidade de instruções divergentes. Este dado é importante para a identificação de upossível perda de desempenho ao se enviar um laço para a GPU, como explicado n Seção 3.1. Ainda na Tabela 5 podemos ver que aproximadamente 26% das instruções encontradas são divergentes. Este número pode ser pequeno olhando o

[5] São analisados 234 laços, para a coleta deste valor não foram considerado laços divergentes.

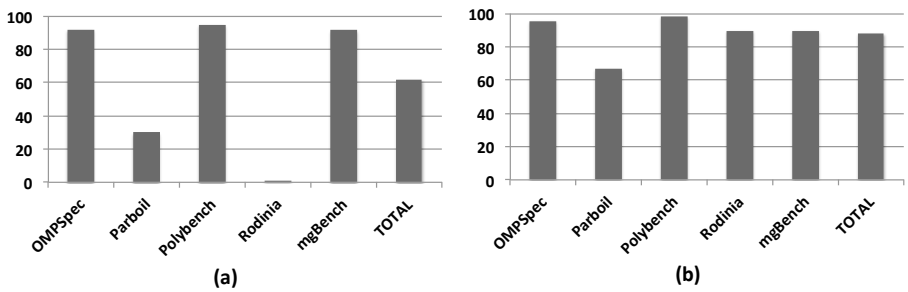

Figura 6: (a)Porcentagem de laços Anotáveis; (b)Porcentagem de memória analizável.

somatório geral, porém o paralelizador não verifica a porcentagem de instruções divergentes presentes no programa e sim a quantidade presente em um laço. Se essa porcentagem for maior que 80% não enviamos o mesmo para a GPU.

A fim de demonstrar a capacidade de nossa análise de lidar com programas reais, nós a utilizamos em vários outros benchmarks além de Polybench. Selecionamos algumas funções de *benchmarks* para avaliar aplicações paralelas. Essas estão listadas a seguir:

- OMPSpec [23] foi criado para mensurar a performace de aplicações baseadas em OpenMP para processamento de aplicações paralelas de memória compartilhada;
- Parboil [26] é uma coleção de aplicações para estudo de performace e computação da taxa de transferência entre arquiteturas heterogêneas;
- Rodinia [4] criado para arquiteturas heterogênias;
- MGBench [6] coleção de 8 aplicações criadas pelos autores para testar um paralelizador baseado em *profiling* de programas.

Nós coletamos os dados e os sumarizamos nos gráficos da Figura 6. Como pode ser visto nossa análise é capaz de anotar aproximadamente 62% dos laços presentes em todos os benchmarks analisados[6]. A inferência de tamanhos de arranjos é capaz de encontrar limites para mais de 80% dos acessos a memória presentes nas funções. Diz-se que uma operação de acesso à memória possui "limites" quando seu menor e maior *offsets* válidos são conhecidos em tempo de compilação. Note que esse valor é maior que a proporção de laços que conseguese anotar. Para anotar um laço é necessário que cada um dentre seus acessos seja "analisável". Assim, basta um acesso com limites desconhecidos dentro de um laço para que deixemos de anotá-lo.

[6] O benchmark Rodinia teve um comportamento discrepante devido a um ajuste feito na análise para não anotar laços que contenham arranjos com duas ou mais dimensões, devido à uma limitação do compilador utilizado, mesmo sendo capaz de analizar os acessos à memória presentes.

4.2 Tempo de Execução

A Figura 7 mostra resultados para o tempo de execução dos benchmarks usados, com e sem as diretivas de paralelização de código. Cada benchmark possui entradas de cinco tamanhos diferentes - a tabela mostra resultados para cada uma delas. Nota-se que há grande variação de tempo de execução entre as versões CPU e GPU de cada benchmark. Além disso, alguns dos benchmarks executaram durante um tempo muito curto, menos de um segundo, mesmo com sua maior entrada. Nesses casos, houve grande variação de tempo relativo entre versões paralelas e sequenciais. Assim, para evitar distorções, a discussão que acontece no restante desta seção fará referência somente aos benchmarks que executaram por mais de 1 segundo.

O maior speedup observado dentre tais programas foi de 121x na execução da multiplicação de matrizes tridimensionais 3MM (293.513secs na CPU vs 2.424secs na GPU). O maior slowdown observado foi de 11x em GRAMSCHM (22.426secs na CPU vs 251.832secs na GPU). A Figura 8 compara esses resultados. Novamente: mostram-se speedups relativos somente entre benchmarks que executaram por mais de um segundo. Pela figura, nota-se que três dos benchmarks (GEMM, 2MM e 3MM) experimentaram grandes melhorias de tempo de execução na GPU. Houve, contudo, pelo menos dois benchmarks que experimentaram lentidão. A razão por trás dessa lentidão é reúso de dados: alguns desses benchmarks possuem complexidade linear. Contudo, o tempo de copiar os dados para a memória da GPU é também linear. Assim, o tempo de cópia termina por remover possíveis ganhos que se obtém com o maior poder computacional da GPU. Técnicas de análise de complexidade computacional, como os trabalhos feitos por Gawlitza *et al.* [8] e Gulavani *et al.* [11] podem ser usados para mitigar esse problema. Nesse caso, envia-se para a GPU somente trabalho super-linear. Tal abordagem foi usada com sucesso por Etino [6], porém, naquele caso, usou-se um profiler para inferir a complexidade asimptótica dos algoritmos paralelizáveis.

5 Trabalhos Relacionados

GPUs proporcionam alto desempenho e computação a baixo custo. Entretanto, programar para este tipo de hardware ainda se mostra uma tarefa difícil para grande parte dos desenvolvedores. Nos últimos anos, fabricantes e pesquisadores propuseram diversos modelos de programação baseados em diretivas, a fim de tentar diminuir esta barreira. Em seu trabalho, Lee e Vetter [17] avaliam os principais modelos de programação baseados em diretivas (hiCUDA, OpenMPC, Acelerador PGI, HMPP, R-Stream, OpenACC e OpenMP) em termos de funcionalidade, escalabilidade e capacidade. Seus resultados apontam que esses modelos alcançam desempenho razoável em comparação com código para GPU escrito manualmente. No entanto, esses modelos necessitam de uma série de otimizações capazes de tirar proveito de detalhes específicos de arquitetura e organização de memória para atingir um desempenho competitivo em certas aplicações. Em nosso trabalho, retiramos do programador o ônus de lidar com estas otimizações

Name	Device	MINI	SMALL	MEDIUM	LARGE	EXTRA LARGE
SYRK	CPU	0.000	0.000	0.006	0.476	5.063
	GPU	0.623	0.620	0.665	1.622	5.358
SYRK_M	CPU	0.001	0.006	0.045	0.293	2.885
	GPU	0.001	0.009	0.051	0.362	2.888
GEMM	CPU	0.000	0.000	0.036	10.891	141.564
	GPU	0.561	0.624	0.629	0.772	1.390
CORR	CPU	0.002	0.013	0.185	5.843	75.948
	GPU	0.727	1.328	6.365	47.407	490.835
COVAR	CPU	0.002	0.013	0.180	5.841	74.172
	GPU	0.716	1.331	6.386	47.502	489.868
3MM	CPU	0.000	0.009	0.969	32.598	293.513
	GPU	0.621	0.622	0.742	1.078	2.424
2MM	CPU	0.000	0.006	0.670	21.665	194.512
	GPU	0.623	0.612	0.698	0.923	1.773
ATAX	CPU	0.000	0.000	0.001	0.014	0.048
	GPU	0.622	0.629	0.629	0.754	1.145
GRAMSCHM	CPU	0.000	0.007	0.069	0.991	22.426
	GPU	0.683	1.076	3.807	27.399	251.832
MVT	CPU	0.000	0.000	0.001	0.059	0.268
	GPU	0.621	0.624	0.610	0.655	0.752
BICG	CPU	0.000	0.000	0.001	0.011	0.035
	GPU	0.623	0.612	0.629	0.739	1.139
FDTD-2D	CPU	0.000	0.002	0.048	4.523	37.831
	GPU	0.620	0.629	0.654	1.407	4.200
2DCONV	CPU	0.001	0.002	0.020	0.057	0.274
	GPU	0.612	0.612	0.673	0.790	1.529
GESUMMV	CPU	0.000	0.000	0.001	0.003	0.013
	GPU	0.628	0.623	0.630	0.658	0.815
3DCONV	CPU	0.000	0.001	0.010	0.072	0.010
	GPU	0.619	0.624	0.631	0.788	1.768

Figura 7: Tempo de execução de programas paralelizados automaticamente, com entradas de cinco diferentes tamanhos.

manuais, inserindo diretivas de paralelização de forma completamente automática.

Lee et al. [16] apresenta em seu trabalho um compilador para tradução automática fonte-a-fonte de aplicações OpenMP para código CUDA GPGPU. O objetivo desta tradução é facilitar a programação e fazer com que aplicações existentes em OpenMP possam ser executados em GPUs. Nosso objetivo neste trabalho é, a partir de um código totalmente sequencial, ser capaz de identificar quais laços devem ser paralelizados, bem como seus conjuntos de dados de entrada. O trabalho de tradução para código CUDA é deixado ao encargo do compilador de OpenACC, neste caso PGCC. Kao e Hsu [14] propõem uma técnica de programação para arquiteturas heterogêneas utilizando um perfilador

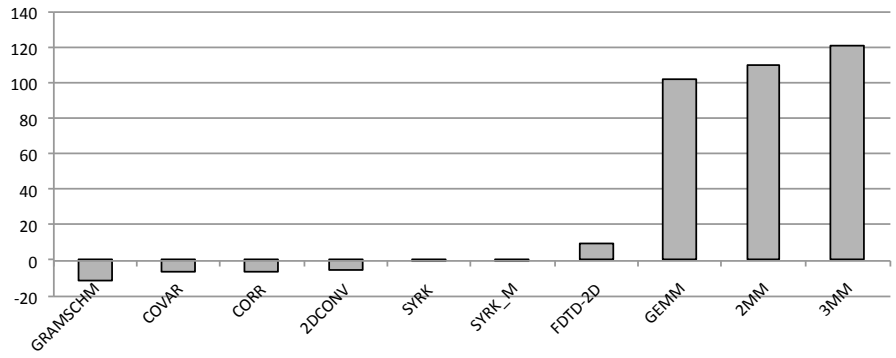

Figura 8: Speedup relativo entre os diferentes programas de Polybench que executaram por mais de um segundo com as maiores entradas disponíveis.

capaz de determinar quando é benéfico enviar parte de um programa para um acelerador. Ao contrário dessa abordagem, a técnica que propomos é completamente estática e não depende de conjuntos específicos de valores de entrada. Várias linguagens de programação oferecem sintaxe e semântica para processamento paralelo. *Haleto* [22], por exemplo, uma linguagem para aplicações de processamento de imagens oferece ao desenvolvedor a oportunidade de identificar quais partes de um programa devem ser executadas em paralelo. A linguagem permite também ao programador especificar a forma como regiões paralelas devem ser distribuidas nos diferentes dispositivos de processamento disponíveis. Ao contrário dessas abordagens, as técnicas aqui propostas são complemtamente automáticas.

6 Conclusão

Neste trabalho descrevemos um conjunto de análises estáticas capazes de anotar código C sequencial com diretivas OpenACC, de forma totalmente automática, possibilitando sua execução em unidades de processamento gráfico. Nossa abordagem é uma extensão da ferramenta Etino, lhe adicionando a capacidade de paralelizar código sem a necessidade de um perfilador. Implementamos nossa técnica sobre o compilador LLVM e a utilizamos para paralelizar automaticamente diversos programas, obtendo ganhos de performance expressivos. Neste trabalho, utilizamos diretivas OpenACC. No entanto, nada impede que uma abordagem similar seja utilizada em conjunto com outros sistemas de anotação, tais como OpenMP 4.0 e OpenSs. Dada a crescente popularidade de hardware heterogêneo, acreditamos que ferramentas como a que apresentamos neste trabalho, capaz de facilitar significativamente a tarefa do programador, serão cada vez mais importantes na indústria de desenvolvimento de software.

Software

O paralelizador descrito neste trabalho está disponível via uma interface web: `http://cuda.dcc.ufmg.br/dawn/`. Essa interface recebe um programa em C, e devolve um programa anotado com pragmas OpenACC. Cabe ao usuário compilar esse programa de saída, usando para tanto um compilador que leia pragmas OpenACC.

Agradecimentos

Este trabalho é patrocinado pela LG Electronics do Brasil, via Lei da Informática. Kézia Andrade é recipiente de uma bolsa de pesquisa financiada pela Intel. Fernando Pereira é suportado pelo CNPq.

Referências

1. Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. Runtime pointer disambiguation. In *OOPSLA*, pages 589–606. ACM, 2015.
2. Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *In ISSAC*, pages 242–249. ACM, 1994.
3. T. Bai, C. Ding, and P. Li. Assessing safe task parallelism in spec 2006 int. In *Cluster, Cloud and Grid Computing (CCGrid)*, pages 402–411, May 2015.
4. Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54. IEEE, 2009.
5. Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and Wagner Meira Jr. Divergence analysis and optimizations. In *PACT*, pages 320–329. IEEE, 2011.
6. Douglas do Couto Teixeira, Kézia Andrade, and Gleison Souza andFernando Pereira. Etino: Colocação automática de computação em hardware heterogêneo. In *SBLP*. SBC, 2015.
7. Michael Garland. Parallel computing experiences with CUDA. *IEEE Micro*, 28:13–27, 2008.
8. Thomas Martin Gawlitza and David Monniaux. Invariant generation through strategy iteration in succinctly represented control flow graphs. *Logical Methods in Computer Science*, 8(3), 2012.
9. Swapnil Ghike, Ruben Gran, María Jesús Garzarán, and David A. Padua. Directive-based compilers for gpus. In James C. Brodman and Peng Tu, editors, *LCPC*, volume 8967 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2014.
10. S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *InPar*, pages 1–10. IEEE, 2012.
11. Bhargav S. Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, pages 370–384. Springer, 2008.

12. John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5th edition, 2011.
13. Julien Jaeger, Patrick Carribault, and Marc Pérache. Fine-grain data management directory for openmp 4.0 and openacc. *Concurrency and Computation: Practice and Experience*, pages 1528–1539, 2015.
14. Chih-Chen Kao and Wei-Chung Hsu. An adaptive heterogeneous runtime framework for irregular applications. *J. Signal Process. Syst.*, 2015.
15. Chris Lattner and Sarita V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *CGO*, pages 75–86, 2004.
16. Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *PPoPP*, pages 101–110. ACM, 2009.
17. Seyong Lee and Jeffrey S. Vetter. Early evaluation of directive-based gpu programming models for productive exascale computing. In *In CHPCNSA*, SC '12. IEEE, 2012.
18. Seyong Lee and Jeffrey S. Vetter. Openarc: open accelerator research compiler for directive-based , efficient heterogeneous computing. In *HPDC*, pages 115–120. ACM, 2014.
19. Dajiang Liu, Shouyi Yin, Leibo Liu, and Shaojun Wei. Polyhedral model based mapping optimization of loop nests for cgras. In *DAC*, pages 19:1–19:8, New York, NY, USA, 2013. ACM.
20. Cor Meenderinck and Ben H. H. Juurlink. Nexus: Hardware support for task-based programming. In *DSD*, pages 442–445. Springer, 2011.
21. Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Ssmart: Smart scheduling of multi-architecture tasks on heterogeneous systems. In *In WACCPD*, pages 1:1–1:11. ACM, 2015.
22. Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, pages 519–530. ACM, 2013.
23. Hideki Saito, Greg Gaertner, Wesley Jones, Rudolf Eigenmann, Hidetoshi Iwashita, Ron Lieberman, Matthijs van Waveren, and Brian Whitney. Large system performance of spec omp benchmark suites. *Int. J. Parallel Program.*, 31:197–209, 2003.
24. Diogo Sampaio, Rafael Martins de Souza, Sylvain Collange, and Fernando Magno Quintão Pereira. Divergence analysis. *ACM Trans. Program. Lang. Syst.*, 35(4):13:1–13:36, 2014.
25. OpenACC Standard. The openacc programming interface. Technical report, CAPs, 2013.
26. John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W-m Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *In CRHPC*, 2012.
27. Sandra Wienke, Paul L. Springer, Christian Terboven, and Dieter an Mey. OpenACC - first experiences with real-world applications. In *Euro-Par*, pages 859–870. Springer, 2012.
28. Minjia Zhang, Swarnendu Biswas, and Michael D. Bond. Relaxed dependence tracking for parallel runtime support. In *In CC*, pages 45–55. ACM, 2016.