Geração Automática de Estênceis Otimizados para GPUs

Alyson D. Pereira[1], **Rodrigo C. O. Rocha**[3], **Márcio Castro**[1], **Luís F. W. Góes**[2]

[1] Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD)
Universidade Federal de Santa Catarina (UFSC) – SC, Brasil

[2] Grupo de Computação Criativa e Paralela (CreaPar)
Pontifícia Universidade Católica de Minas Gerais (PUC Minas) – MG, Brasil

[3] Institute for Computing Systems Architecture (ICSA)
University of Edinburgh – Escócia, Reino Unido

alyson.pereira@posgrad.ufsc.br, r.rocha@ed.ac.uk,
marcio.castro@ufsc.br, lfwgoes@pucminas.br

Resumo. *Neste artigo propomos uma ferramenta que utiliza uma análise estática para detectar computações estêncil em laços aninhados em um códigos C/C++ e um gerador de código que, baseado nas informações do padrão de vizinhança da computação estêncil, gera um código CUDA otimizado. Para validar a nossa ferramenta, analisamos um conjunto de códigos presentes no benchmark Polybench, o qual contem códigos dos domínios de estatística, algebra linear e estêncil. Os resultados mostraram que a análise estática foi capaz de detectar corretamente o padrão estêncil. Além disso, o código gerado pela ferramenta proposta apresentou desempenho de até* $2.25\times$ *ao código gerado automaticamente por um compilador referência no estado da arte.*

1. Introdução

Um importante padrão paralelo utilizado em aplicações científicas é o estêncil, obtido através da discretização de equações diferenciais parciais. Este padrão opera em estruturas de dados multidimensionais, nas quais uma mesma computação é realizada sobre cada elemento utilizando os valores de seus vizinhos. Pesquisas recentes tentam explorar o uso eficiente de Unidades de Processamento Gráfico (*Graphics Processing Units* - GPUs) para a execução de aplicações estêncil, tendo em vista o seu alto grau de paralelismo de dados [Meng and Skadron 2011, Maruyama and Aoki 2014, Holewinski et al. 2012].

Diversas são as abordagens para a programação de GPUs: linguagens de baixo nível, bibliotecas de esqueletos paralelos, linguagens de domínio específico e diretivas de compilação. A linguagem de baixo-nível CUDA permite um controle maior sobre otimizações de código, porém requer um conhecimento da arquitetura da GPUs e pode ser desafiadora para os menos experientes. As bibliotecas de esqueletos paralelos e as linguagens de domínio específico facilitam a programação ao mesmo tempo que permitem explorar determinadas otimizações, no entanto requerem uma reescrita de códigos estêncil já existentes e geralmente apresentam certas restrições ao programador. Com o uso de diretivas, como OpenACC e OpenMP 4.5, o programador pode utilizar um código-fonte sequencial já existente e apenas anotá-lo com as devidas diretivas, indicando os trechos de códigos que serão executados na GPU. Apesar do uso das diretivas ser mais simples do ponto de vista de programação, elas não exploram otimizações de baixo nível que podem ser obtidas ao se utilizar as abordagens anteriores.

Neste artigo propomos uma abordagem que utiliza uma análise estática que detecta computações estêncil em laços aninhados em um código-fonte C/C++ e um gerador de código que, baseado nas informações do padrão de vizinhança da computação estêncil obtido através da analise anterior, gera um código CUDA otimizado. Ambas as contribuições foram implementadas utilizando a infra-estrutura de compilação LLVM e integradas em uma ferramenta denominada `PSkelCC`. Para validar a nossa ferramenta, analisamos um conjunto de códigos presentes no *benchmark* Polybench[1], que contem códigos dos domínios de estatística, algebra linear e estêncil. A análise estática foi capaz de detectar corretamente os programas selecionados deste *benchmark* que são estêncil e rejeitar os que não são. Além disso, analisamos um conjunto de códigos que implementam um estêncil laplaciano 3D e comparamos o desempenho do código gerado pela nossa ferramenta frente a implementações otimizadas manualmente e a de códigos gerados automaticamente por um compilador *source-to-source* referência no estado da arte. Obtemos um desempenho de até $2.25\times$ sobre o compilador de referência e até $1.26\times$ sobre códigos otimizados manualmente.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta os principais conceitos do padrão estêncil, modelo de programação CUDA e do compilador LLVM. As Seções 4 e 5 discutem a detecção do padrão estêncil em códigos sequenciais e a geração de código CUDA para os estênceis detectados, respectivamente. Os resultados obtidos são apresentados na Seção 6. Por fim, a Seção 3 apresenta os trabalhos relacionados ao tema abordado e a Seção 7 apresenta as conclusões deste trabalho.

2. Fundamentação Teórica

2.1. Computação Estêncil

O padrão estêncil opera em dados multidimensionais e aplica uma computação em cada um de seus elementos, utilizando os valores dos seus vizinhos. Em estênceis iterativos, este procedimento se repete, utilizando os dados de saída como entrada para a próxima iteração. A Figura 1 ilustra alguns padrões de vizinhanças que ocorrem em estênceis 2D e 3D.

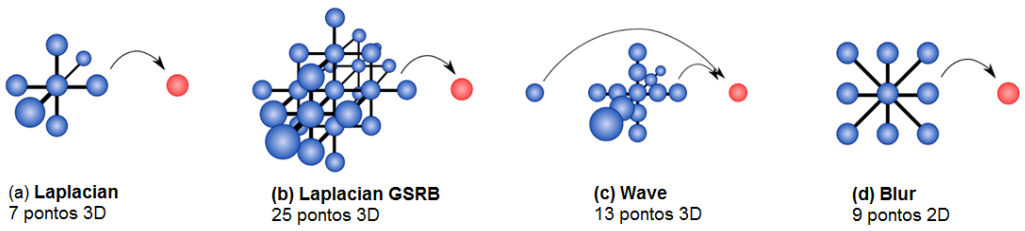

Figura 1. Diferentes vizinhanças do padrão estêncil [Christen et al. 2012].

O Código 1 exemplifica uma computação estêncil iterativa de uma dimensão. Neste exemplo, o número de iterações é determinado pelo parâmetro `tsteps` (linha 1). A computação estêncil é realizada sobre cada elemento do vetor `A`, de tamanho `N`, utilizando o coeficiente `c1` e uma vizinhança de 2 elementos (linha 7), exceto para os

[1]http://web.cse.ohio-state.edu/ pouchet.2/software/polybench/

elementos em sua borda. O resultado da computação é armazenado em um elemento correspondente do vetor `B`. Após efetuar esta computação em todos os elementos do vetor, existe um laço para a troca de dados entre os vetores `A` e `B` (linhas 9 a 10), pois o vetor de resultado de uma iteração t deverá ser utilizada como vetor de entrada da iteração $t + 1$.

```
1  void jacobi(int tsteps, int N, float *A, float *B){
2    int t, i, j;
3    float c1 = 0.33333;
4
5    for (t = 0; t < tsteps; t++) {
6      for (i = 1; i < N - 1; i++)
7        B[i] = c1 * (A[i-1] + A[i] + A[i + 1]);
8
9      for (j = 1; j < N - 1; j++)
10       A[j] = B[j];
11   }
12 }
```

Código 1. Exemplo de um código estêncil 1D iterativo em C.

2.2. Modelo de Programação CUDA

No modelo de programação CUDA, a GPU é vista como um co-processador com massivo paralelismo de dados, capaz de executar centenas de *threads* em paralelo no modelo de execução *Single Instruction, Multiple Threads* (SIMT). Para tal, uma função denominada *kernel* opera, geralmente, em cada elemento dos dados de entrada e é compilada para o conjunto de instruções (*Instruction Set Architecture* - ISA) da GPU. Para manter uma portabilidade entre diferentes microarquiteturas, é utilizado um conjunto de instruções virtual, denominado *Parallel Thread Execution* (PTX). Em tempo de execução, o programa resultante é mapeado em *threads*. Para organizar as diferentes *threads* que executam um mesmo *kernel* no dispositivo, conjuntos de *threads* são combinados em um bloco de *threads*, permitindo que elas compartilhem dados e sincronizem entre si.

Uma GPU moderna é composta por um conjunto de *Streaming Multiprocessors* (SM), onde cada SM é composto por núcleos de processamento, unidades aritméticas, unidades de controle, e sua hierarquia de memória. Podemos considerar que existem três conexões na hierarquia de memória: memória global $\leftrightarrow$ *cache* L2 $\leftrightarrow$ memórias locais $\leftrightarrow$ registradores. A memória *cache* L2 é compartilhada entre todos os SMs da GPU. As memórias locais (internas ao SM) incluem *cache* L1, *cache* somente leitura e memória compartilhada. A largura de banda entre cada uma dessas conexões são consideravelmente diferentes, sendo a largura de banda da memória global a menor de todas. O volume de dados também se difere, sendo o maior de todos entre a memória local e os registradores. Por esse motivo, apesar das GPUs possuírem um alto poder de computação teórico (GPUs atuais alcançam um desempenho na ordem de TeraFLOPS[2]), o desempenho de computações estêncil se torna limitado à largura de banda das GPUs, sendo necessário manter a maior quantidade de dados disponíveis em *cache* para reuso pelos blocos de *threads*, explorando a localidade de dados tanto de maneira espacial (compartilhando dados entre as *threads* de um mesmo bloco através das memórias locais) quanto de maneira temporal (realizando mais de uma iteração do estêncil internamente, reutilizando os dados já armazenados nas memórias locais e reduzindo a comunicação com as memórias externas, ao custo de computações redundantes).

[2]http://www.nvidia.com/object/tesla-p100.html

2.3. Análise Estática com LLVM

O *Low Level Virtual Machine* (LLVM) [Lattner and Adve 2004] é uma infraestrutura de compilação amplamente utilizada na indústria. Ele é composto principalmente por uma representação intermediária (*LLVM Intermediate Representation* - IR[3]) e bibliotecas para análise e otimização de códigos. O processo de compilação é composto por três etapas: um *frontend*, um otimizador e um *backend*, como exibido na Figura 2. O *frontend* é responsável por realizar um *parsing* no código fonte, verificando por erros sintáticos e semânticos, e transformá-lo na representação intermediária do LLVM. O otimizador é responsável por realizar uma vasta quantidade de análises e otimizações sobre a IR de maneira independente da arquitetura alvo. Por fim, o *backend* traduz o código da representação intermediária para o conjunto de instruções da máquina alvo.

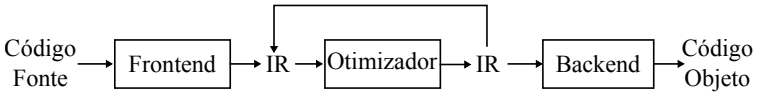

Figura 2. Visão geral da compilação em três passos do LLVM.

A IR do LLVM é um conjunto de instruções de baixo nível do tipo RISC, com tipagem forte, que opera em uma arquitetura *load/store* considerando um número infinito de registradores virtuais. Além disso, a IR segue a forma *Static Single Assignment* (SSA), onde cada registrador virtual é escrito uma única vez e seu uso ocorre após a sua definição. A principal vantagem de se utilizar esta forma é que ela simplifica e melhora diversas otimizações e análises [Cytron et al. 1991]. Uma outra característica da forma SSA é o uso da função ϕ. Esta função é aplicada na transformação para a forma SSA canônica, quando são atribuídos para uma mesma variável valores distintos de acordo com o fluxo de controle do programa.

Informações sobre a IR de um programa podem ser obtida através das diversas análises do LLVM, implementadas como passes[4]. Neste trabalho, fazemos uso de duas análises para a detecção do padrão estêncil: `Scalar Evolution` e `LoopInfo`. O passe `Scalar Evolution` calcula para cada registrador inteiro uma expressão fechada, denominada evolução escalar, que descreve o seu valor durante a execução do programa [Bachmann et al. 1994]. A vantagem de seu uso é que ela abstrai as instruções que compõem o valor de um registrador e foca no cálculo de uma maneira geral. A expressão de uma evolução escalar é construída recursivamente através de diversos elementos. Existem dois elementos base principais e um conjunto de elementos definidos indutivamente. Os elementos base são *integer constant*, que correspondem as constantes inteiras conhecidas em tempo de compilação, e o *unknown value*, que corresponde a valores dinâmicos, conhecidos apenas em tempo de execução, por exemplo, como parâmetros de função ou uma instrução de `load`. Alguns dos elementos indutivos são as operações n-árias como adição (+), multiplicação (`*`), máximos com e sem sinal (`smax` e `umax`) e a adição recursiva. A adição recursiva representa expressões que se alteram durante a avaliação de um laço. Elas contêm o formato {<base>, +, <step>}<loop>. A *base* de uma adição recursiva corresponde ao seu valor na primeira iteração do laço *loop* e o *step* define o valor adicionado em cada iteração subsequente. Caso o registrador esteja

[3]http://llvm.org/docs/LangRef.html
[4]https://llvm.org/docs/Passes.html

contido em laços aninhados e seu valor dependa dos laços externos (como o indice do laço mais externo, por exemplo), a base da adição recursiva será uma outra adição recursiva.

O passe `LoopInfo` obtém informações sobre laços que utilizam, por exemplo, a construção `for`. Com o uso do passe `LoopInfo`, é possível obter a quantidade de vezes que o laço é executado, na forma de uma expressão escalar, e qual a sua variável de indução canônica, na forma de uma instrução ϕ. Uma variável de indução canônica representa uma variável com valor inicial zero e que é incrementada em um à cada iteração do laço. Uma variável de indução que é incrementada em um laço mas não se inicia em zero não é canônica.

3. Trabalhos Relacionados

Alguns trabalhos recentes propuseram abordagens para a geração automática de um código sequencial em um código correspondente para GPUs. O compilador PPCG é o atual estado da arte em compilação automática de códigos sequenciais para CUDA [Verdoolaege et al. 2013]. O compilador é baseado no modelo *poliedral* e é capaz de realizar análise de dependências e transformações de laços sem intervenção do usuário. No entanto, existe a restrição de que apenas laços estáticos sejam transformados.

Em um trabalho prévio [Pereira et al. 2017] foi proposto uma diretiva de compilação específica para o padrão estêncil e um compilador *source-to-source* que transforma códigos sequenciais anotados com a diretiva estêncil em códigos equivalentes de um *framework* de esqueletos paralelos [Pereira et al. 2015]. Neste presente trabalho, eliminamos a necessidade do uso de diretivas e propomos uma análise estática para detecção de computações estêncil em um código sequencial. Além disso, neste trabalho realizamos uma geração de código diretamente em CUDA, o que permite uma maior flexibilidade na escolha de otimizações e transformações de código.

O compilador *BONES* [Nugteren and Corporaal 2014] é um compilador *source-to-source* que utiliza um conceito denominado "espécies algorítmicas", que classifica trechos de código de acordo com o padrão de acesso à memória: *element*, *neighbourhood*, *chunk*, *full* e *shared*. Uma espécie correspondente ao padrão estêncil é formada pela combinação dos padrões *neighbourhood* e *element*. Bones recebe um código C anotado com as espécies algorítmicas e gera um código CUDA adequado.

O compilador CUDA-Chill transforma códigos sequenciais em códigos CUDA baseado em um conjunto de regras informadas pelo usuário [Khan et al. 2013]. Estas regras incluem o mapeamento de dados na hierarquia de memória, o tamanho dos *tiles* e níveis para desenrolamento de laços.

Por fim, o compilador DawnCC, diferentemente das abordagens anteriores, realiza a anotação automática das diretivas de compilação dos padrões OpenACC e OpenMP 4.5 em um código-fonte sequencial, permitindo sua execução em GPU. A ferramenta é implementada como uma conjunto de análises estáticas em LLVM, capaz de detectar se um laço é paralelizável [Mendonca et al. 2017].

4. Detecção do Padrão Estêncil

Nesta seção, ilustramos as etapas da detecção de um padrão estêncil implementado na ferramenta `PSkelCC`. A detecção foi implementada como um passe de otimização que é

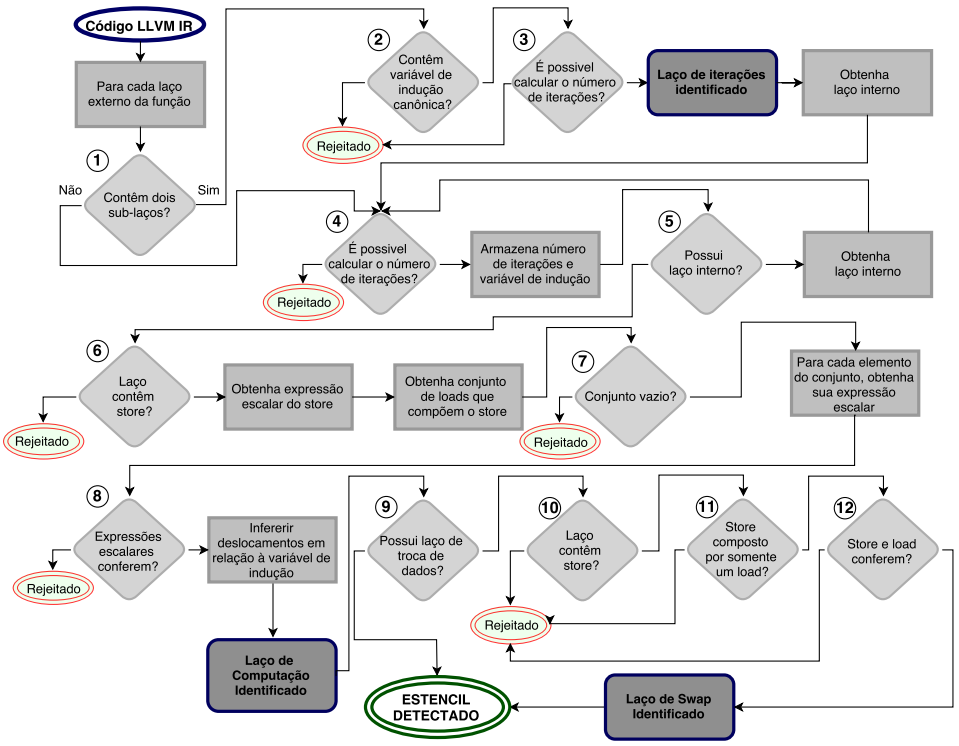

Figura 3. Visão geral da detecção do padrão estêncil.

executado para cada função existente no código IR. Na implementação atual, consideramos como pré-condições a verificação entre as dependências de dados e que os ponteiros analisados não possuem *aliasing* entre si. Uma análise de dependências de dados poderia, por exemplo, verificar as condições básicas Bernstein [Bernstein 1966]. Tais condições garantem que um código pode ser executado em paralelo, mantendo a semântica original. Para que as condições de Bernstein possam ser verificadas, é necessário que se tenha uma análise de *aliasing* de ponteiros, para que as condições de leitura e escrita em memória possam ser garantidas mesmo com o uso de ponteiros.

A Figura 3 exibe o fluxograma do processo de detecção do padrão estêncil proposto neste artigo. Primeiramente, obtemos os resultados da análise `LoopInfo` para que se possa analisar a estrutura dos laços. Para cada laço mais externo da função analisada, realiza-se as verificações enumeradas de 1 a 12 no fluxograma. As etapas de 1 a 3 se referem a detecção do laço de iterações. O laço de iterações deve conter dois sublaços (etapa 1), um correspondente ao laço mais externo da computação estêncil (linhas 6 e 7 do Código 1) e outro correspondente ao laço da troca de ponteiros (linhas 9 e 10 do Código 1). Caso ele contenha apenas um sublaço, ele é considerado um laço de computação e executa-se a etapa 4. Caso ele contenha dois sublaços, obtém-se uma expressão que corresponde ao número de vezes que o laço iterações é executado. A análise de evolução escalar é utilizada para inferir o número de iterações executadas pelo laço. Verifica-se também se a variável de indução do laço de iterações é utilizada apenas para o controle do laço. Para isso, verificamos se a varíavel de indução é utilizada apenas nos blocos básicos de cabeçalho e retorno do laço, sem nenhuma utilização no corpo do laço.

As etapas 4 a 8 correspondem a detecção dos laços de computação. A quantidade de laços aninhados corresponde à quantidade de dimensões do estêncil. De maneira similar ao laço de iterações, para cada laço aninhado de computação, obtém-se a expressão correspondente à quantidade de execuções do laço bem como sua variável de indução. As expressões, obtidas na forma de evolução escalar, correspondem ao tamanho de cada dimensão do estêncil. Essas informações serão utilizadas para analisar as expressões de indexação dos acessos à memória.

As principais instruções utilizadas para realizar manipulação de memória são as instruções de `load` e `store`. Essas instruções recebem como um de seus operandos um endereço de memória. Estes endereços, por sua vez, são computados pela instrução `getelementptr`, dado um endereço base e o índice desse elemento. Assim, identificar uma vizinhança estêncil corresponde a identificar todas as instruções de `load`, utilizadas para acessar os dados de entrada, e a instrução de `store`, responsável por armazenar o resultado de saída. As instruções de `load` não podem utilizar um mesmo ponteiro base que é utilizado pela instrução de `store`.

A etapa 6 verifica se o laço mais interno contêm uma instrução de `store` e obtém a expressão de evolução escalar correspondente à esta instrução. Por exemplo, no Código 2, referente à atribuição de um valor ao endereço de memória `B[i]`, a expressão de evolução escalar referente ao registrador `%storeidx` é definida por {`(4 + %B),+,4`}`<%for.cond>`}. Como o ponteiro base %B é do tipo `float`, cada posição de memória corresponde à 4 bytes. A evolução escalar considera a primeira iteração do laço e, como neste laço o valor inicial de sua variável de indução é 1, a base da adição recursiva corresponde ao endereço do ponteiro %B incrementado pelo valor de uma posição de memória. Outras informações obtidas desta expressão é que a instrução está contido no laço que tem início no bloco básico `for.cond`, a cada iteração o endereço base é incrementado em uma posição de memória (*step* da adição recursiva).

```
1  for.cond:
2    %i = phi i64 [ 1, %for.body ], [ %inc, %for.inc ]
3    ...
4  for.body:
5    ...
6    %storeidx = getelementptr float, float* %B, i64 %i
7    store float %val, float* %storeidx
8    ...
9  for.inc:
10   %inc = add i32 %i, 1
11   br label %for.cond
```

Código 2. Armazenamento em um endereço de memória em LLVM-IR.

Posteriormente, na etapa 7, as instruções que compõem o valor a ser armazenado pela instrução `store` são percorridas e cada instrução de `load` é armazenada em uma lista. Para cada elemento desta lista, é gerado a sua expressão de evolução escalar. De maneira que esses acessos componham uma vizinhança estêncil, os valores de *step* e dos laços da adição recursiva devem ser iguais, além de possuirem um mesmo ponteiro base, diferente do ponteiro base da operação de `store` (etapa 8). O ponteiro base de `load` é definido como entrada do estêncil. O valor do deslocamento em relação ao elemento central da computação estêncil é inferido analisando a expressão de indexação de acesso à memória. No Código 3, referente à leitura de um valor no endereço de memória `A[i+1]`, a expressão de acesso é composta pelo valor da variável de indução (registrador `%i`)

incrementado pelo deslocamento em relação ao elemento central da computação estêncil, acessando a partir do ponteiro base %A.

```
1  %add = add i64 %i, 1
2  %loadidx = getelementptr float, float* %A, i64 %add
3  %data = load float, float* %loadidx
```

Código 3. Leitura de um endereço de memória em LLVM-IR.

As etapas 9 a 12 são executadas caso tenha sido detectado previamente um laço de iteração. Caso contrário, o conjunto de laços analisados compõem uma computação estêncil de uma única iteração. Nestas etapas, é verificado se a instrução `store` é composta por apenas uma instrução de `load`, e se o ponteiro base das instruções de `load` correspondem ao ponteiro de saída, e o ponteiro base da instrução de `store` corresponde ao ponteiro definido como entrada. Caso tais condições sejam satisfeitas, o conjunto de laços analisados compõem uma computação estêncil iterativa. Ao final da detecção, é gerado uma estrutura de dados contendo todas as informações obtidas que caracterizam a computação estêncil. Esta estrutura será acessada pelo passe de otimização responsável pela geração de código CUDA.

5. Geração de Código CUDA

Uma vez que as informações sobre o padrão de vizinhança da computação estêncil já foram obtidas pelo passe descrito na seção anterior, é executado um segundo passe LLVM realizando a geração de código na linguagem CUDA. Na implementação atual, limitamos o laço de computação mais interno para conter apenas um bloco básico sem chamada de função. Essa limitação simplifica ambas a detecção do padrão estêncil como a geração de código CUDA, mas representa uma grande classe das aplicações no padrão estêncil. Além disso, limitamos a geração de código somente para estênceis 3D.

O passe de geração de código em CUDA realiza a extração do código de computação do laço mais interno, criando uma nova função *kernel* para execução em GPU. Neste *kernel*, as variáveis de indução dos laços de computação são transformados em variáveis que indexam o acesso dos dados na memória da GPU. Além disso, é criada uma função que gerencia a transferência de dados para a GPU e é gerado um laço, correspondente ao laço de iterações da computação estêncil, que realiza chamadas para a execução do *kernel* em GPU.

Realizamos duas gerações de códigos distintas, uma simples e uma com otimizações. A geração de código simples realiza uma tradução direta do bloco básico em LLVM para o código CUDA. Para a geração de código mais otimizada, foi escolhida uma otimização que apresentou um melhor resultado geral em trabalhos anteriores da literatura [Maruyama and Aoki 2014, Nasciutti and Panetta 2016]. Nesta otimização, é realizado uma iteração sobre o eixo Z do estêncil 3D, armazenando os valores deste eixo na memória *cache* somente leitura, e é explorado a localidade de dados do eixo XY, armazenando estes valores na memória compartilhada. A seleção dos valores que serão armazenadas na memória compartilhada (eixo XY) ou na *cache* somente leitura (eixo Z) é feita a partir dos deslocamentos em relação ao elemento central do estêncil, previamente obtidos no passe de detecção. Adicionalmente, é realizado um bloqueio temporal de duas iterações, realizadas internamente na GPU, a fim de diminuir a comunicação entre a *cache* L2 e a memória global, porém ao custo da realização de computações redundantes.

6. Experimentos

Nesta seção validamos o PSkelCC em relação à sua detecção do padrão estêncil e o desempenho obtidos pelos códigos gerados automaticamente em relação aos códigos otimizados escritos manualmente e aos códigos gerados automaticamente pelo compilador do estado da arte PPCG, na versão 0.07. O PSkelCC foi desenvolvido utilizando LLVM 3.7. Todos os códigos gerados foram compilados utilizando-se CUDA 8.0. Os experimentos foram realizadas em uma GPU Tesla K40c, composta por 2.880 núcleos CUDA, 12GB de memória memória global (DRAM), 1.5MB de memória *cache* L2, 48KB de *cache* somente leitura, 48KB de memória compartilhada e 16KB de *cache* L1.

6.1. Detecção de Códigos Estêncil

Utilizamos um conjunto de códigos do *benchmark* Polybench para avaliar a detecção do padrão estêncil e selecionamos os seguintes códigos: `conv2d`, `conv3d`, `jacobi1d` e `jacobi2d`. Além disso, foram implementadas versões sequenciais do estêncil laplaciano 3D com diferentes vizinhanças. Todos estes códigos estêncil foram detectados corretamente pelo PSkelCC. Todos os demais código do *benchmark* pertencentes a outro domínio também foram analisados corretamente, como não sendo pertencentes ao padrão estêncil. Estes códigos foram os seguintes: `2mm`, `3mm`, `atax`, `bicg`, `correlation`, `covariance`, `gemm`, `gemsumv`, `mvt`, `syrk` e `syr2k`. Os códigos estêncil `adi`, `fdtd-2d`, `fdtd-ampl e seidel` não foram analisados pois, apesar de possuirem o padrão de vizinhança estêncil, possuem um padrão de computação que, no momento, não é detectado pelo PSkelCC.

6.2. Desempenho

Nesta seção apresentamos o desempenho das implementações de um estêncil laplaciano 3D. Foram realizadas diferentes implementações variando a dimensão do raio da vizinhança do estêncil de 1 a 5, sobre cada eixo do espaço 3D, totalizando uma vizinhança de 7, 13, 19, 25 e 31 elementos, respectivamente. Cada um desses estênceis foram executados com tamanhos de 64^3, 128^3 e 256^3, com 50 iterações cada, em um total de 10 execuções. Foi comparado o desempenho do código gerado pelo PSkelCC sem otimização (pskelcc-base) e com uma otimização utilizando memória cache somente leitura, internalização do eixo Z e bloqueio temporal utilizando a memória compartilhada (pskelcc-opt-temp) em relação aos códigos gerados pelo compilador PPCG de maneira completamente automática (ppcg-base) e configurado manualmente para um melhor desempenho (ppcg-opt), além de códigos escritos manualmente sem otimização (manual-base), somente com otimização do uso da memória cache somente leitura e internalização do eixo Z (manual-opt) e com uso de memória cache somente leitura, internalização do eixo Z e bloqueio temporal utilizando a memória compartilhada (manual-opt-temp).

A Figura 4 apresenta o desempenho em GFlop/s de cada uma das implementações apresentadas anteriormente. Além disso, é apresentado uma média geométrica das execuções das diferentes vizinhanças. A Tabela 1 apresenta a ocupação medida através de *profiling* para cada umas das implementações. A ocupação representa a relação entre a média de *warps* ativos por ciclo e o número máximo de *warps* suportados pela GPU. Ela está diretamente ligada a eficiência obtida na GPU, isto é, ela indica se os recursos da GPU estão sendo utilizados de maneira eficiente.

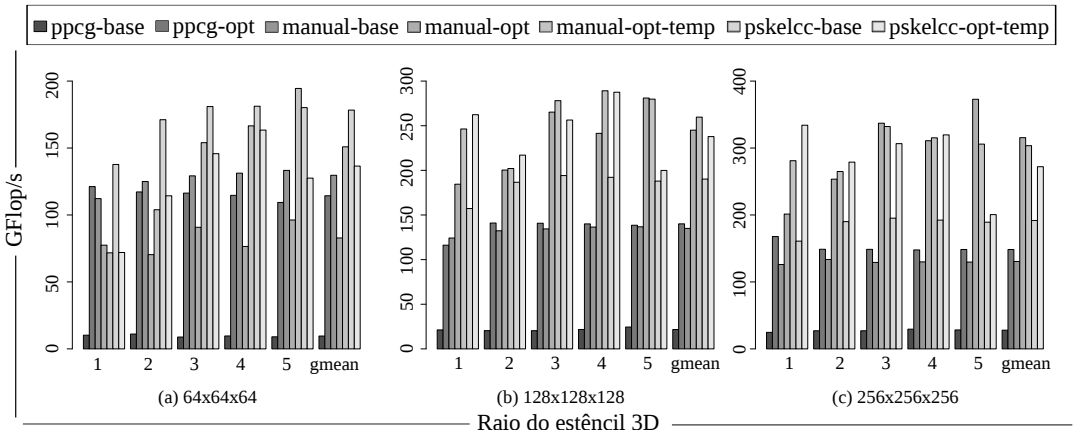

Figura 4. Desempenho de estênceis 3D.

A implementação base do PPCG foi a que obteve o pior desempenho para todos os casos. O baixo desempenho se deve ao uso de condicionais no código-fonte gerado, ocasionando divergências e diminuindo a eficiência da execução das *warps* (eficiência média de 92%) e a ocupação da GPU. Por outro lado, as implementações manuais e as geradas pelo PSkelCC possuem 100% de eficiência, já que não possuem condicionais. No geral, o pskelcc-base obteve um *speedup* médio de 10.98x sobre o ppcg-base.

A implementação ppcg-opt obteve um desempenho consideravelmente melhor que o ppcg-base, com um *speedup* médio de 11.92x. No entanto, o pskelcc-base apresenta uma maior ocupação em todos os casos, resultando em um *speedup* médio de 1.47x sobre o ppcg-opt (máximo de 1.64x). Por outro lado, a implementação otimizada do PSkelCC apresenta uma baixa ocupação, e obtem um menor desempenho em relação ao ppcg-opt para estênceis de raio 1 e 2 com tamanho de 64^3. No geral, a implementação pskelcc-opt-temp apresenta um *speedup* médio de 1.59x (máximo de 2.25x) em relação ao ppgc-temp.

Em relação às implementações manuais, o pskelcc-base obteve um desempenho melhor que o manual-base para todos os casos, com um *speedup* médio de 1.38x (máximo de 1.48x), apesar de ambos apresentarem ocupações similares. Como ambas as implementações se diferem apenas na ordem das instruções e na forma de indexação dos dados em memória, esta melhoria de desempenho se dá, principalmente, pela alocação de registradores realizada pelo compilador do CUDA. Ao analisar a quantidade de registradores alocados para cada *kernel*, observamos que a quantidade alocada para os códigos gerados pelo PSkelCC é menor, justificando o aumento de desempenho. A implementação pskel-opt-temp, que utilizada bloqueio temporal, somente não alcançou um desempenho melhor que o manual-base em três casos, para os estênceis de raios 1, 2 e 5, com tamanho 64^3. Um dos motivos para isto é que, por este tamanho ser relativamente pequeno, o uso das computações redundantes nestes estênceis não apresentou benefício em relação à diminuição dos acessos à memória global. Para o estêncil de raio 5, ao analisar a quantidade de registradores alocados ao kernel, observamos que foram alocados um número alto de registradores, que degrada o desempenho, além de diminuir a ocupação. Para os demais casos, é alcançado um *speedup* médio de 1.83x (máximo de 2.65x).

Ao compararmos o desempenho dos códigos manuais otimizados, observa-se que

Implementação	Ocupação Medida da GPU														
	1			2			3			4			5		
	64^3	128^3	256^3	64^3	128^3	256^3	64^3	128^3	256^3	64^3	128^3	256^3	64^3	128^3	256^3
ppcg-base	0.23	0.27	0.48	0.22	0.26	0.47	0.21	0.25	0.47	0.19	0.24	0.46	0.18	0.24	0.45
ppcg-opt	0.62	0.68	0.70	0.43	0.45	0.48	0.43	0.45	0.48	0.41	0.45	0.48	0.40	0.45	0.47
manual-base	0.84	0.84	0.83	0.87	0.85	0.85	0.87	0.87	0.87	0.89	0.89	0.89	0.68	0.68	0.68
manual-opt	0.49	0.53	0.72	0.24	0.53	0.72	0.24	0.53	0.72	0.24	0.52	0.72	0.24	0.53	0.73
manual-opt-temp	0.24	0.60	0.86	0.24	0.41	0.48	0.24	0.41	0.48	0.26	0.46	0.49	0.34	0.47	0.48
pskelcc-base	0.83	0.84	0.84	0.84	0.85	0.85	0.85	0.86	0.86	0.86	0.87	0.86	0.87	0.86	0.86
pskelcc-opt-temp	0.24	0.59	0.86	0.24	0.41	0.48	0.24	0.42	0.49	0.26	0.45	0.49	0.24	0.24	0.24

Tabela 1. Ocupação da GPU para diferentes raios de vizinhanças e tamanhos.

o uso do bloqueio temporal não apresentou melhoria em 4 dos 15 casos, para os estênceis de raio 1 com tamanho 64^3, raio 3 com tamanho 256^3 e raio 5 com tamanhos 128^3 e 256^3. No entanto, para os demais casos, esta otimização apresenta um *speedup* médio de 1.35x (máximo de 2.17x) sobre as implementações otimizadas sem bloqueio temporal. Ao compararmos com as implementações não otimizadas, o uso do bloqueio temporal, similar ao código gerado pelo PSkelCC, somente não apresenta melhoria para os casos com raios 1 e 2, com tamanho 64^3. Para os demais casos, um speedup médio de 1.9x (máximo de 2.57x) é alcançado.

Ao comparar o desempenho da implementação otimizada gerada pelo PSkelCC em relação à implementação manual otimizada com bloqueio temporal, obtêm-se um melhor desempenho em 7 dos 15 casos, com um *speedup* médio de 1.06x (máximo de 1.19x). Apesar de ambos os códigos implementerem o mesmo algoritmo de bloqueio temporal, certas diferenças na ordem de instruções e na maneira de acesso a memória fazem com que estes códigos tenham uma alocação de registradores diferentes entre si, similar ao que ocorre nas implementações base. Para os casos em que há melhoria de desempenho, foram alocados menos registradores e, nos casos em que há queda de desempenho, foram alocados mais registradores. Este aspecto teve um maior impacto no estêncil de raio 5, em que os códigos gerados alcançaram entre 65% e 71% do desempenho da implementação otimizada manualmente.

7. Conclusões e Trabalhos Futuros

Neste trabalho apresentamos uma ferramenta para detecção e geração automática de códigos estênceis para GPUs a partir de análises estáticas no código-fonte sequencial. A ferramenta proposta foi capaz de identificar corretamente código do padrão estêncil no conjunto de aplicações do benchmark Polybench, além de gerar códigos estênceis com desempenho até $2.25\times$ sobre o compilador do estado da arte PPGC e com desempenho similar ou superior à grande parte dos códigos gerados manualmente.

Trabalhos futuros incluem investigar os aspectos da geração de código que impactam na alocação de registradores e no desempenho, além de incluir e avaliar a geração de código para estênceis 1D, 2D e miniaplicações. Pode-se citar também a inclusão de análises de *aliasing* de ponteiros, dependência de dados e verificações para casos particulares de determinados estênceis. Por fim, pode-se incluir um modelo de custo para o uso da hierarquia de memória da GPU.

Referências

Bachmann, O., Wang, P. S., and Zima, E. V. (1994). Chains of recurrences: a method to expedite the evaluation of closed-form functions. In *Int. Symp. on Symbolic and Algebraic Computation*, pages 242–249.

Bernstein, A. J. (1966). Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763.

Christen, M., Schenk, O., and Cui, Y. (2012). PATUS for convenient high-performance stencils: Evaluation in earthquake simulations. In *Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 11:1–11:10.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490.

Holewinski, J., Pouchet, L.-N., and Sadayappan, P. (2012). High-performance code generation for stencil computations on GPU architectures. In *ACM Int. Conf. on Supercomputing (ICS)*, pages 311–320.

Khan, M., Basu, P., Rudy, G., Hall, M., Chen, C., and Chame, J. (2013). A script-based autotuning compiler system to generate high-performance CUDA code. *ACM Trans. Archit. Code Optim.*, 9(4):31:1–31:25.

Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *Int. Symp. on Code Generation and Optimization*.

Maruyama, N. and Aoki, T. (2014). Optimizing stencil computations for NVIDIA Kepler GPUs. In *Int. Workshop on High-Performance Stencil Computations*, pages 89–95.

Mendonca, G. S. D., Guimaraes, B. C. F., Alves, P. R. O., Pereira, M. M., Araujo, G., and Pereira, F. M. Q. (2017). Dawncc: Automatic annotation for data parallelism and offloading. *ACM Trans. Archit. Code Optim.*, 14:13:1–13:25.

Meng, J. and Skadron, K. (2011). A performance study for iterative stencil loops on GPUs with ghost zone optimizations. *Int. Journal of Parallel Programming*, 39(1):115–142.

Nasciutti, T. C. and Panetta, J. (2016). Impacto da arquitetura de memória de gpgpus na velocidade da computação de estênceis. In *XVII Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD-SSC)*, pages 1–12.

Nugteren, C. and Corporaal, H. (2014). Bones: An automatic skeleton-based c-to-cuda compiler for gpus. *ACM Trans. Archit. Code Optim.*, 11(4):35:1–35:25.

Pereira, A. D., Castro, M., Dantas, M. A. R., Rocha, R. C. O., and Góes, L. F. W. (2017). Extending OpenACC for efficient stencil code generation and execution by skeleton frameworks. In *Int. Conf. on High Perf. Comp. Simulation (HPCS)*, pages 719–726.

Pereira, A. D., Ramos, L., and Góes, L. F. W. (2015). PSkel: A stencil programming framework for CPU-GPU systems. *Concur. and Comp.: Practice and Experience*, 27(17):4938–4953.

Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., and Catthoor, F. (2013). Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23.