# COLOCAÇÃO AUTOMÁTICA DE COMPUTAÇÃO EM HARDWARE HETEROGÊNEO

KÉZIA CORRÊA ANDRADE MOREIRA

# COLOCAÇÃO AUTOMÁTICA DE COMPUTAÇÃO EM HARDWARE HETEROGÊNEO

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

ORIENTADOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte, MG

Maio de 2015

.

**Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG**

Moreira, Kézia Corrêa Andrade.

M838c Colocação automática de computação em hardware heterogêneo. / Kézia Corrêa Andrade Moreira – Belo Horizonte, 2015.
xx, 58 f.: il.; 29 cm.

Dissertação (mestrado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Fernando Magno Quintão Pereira.

1. Computação – Teses. 2. Compiladores (Computadores). 3. Linguagem de programação (Computadores) 4. Análise estática. I. Orientador. II. Título.

CDU 519.6*33(043)

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

# FOLHA DE APROVAÇÃO

Colocação Automática de Computação em Hardware Heterogêneo

**KÉZIA CORRÊA ANDRADE MOREIRA**

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

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

PROFA. ANOLAN YAMILÉ MILANÉS BARRIENTOS
Departamento de Computação - CEFET

PROF. LUIS CARLOS ERPEN DE BONA
Departamento de Informática - UFPR

Belo Horizonte, 24 de junho de 2016.

*Dedico essa dissertação à pessoas mais presentes em minha vida:*
*Aos meus pais pelo amor incondicional.*
*Ao meu esposo por sempre estar ao meu lado.*
*Aos meus irmãos por todo apoio e incentivo.*

# Agradecimentos

Agradeço primeiramente a Deus, por ser a razão do meu viver, e por ter colocado pessoas tão especiais em minha vida. Por ser o meu suporte interior nos momentos mais difíceis.

Agradeço aos meus pais, Clemilda e Jozué, por me apoiarem em todas as minhas decisões, por todo investimento e incentivo em minha educação. Agradeço por sempre terem acreditado em minha capacidade o que me motivou e me fortaleceu e me fez tentar. Por todas as orações a meu favor e pelo amor incondicional.

Agradeço ao meu querido esposo Lucas por todo amor, e por estar ao meu lado nos momentos mais difíceis e mais felizes. Por me compreender nos momentos de ausência e por todo o incentivo que não me deixou desistir.

Aos meus irmãos, Kênia e Tiago, meu agradecimento especial, pois, a seu modo, sempre se orgulharam de mim e confiaram em meu trabalho. Obrigada pela confiança, apoio e amizade.

Agradeço ao meu sobrinho Nick e ao meu afilhado João Gabriel por trazerem alegria nos momentos de dificuldade.

Queria fazer um agradecimento especial ao meu orientador Fernando por todo o conhecimento compartilhado, pelas idéias oferecidas sem as quais seriam impossível desenvolver este trabalho. Agradeço por toda oportunidade e conselhos dados.

Agradeço aos meus amigos do DESIS que me incentivaram a seguir meus estudos e me fizeram acreditar que é possível fazer amigos no trabalho. Agradeço ao pessoal do laboratório por toda ajuda e por deixar o ambiente de estudo e trabalho mais agradável.

Também não poderia de deixar de agradecer a minha sogra Meire por toda palavra de incentivo, e todas as oraçoes à meu favor.

Gostaria de agradecer à todos os meus amigos e famíliares e em especial à Cinthia, Flávia e Maíra que me acompanharam de perto nesses dois anos sempre com palavras de apoio, incentivo e amizade.

Enfim gostaria de agradecer a todos que de alguma forma fizeram parte da minha vida no decorrer deste trabalho, e os quais não citei o nome neste trabalho.

# Resumo

As placas de processamento gráfico (GPUs) revolucionaram a programação de alto desempenho, pois elas reduziram o custo do *hardware* paralelo. A programação desses dispositivos, contudo, é um desafio, pois programadores ainda não são treinados para escrever código que coordene a atuação simultânea de milhares de *threads*. A fim de lidar com esse problema, a indústria e a academia vem introduzindo sistemas de anotações, como OpenMP 4.0, OpenSs e OpenACC, que permitem indicar quais partes de um programa C ou Fortran deveriam executar em GPU ou em CPU. Essa abordagem possui duas vantagens. Primeiro, ela mantém programadores em sua zona de conforto: eles podem continuar desenvolvendo código em sua linguagem de programação preferida. Segundo, as anotações protegem programadores de minúcias do *hardware* paralelo, uma vez que elas passam a tarefa de paralelizar programas para o gerador de código.

Apesar de a inserção de diretivas no código esconder detalhes de *hardware*, ela não resolve todos os problemas do programador: ele ainda precisa identificar quando será vantajoso executar um dado trecho de código na GPU e quando não. Nesse contexto, o objetivo desta dissertação é apresentar uma solução para tal problema. Foram projetadas, implementadas e testadas técnicas para identificar de forma automática quais porções do código devem executar na GPU. Para isso foram utilizadas informações de dependência, *layout* de memória, intensidade aritmética e fluxo de controle.

Foram criadas um conjunto de análises estáticas de código que realizam três tarefas: (*i*) identificar quais laços são paralelizáveis; (*ii*) inserir anotações para copiar dados entre a CPU e a GPU; (*iii*) 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. A plataforma que ora é apresentada foi implementada sobre dois compiladores. As análises foram feitas sobre a infra-estrutura de compilação disponível em LLVM. A geração de código paralelo a partir de programas anotados é feita por PGCC.

A abordagem desenvolvida, até o presente momento, é totalmente estática: é de-

cidido onde cada função deve executar durante a compilação do programa, e tal decisão não muda durante a sua execução. Outro benefício deste arcabouço é que ele é completamente automático, ou seja, não possui nenhuma intervenção do programador(a). Os programas produzidos – de forma totalmente automática – obtiveram *speedups* de até 121x.

# Abstract

Graphics processing Units (GPUs) have revolutionized high performance programming. They reduced the cost of parallel hardware, however, programming these devices is still a challenge. Programmers are not able (yet) to write code to coordinate the simultaneous performance of thousands of threads. To deal with this problem, the industry and the academia have introduced annotation systems. Examples of those systems are OpenMP 4.0, OpenSS and OpenACC, which allow developers to indicate which parts of a C or Fortran program should perform on GPU or CPU. This approach has two advantages. First, it lets programmers to obtain the benefits of the parallel hardware while coding in their preferred programming languages. Second, the annotations protect programmers from details of the parallel hardware, once they pass the task of parallelizing programs for the code generator.

The inclusion of pragmas in the code to hide details of the hardware does not solve all the problems that developers face when programming GPUs: they still need to identify when it will be advantageous to run a given piece of code on the GPU. In this context, the objective of this dissertation is to present a solution to such a problem. It was designed, implemented and tested techniques to automatically identify which portions of the code must run on the GPU. For this, we used dependency information, memory layout and control flow.

Were created a set of static analysis that performs three tasks: (*i*) identify which loops are parallelizable; (*ii*) insert annotations to copy data between the CPU and the GPU; (*iii*) estimate which loops, once tagged as parallels, are most likely to lead to performance gains. These tasks are totally automatic, are carried out without any user intervention. The platform that is presented has been implemented on two compilers. The analyses were built on top of the infrastructure available in LLVM. The parallel code generation, from annotated programs, is made by PGCC.

The approach that we have developed is completely static: we decide where each function must run during the compilation of the program. This decision does not rely on any runtime system such as a middleware, or special computer architecture hooks.

Another benefit of this framework is that it is completely automatic, i.e. it does not require any intervention from the programmer. As a result, programs that we produce — automatically -– can be up to 121x faster than their original versions.

# Lista de Figuras

# Lista de Tabelas

# Sumário

# Capítulo 1

# Introdução

## 1.1 Motivação

Unidades de Processamento Gráfico (GPUs) têm-se tornado cada vez mais populares no mundo da computação de alto desempenho. Apesar de terem sido criadas para processamento gráfico, o uso de GPUs para computação de propósitos gerais (GPGPU - *General Purpose on Graphics Processing Units*) já está sendo feito há algum tempo [Harris et al., 2002]. Elas proveem uma plataforma simples, barata e eficiente para a execução de programas paralelos [Nickolls & Dally, 2010]. Este alto desempenho ocorre devido sua arquitetura ter sido projetada com o foco em processamento gráfico, podendo conter milhares de núcleos. Na Figura 1.1 pode ser visualizado o comparativo de pico teórico de desempenho entre GPUs e CPUs convencionais. GPUs possui o desempenho na ordem de 3000 vezes a mais que CPUs convecionais.

Esta diferença ocorre devido à GPUs se dedicarem muito mais ao processamento de dados do que ao controle e armazenamento em memória. Como pode ser visto na Figura 1.2, CPUs possuem alguns núcleos otimizados para processamento de série seqüencial. Enquanto uma GPU possui milhares de núcleos menores, com ALUs (Unidades Lógicas e Aritméticas) simplificadas, porém mais eficientes, projetados para realizar várias operações sobre o mesmo dado de forma simultânea.

Desde o lançamento de CUDA (*Compute Unified Device Architecture*), no início de 2006 [Garland, 2008], uma enorme gama de padrões de programação e algoritmos têm sido projetados para executar nesse ambiente. Isso se justifica pela grande versatilidade das GPUs, que podem ser usadas para resolver problemas tão distintos quanto sequenciamento genético [Sandes & de Melo, 2010], roteamento IP [Mu et al., 2010], álgebra linear [Zhang et al., 2010] e análise de programas [Prabhu et al., 2011].

Entretanto, escrever programas para GPUs, usando CUDA, ainda é uma tarefa

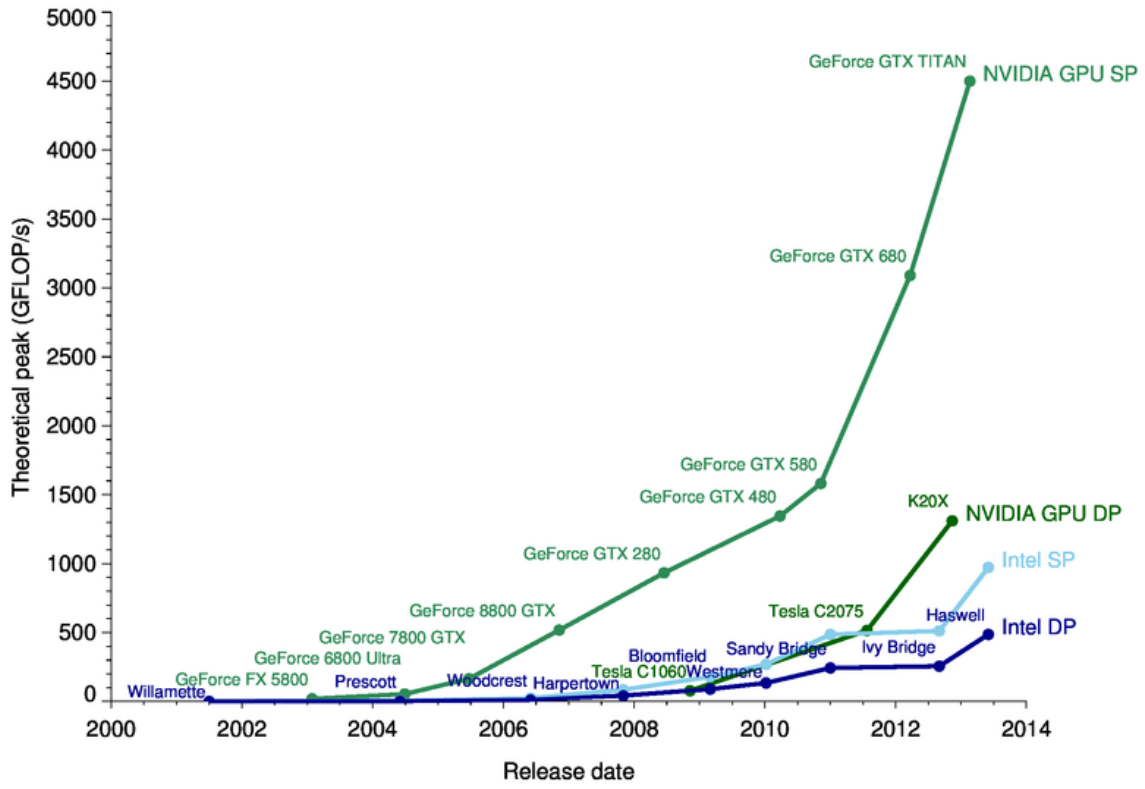


Figura 1.1: Gráfico: Comparativo de pico teórico de desempenho entre GPUs e CPUs convencionais.

difícil, visto que apesar de ser possível programá-las em linguagens como C e C++, o programador ainda precisa pensar em detalhes do *hardware* subjacente. Na Figura 1.3 é mostrado uma função de um programa escrito em CUDA, e como pode ser visto por essa porção do código o modelo de execução paralela presente em GPU não está explícito. O programador precisa conhecer o programa sequencial e ser capaz de reescrevê-lo para execução em milhares de *threads*.

Recentemente, tanto a indústria quanto a academia tem somado forças para facilitar a programação de GPUs. Um dos esforços nesse sentido é o desenvolvimento de sistemas de anotação de código, os quais, têm sido amplamente utilizados na comunidade científica e na indústria [Planas et al., 2015]. Esses sistemas foram criados com o intuito de aumentar a produtividade dos programadores. Desenvolvedores podem escrever código sem preocupar-se com a sintaxe para criação de *threads* ou a cópia de dados entre diferentes processadores. Ainda assim, programadores 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 [Jaeger et al., 2015], OpenSs [Meenderinck & Juurlink, 2011] e Ope-

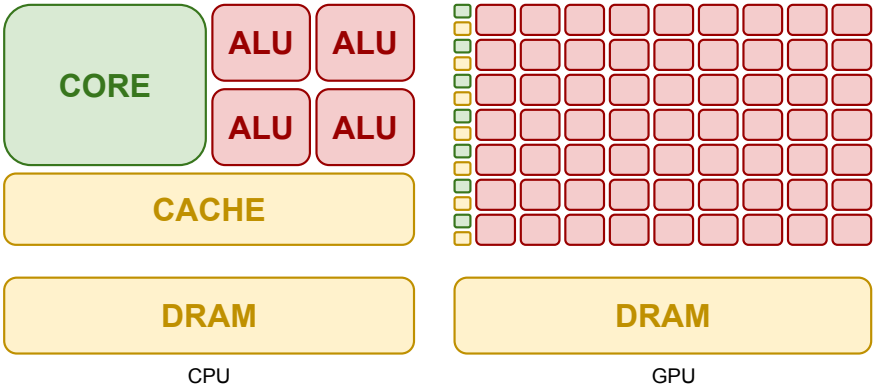


Figura 1.2: Comparação da arquitetura interna de um CPU e uma GPU. Uma CPU possui múltiplos núcleos (*cores*) e ALU (Unidades Lógicas Aritméticas), porém GPUs atuais possuem milhares.

```
__global__ void MatMul(Matrix A, Matrix B, Matrix C) {
  float val = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  if(row > A.height || col > B.width)
    return;
  for (int e = 0; e < A.width; ++e)
    val += A.elements[row*A.width+e] * B.elements[e*B.
width+col];
  C.elements[row*C.width+col] = val;
}
```

Figura 1.3: Função *kernel* escrita em CUDA para realizar a multiplicação de matrizes.

nACC [Standard, 2013], sendo este último um dos membros mais proeminentes desta família. De forma resumida, OpenACC permite ao programador indicar quais laços devem ser executados no *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 [Wienke et al., 2012].

## 1.2 Problema

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 [Bai et al., 2015]. Do ponto de vista de corretude, é deixada ao programador a tarefa de identificar quais laços são paralelizáveis. Laços são ditos paralelizáveis quando não apresentam condições de corrida, uma situação que pode ocorrer quando duas ou mais *threads* usam a mesma posição de memória e pelo menos um desses usos é uma escrita [Hennessy & Patterson, 2011; Zhang et al., 2016]. Em seguida, o desenvolvedor

deve avaliar o quão vantajoso seria enviar determinado laço à GPU. Porém para se executar uma instrução deve ser calculado o custo de mover, para a GPU, os dados necessários à computação somado a um número potencialmente alto de divergências no corpo do laço, o que pode provocar um aumento considerável do tempo de execução de um programa [Coutinho et al., 2011]. 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.

## 1.3 Solução Proposta

Na tentativa de retirar do programador o peso de decidir quais trechos de código devem ser movidos para a GPU, foram desenvolvidas várias técnicas para automatizar essa decisão. Foi criado um conjunto de análises estáticas de código que realiza três tarefas: (*i*) identificar quais laços são paralelizáveis; (*ii*) estimar quais laços, uma vez marcados como paralelos, possuem maior probabilidade de levar a ganhos de desempenho; (*iii*) inserir anotações para copiar dados entre a CPU e a GPU. Essas tarefas são realizadas sem qualquer intervenção do usuário. A plataforma que ora é apresentada foi implementada sobre dois compiladores. As análises detalhadas no capítulo 3 foram feitas sobre a infraestrutura de compilação disponível em LLVM [Lattner & Adve, 2004]. A geração de código paralelo a partir de programas anotados é feita por PGCC [Ghike et al., 2014]. LLVM é uma infraestrutura 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.

A abordagem já desenvolvida, até o presente momento, é totalmente estática: é decidido onde cada laço deve executar durante a compilação do programa, e tal decisão não muda durante a sua execução. Outro benefício deste arcabouço é que ele é completamente automático, ou seja, não possui nenhuma intervenção do programador(a). Como consequência, os programas produzidos – de forma totalmente automática – obtiveram *speedups* de até 121x comparados ao programa executado na CPU.

## 1.4 Contribuições

O objetivo dessa dissertação é decidir, em tempo de compilação, em que parte de um *hardware* heterogêneo os diferentes laços de um programa devem ser executados. A

ferramenta proposta é totalmente voltada para laços [1]. É esperado que este estudo auxilie programadores que visam ter um ganho de performance em programas que possam ser paralelizados. Para alcançar tal objetivo foram criadas e estão publicamente disponíveis as seguintes análises:

- Uma análise capaz de identificar quais os laços de um programa que podem ser paralelizados;
- Uma análise que infere quais dados devem ser enviados para a GPU, e insere diretivas OpenACC no código fonte para, dessa forma, informar ao compilador quais laços devem ser paralelizados;
- Dois algoritmos para estimar o impacto da "divergência de controle" e da "divergência de acesso a memória" no desempenho do código para GPU que foi gerado. Se o impacto for muito grande, ou melhor, se a percentagem de instruções e/ou acessos de memória divergentes for acima de 80% o código não será enviado para a GPU.

Como contribuição principal dessa dissertação foi feito um arcabouço que engloba todas as análises citadas acima. Esse arcabouço é totalmente estático, ou seja, é realizado durante a compilação do programa, e é totalmente automático não é necessário a intervenção do usuário. Essa ferramenta é capaz de, a partir de um código fonte totalmente sequencial, gerar um código anotado com diretivas OpenACC, capaz de ser compilado e executado em qualquer compilador para este fim.

## 1.5 Publicações e software

Como resultado desse trabalho foram submetidos três artigos. "Etino: Colocação Automática de Computação em Hardware Heterogêneo" foi publicado no Simpósio Brasileiro de Linguagens de Programação (SBLP 2015) [do Couto Teixeira et al., 2015a] e na seção de ferramentas do Congresso Brasileiro de Software (*CBSoft Tools*) [do Couto Teixeira et al., 2015b]. Esses trabalhos propõem um anotador semiautomático que utiliza informações de perfilamento para decidir quais laços devem ser enviados para a GPU, este foi detalhado na seção 3.2. E foi submetido para o SBLP de 2016 um artigo com o título "Paralelização Automática de código com diretivas OpenACC", o qual propõe um anotador totalmente estático e automático, esse foi detalhado na seção 3.4 e foi a base para essa dissertação.

[1] As análises implementadas foram feitas especificamente para laços *for*, laços *while* ainda não são cobertos

**Software** O anotador 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 diretivas OpenACC. Cabe ao usuário compilar esse programa de saída, usando para tanto um compilador que leia diretivas OpenACC.

## 1.6 Estrutura do trabalho

Esta dissertação está organizada em cinco capítulos. No presente capítulo é feita uma rápida introdução sobre o tema a ser abordado e a solução encontrada. O segundo capítulo possui uma revisão bibliográfica, onde são tratados os principais trabalhos e conceitos relacionados a paralelização automática de código. No Capítulo 2 também são discutidos temas como modelos de custos e conceitos relacionados e essenciais para a compreensão do anotador proposto.

O Capítulo 3 trata da solução proposta neste trabalho. Neste capítulo são apresentadas soluções similares presentes na literatura e é descrita uma solução preliminar que foi a motivação do anotador descrito neste capítulo. No Capítulo 4 são apresentadas todas as ferramentas utilizadas para o desenvolvimento do projeto, os *benchmarks* e o ambiente de testes bem como os resultados obtidos. No Capítulo 5 procura-se avaliar o projeto como um todo seguindo critérios como generalidade, facilidade de uso e melhoria de desempenho. Além disso, tal capítulo apresenta limitações do anotador e possíveis linhas de pesquisa que podem ser desenvolvidas a partir desta dissertação.

# Capítulo 2

# Revisão Bibliográfica

Neste capítulo é apresentado uma revisão bibliográfica acerca dos temas abordados neste trabalho. Foi realizada uma pesquisa de temas pertinentes à compiladores os quais são necessários para o entendimento e desenvolvimento das análises presentes nesse trabalho. É detalhado como se realiza a análise para inferir paralelismo em um programa e são citadas quais as análises disponíveis hoje na indústria e na comunidade científica. São apresentados sistemas de anotação de código e é explicado o sistema OpenACC de maneira mais detalhada visto que é o sistema utilizado no presente trabalho. É abordado o que são análises de divergências e o porquê de sua importância quando o assunto é paralelização de código. É detalhado como ocorre divergência de controle e de memória em um código paralelo.

## 2.1 Compilação

Conforme Aho *et al.* [Aho et al., 1986] um compilador é um programa capaz de, a partir de um código fonte escrito em uma linguagem de programação, criar um programa semanticamente equivalente em linguagem de máquina. Outro papel importante do compilador é relatar erros detectados no código fonte durante o processo de tradução. Um compilador possui três[1] etapas principais para tradução de um programa: (*i*) o *front-end*, onde a análise léxica, a análise sintática e a análise semântica são realizadas; (*ii*) o *middle-end*, onde as otimizações e análises são realizadas; e (*iii*) o *back-end*, onde o código de máquina finalmente é gerado. Nesta dissertação é trabalhado na etapa do *middle-end*, ou seja, as análises foram criadas para otimização do programa

[1]Alguns autores subdividem as etapas de processamento em um compilador apenas em duas: análise e síntese ou *front-end* e *back-end*, porém nessa dissertação utilizamos a abordagem que quebra a etapa de análise em duas visto que as análises aqui desenvolvidas são análises de otimização de código e não para geração de código intermediário.

fonte. Para a compreensão completa dessas análises nessa seção será abordado alguns conceitos primordiais em compiladores.

### 2.1.1 Grafo de fluxo de Controle - CFG

Após a geração do código intermediário, onde é feita a representação do programa de forma a ser, de acordo com Aho *et al.* [Aho et al., 1986], facilmente produzida e traduzida para a máquina alvo, os compiladores denotam esses programas (com instruções de três endereços) como um CFG (Control Flow Graph) [2]. Na Figura 2.1 é mostrado um exemplo simples de como o CFG é criado: a função 2.1 (a) é compilada e como resultado o CFG 2.1 (b) é criado. Na figura é apresentado um exemplo ilustrativo de funcionamento, em um compilador real a representação das instruções a serem executadas seria em *byte code*, e os nós do CFG (os blocos básicos) seriam sequências de *byte codes*.

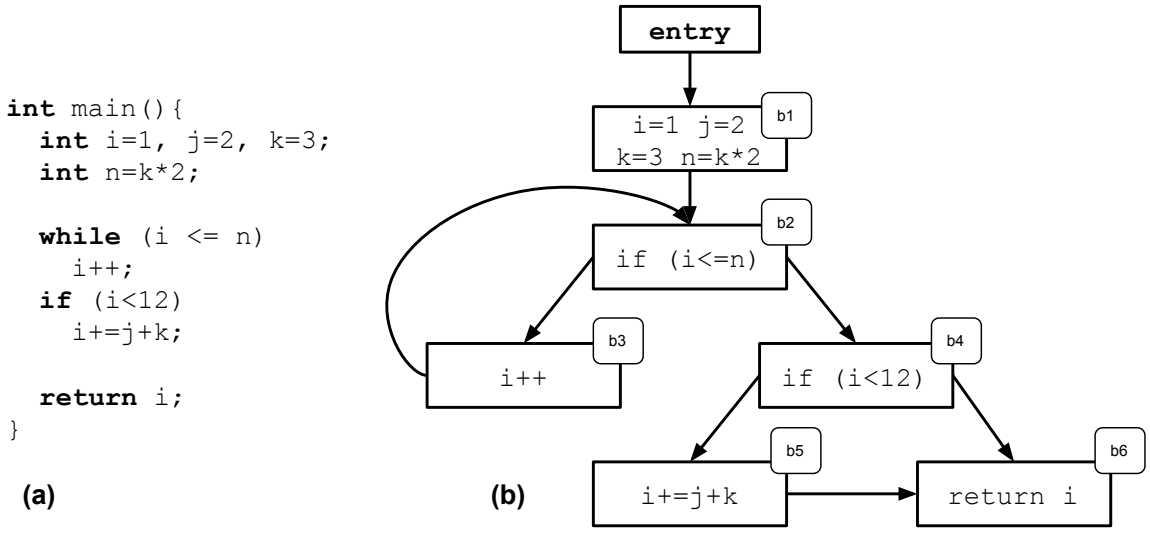


Figura 2.1: (a) Programa de entrada. (b) CFG gerado para ele.

A partir da análise de fluxo de controle [Allen, 1970], a qual determina a ordem de execução das instruções de um programa, é criado o CFG [Allen, 1970]. O CFG determina todos os possíveis caminhos de execução de um programa. Ele é um Grafo

$$G =< N, E >$$

[2] Em 1970 Frances Allen ganhou o prêmio Turing pelo pioneirismo em contribuição para a teoria e a prática de otimização de técnicas de compilador que lançaram as bases para os modernos compiladores de otimização e execução paralela automática `http://amturing.acm.org/award_winners/allen_1012327.cfm`.

direcionado, onde $N$, os nós do grafo, são os blocos básicos de um programa e $E$, as arestas, respeitam a seguinte regra:

$$(x,y) \in E \begin{cases} \text{(i) a primeira instrução de um bloco básico y segue, imediatamente após, a última instrução do bloco básico x;} \\ \text{(ii) a primeira instrução de y é alvo de uma instrução condicional ou um desvio do bloco básico x;} \\ \text{(iii) a primeira instrução de y é a próxima instrução depois da ultima instrução de x em memória e a ultima instrução de x que não é um desvio;} \end{cases} \tag{2.1}$$

Um grafo de fluxo de controle é utilizado para modelar a transferência de controle em um procedimento. Ele é responsável por facilitar diversas ferramentas de otimização de código. Através dele é possível identificar a alcançabilidade de um determinado trecho de código, podendo dessa forma realizar a remoção de código morto. Se o bloco de saída é inalcançável a partir do bloco de entrada, há algum tipo de laço infinito no código, por exemplo. Enfim, através dessa representação é possível realizar diversas análises estáticas para otimização do código fonte gerado.

### 2.1.2 Bloco Básico

De acordo com Allen [Allen, 1970] um "*bloco básico é uma sequência linear de instruções de um programa que contém apenas um ponto de entrada e um ponto de saída*". Ou seja, um bloco básico é a sequência máxima de instruções onde só é possível entrar pela primeira instrução e sair pela ultima instrução do mesmo, dessa forma não é permitido *jumps* ou *branches* que quebrem a execução do mesmo. Um bloco básico pode conter vários predecessores e vários sucessores, podendo ser ele mesmo o seu próprio sucessor (no caso de um laço, por exemplo). Em um CFG são criados os blocos de entrada (*entry*) os quais não possuem predecessores, e os blocos de saída (*exit*) que nunca terão sucessores.

Para a correta identificação de um bloco básico é preciso primeiramente realizar a identificação da primeira instrução de cada bloco, a qual é chamada de líder. De acordo com Aho *et al.* [Aho et al., 1986] uma instrução é líder se ela possuir pelo menos uma das seguintes características:

- Seja a primeira instrução de três endereços no código intermediário;

- Seja o destino de algum desvio condicional ou incondicional, por exemplo um *jump*;
- Seja a primeira instrução, imediatamente após uma instrução de desvio condicional ou incondicional.

Na Figura 2.2 é exemplificado como o compilador a partir de um código fonte identifica os líderes de cada bloco básico e cria o CFG do mesmo. A Figura 2.2(b) representa o código intermediário correspondente à 2.2(a). Nessa etapa são criados rótulos para determinar o fluxo de execução do programa. Como foi explicado anteriormente, todo rótulo e toda instrução imediatamente após uma instrução de desvio é um líder. Desta forma, é criado o CFG mostrado na Figura 2.2(c) com todos os blocos básicos identificados e com todos os caminhos possíveis de um nó ao outro do grafo.

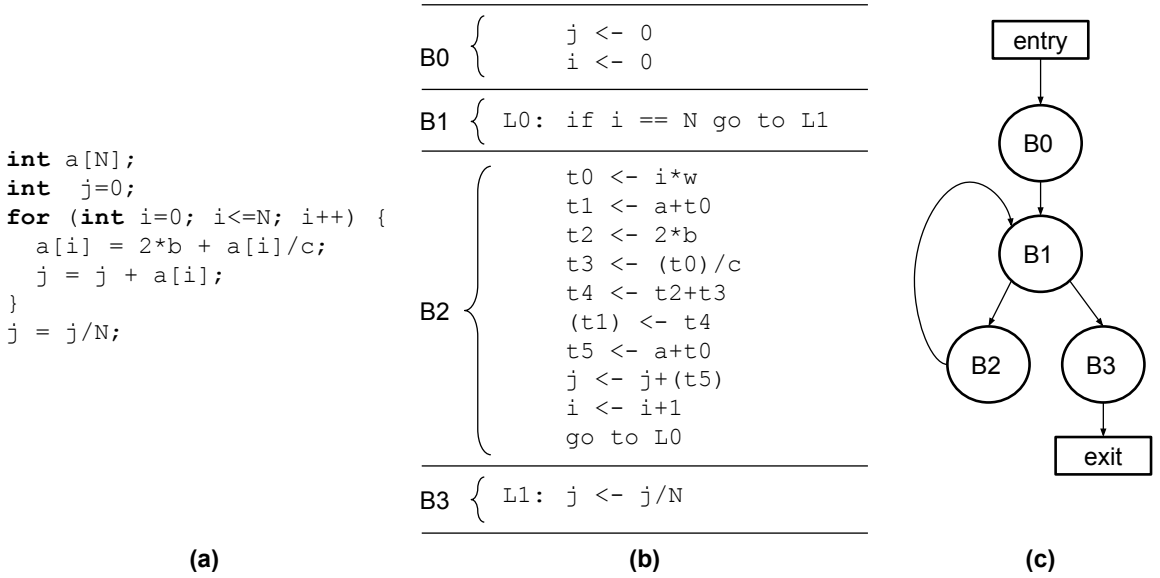


Figura 2.2: (a)Código fonte; (b)Código intermediário, com a definição de cada bloco básico; (c)CFG.

Então, conforme Aho *et al.* [Aho et al., 1986], "para cada líder, seu bloco básico consiste em si mesmo e em todas as instruções até o próximo líder, sem incluí-lo, ou até o fim do programa intermediário." Dessa forma outra gama de otimizações locais é possibilitada, tais como: remoção de sub-expressões comuns, eliminação de código morto, uso de identidades algébricas, dentre outras.

### 2.1.3 Static Single Assignment - SSA

O formato SSA do inglês *Static Single Assignment* [Cytron et al., 1991] é uma representação intermediária criada para simplificar a análise e otimização de código.

Atualmente esse formato é utilizado pela maioria dos compiladores, tais como: LLVM, gcc, Ocelot, Mozilla's, dentre outros. A premissa de um programa no formato SSA é: **cada variável possuir apenas uma definição**, por isso o nome "*Single Assignment*", e toda variável é definida antes do seu uso. Todas as variáveis existentes na representação intermediária (IR) original são subdivididas em versões, ou novas variáveis, normalmente essas novas variáveis são representadas, nos livros, com o nome original seguido de um subscrito que indicaria a ordem que ela aparece no programa. No formato SSA todas as correntes de uso-definição são muito bem definidas e cada uma contêm apenas um elemento. Na Figura 2.3 pode ser visto como o programa mostrado à esquerda 2.3(a) ficará no formato SSA 2.3(b).

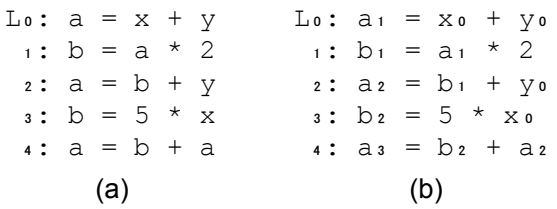


Figura 2.3: (a)Programa em um representação intermediária; (b)Programa convertido para o formato SSA.

Alguns desafios são criados, por se ter apenas uma atribuição estática para cada variável. Quando se pensa em um programa real com diversos laços e desvios, alcançar essa premissa passa a ser complicado. A mesma variável pode ser definida em mais de um caminho no fluxo de controle do programa. No exemplo mostrado na Figura 2.4 a variável $x$ terá valores distintos após a saída da instrução condicional *if*.

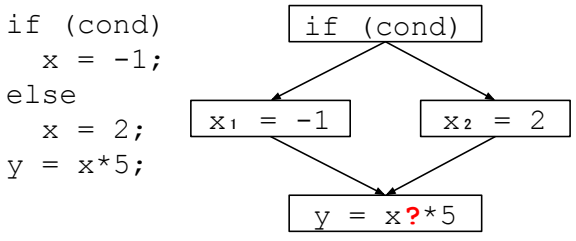


Figura 2.4: Programa com condicionais.

Para resolver este tipo de problema são criadas funções-phi. Funções-$\phi$ são funções inseridas no início de cada bloco básico em que está chegando mais de uma definição de uma variável, ou seja, serão inseridas $N$ funções-$\phi$ com $k$ argumentos, um para cada cópia paralela. Para inserir uma função-$\phi$ para uma variável $b$ em um nó $z$ todas as regras a seguir devem ser satisfeitas:

1. Exista um bloco $x$ que contenha uma definição de $b$;
2. Exista um bloco $y$, onde $y \neq x$, que contenha uma definição de $b$;
3. Exista um caminho não vazio $P_{xz}$, ou seja, uma aresta de $x$ para $z$;
4. Exista um caminho não vazio $P_{yz}$, ou seja, uma aresta de $y$ para $z$;
5. Os caminhos $P_{xz}$ e $P_{yz}$ não tenham nenhum nó em comum além de $z$;
6. O nó $z$ não está presente em nenhum dos caminhos: $P_{xz}$ e $P_{yz}$.

Na Figura 2.5 é mostrado o resultado para o programa mostrado na Figura 2.4. Quando em um determinado nó estão chegando duas definições para uma mesma variável, no exemplo a variável $x_1$ e $x_2$, é inserida uma função-$\phi$, dessa forma é criada mais uma definição para a variável $x$ ($x_3$) e na instrução em que ela é usada será utilizada essa nova definição.

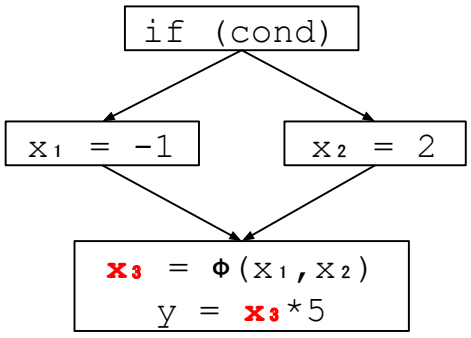


Figura 2.5: Programa no formato SSA exemplificando o uso de funções-phi.

Diversas otimizações foram possibilitadas e facilitadas devido ao uso do formato SSA. Alguns exemplos de otimizações facilitadas são: eliminação de código morto, propagação de constante, alocação de registradores, dentre outras. Nessa dissertação todas as análises são aplicadas no programa após a conversão para o formato SSA.

### 2.1.4 Laços

Laços ou *loops* de acordo com Aho *et al.* [Aho et al., 1986] são construções criadas em linguagens de programação para permitir a repetição de determinado trecho de código, ex: *while*, *do-while*, *for*. Como a maior porção de tempo gasto na execução de um programa é em um laço, é muito importante que o compilador gere um bom código intermediário para eles. Várias otimizações são feitas nesses tipos de construção, a proposta nessa dissertação, por exemplo, é totalmente voltada para laços. Devido a

isto, algumas regras são verificadas para a correta identificação de um laço em um programa:

1. Exista um nó $L$ chamado *entrada do loop* com a propriedade de que nenhum outro nó em $L$ tenha um predecessor fora de $L$, ou seja, todo caminho a partir da entrada do CFG inteiro para qualquer nó em $L$ passa pela entrada do *loop*.
2. Cada nó em $L$ possui um caminho não vazio, completamente dentro de $L$, para a entrada de $L$. [Aho et al., 1986]

Diversos conceitos são necessários para o entendimento de *loops*: dominadores, arestas para trás (*back edges*), ordenação em profundidade, profundidade em um grafo, dentre outros, contudo este não é o foco dessa dissertação, logo não será detalhado cada conceito. O primeiro citado, sendo o primordial para a compreensão de como o compilador representa um laço, é detalhado a seguir:

**Dominadores** De acordo com Aho *et al.* [Aho et al., 1986] "o nó $d$ de um grafo de fluxo *domina* o nó $n$,(...) se todo caminho a partir do nó de entrada do grafo de fluxo para $n$ passar por $d$." Na Figura 2.6(a) é representado o grafo de fluxo e por ele podem-se identificar vários laços, e os dominadores de cada nó. O nó 1 domina todos os nós do grafo, o nó 2 domina os nós 3,4, 5, 6, 7, 8 e 9; o mesmo raciocínio é feito para os demais nós. Para facilitar essa representação é feita a árvore de dominância ou árvore de dominadores, nessa árvore é representada a relação de dominador/dominado no grafo de fluxo, e onde cada nó $d$ domina todos os seus descendentes, como pode ser visto na figura 2.6(b).

Ainda conforme Aho *et al.*, através da árvore de dominadores algumas propriedades são verificadas:

> (...) cada nó $n$ possui um *único dominador imediato* $m$, que é o último dominador de $n$ em qualquer percurso a partir do nó de entrada até $n$. Em termos da relação *dom*, o dominador imediato $m$ tem essa propriedade de que, se $d \neq n$ e $d$ *dom* $n$, então $d$ *dom* $m$. [Aho et al., 1986]

Através das propriedades de dominância, obtidas através do grafo de fluxo de controle e da árvore de dominadores, algumas análises são possibilitadas. Duas das análises utilizadas no anotador proposto (Análise de Divergências e Análise de Paralelismo) nessa dissertação utilizam este conceito. Como foi citado anteriormente, otimizações baseadas em laços são de extrema importância visto que eles são responsáveis pela maior parte do tempo de execução de um programa. Todas as análises explicadas e utilizadas nessa dissertação foram feitas para laços.

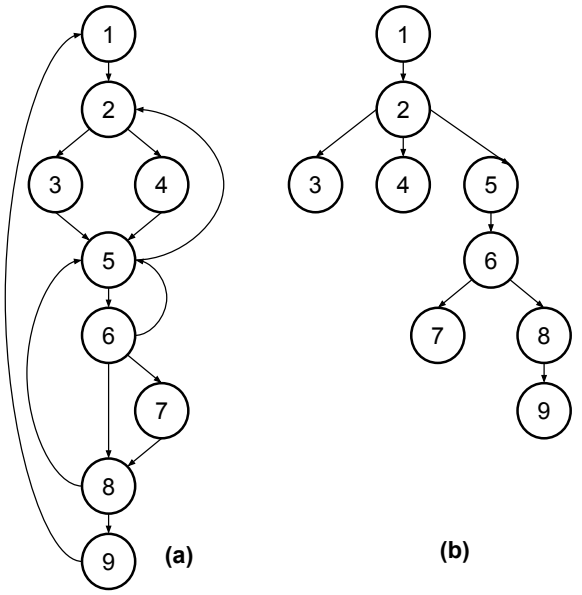


Figura 2.6: (a)Grafo de Fluxo; (b)Árvore de dominadores.

## 2.2 Modelos de Custo

Nessa dissertação são realizadas comparações de tempos de execução e de quantidade de dados coletados para mostrar a eficiência da ferramenta aqui proposta. Devido a isto, serão mostrados nessa seção alguns modelos de custo para se determinar desempenho de sistemas. Algumas características são avaliadas para fazer essa verificação, como: a duração de determinado evento, ou função, em um programa; a contagem de quantas vezes determinado evento ocorre; ou o valor atribuído a um determinado evento, ou a repetibilidade do mesmo. Através dessas medições é possível derivar uma métrica para avaliação do desempenho de sistemas. Para classificar se a métrica criada é consistente são avaliadas algumas características de acordo com Jalby *et al.* [Jalby et al., 2012]:

- Linearidade, é verificado se o valor da métrica varia a uma taxa constante, e se o desempenho da máquina varia na mesma proporção;
- Confiança, é verificado se dados dois sistemas distintos, em que um tem o desempenho superior ao outro, se a métrica indicar isso. Nesse caso a métrica adotada é confiável;
- Repetibilidade, toda vez que o experimento é executado, ele retorna o mesmo resultado, ou seja, a métrica adotada deve ser determinística;
- Fácil medição, caso uma a métrica seja de difícil medição ela não será utilizada em nenhum estudo.

Devido à alta complexidade de determinados sistemas e em consequência sua verificação, várias métricas de desempenho propostas na computação não são adequadas ou são interpretadas de forma incorreta. Medir o desempenho de um determinado sistema também é algo complexo, visto que depende de diversos fatores. Efetuar medições em aplicações que possuem diversos tipos de instruções, funções e representá-las em apenas um valor pode levar o analista a falsas decisões. Para amenizar esse erro e relacionar todas as medições em um valor, são realizados alguns cálculos matemáticos de médias para reduzir ao máximo o erro.

Nessa dissertação a partir da execução dos *benchmarks* selecionados, será feito um comparativo do desempenho do programa sequencial e do programa paralelo gerado. Porém, a definição de desempenho é relativa. Uma das primeiras e mais aceitas medidas de desempenho (e a qual é utilizado neste trabalho) é o tempo gasto. Utilizando as regras descritas anteriormente e aplicando uma média ponderada é feita a avaliação do desempenho do arcabouço proposto nessa dissertação.

## 2.3 Sistemas de Anotação para Hardware Heterogêneo.

Sistemas de anotação de código, voltados para o desenvolvimento de aplicações paralelas em arquiteturas heterogêneas, foram criados com o intuito de aumentar a produtividade do programador. Estes sistemas têm sido amplamente utilizados na comunidade científica e na indústria [Planas et al., 2015]. Neste trabalho, o sistema adotado para determinar em qual processador cada laço de um programa deve ser executado foi o OpenACC [Standard, 2013]. Existem vários outros sistemas de anotação similares a OpenACC. A mais conhecida, e mais antiga, dentre tais metalinguagens é o OpenMP [Jaeger et al., 2015], criada em 1997. OpenMP 4.0 possui, atualmente, um conjunto de diretivas, rotinas e variáveis de ambiente que dão ao compilador informação suficiente para transformarem código em C para código CUDA. Um terceiro sistema de anotações é OpenSs [Meenderinck & Juurlink, 2011], produto de pesquisa realizada no Centro de Computação de Barcelona. O OpenACC possui um grande suporte por parte da indústria, e devido a isto foi escolhido para a utilização nessa dissertação. 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* [Garland, 2008]. Neste trabalho foi utilizado PGCC, um compilador fechado desenvolvido pelo *Portland Group* (PGI). Existem, contudo alternativas simi-

lares hoje publicamente disponíveis. O OpenARC [Lee & Vetter, 2014] é 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*, *CAPS entreprise*. A versão mais recente do GCC [3] possui suporte experimental ao OpenACC através da biblioteca GOMP . A técnica de compilação descrita neste trabalho não tem 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*, ou diretiva, força a paralelização do código, mesmo que o tradutor de OpenACC para CUDA não consiga provar que as iterações são independentes.

## 2.4 Paralelismo

Para verificação de qual laço poderia ser enviado para a GPU foi feita uma análise para verificar qual laço pode ser paralelizado. Esta análise identifica, a partir de um programa de entrada, quais os laços cujas iterações podem ser executadas totalmente em paralelo. Para isso, é feita uma análise de dependência [Cytron et al., 1991]. Apenas laços onde todas as iterações podem ser executadas em paralelo, inclusive nas de seus laços mais internos, são marcados como paralelos.

Relações de dependência de dados são verificadas para indicar a ordenação existente entre as instruções de um programa. Essa ordem deve ser preservada em qualquer transformação do compilador para garantir a validade do código que será gerado. As

[3] `https://gcc.gnu.org/wiki/OpenACC`

principais classificações de dependências de dados e em que situações elas ocorrem são apresentadas a seguir.

1. **Dependência Verdadeira ou Dependência de Fluxo (*RAW - Read-After-Write*):** No código do exemplo mostrado na figura 2.7 as instruções presentes na linha 1 e 2 não podem ser executadas ao mesmo tempo uma vez que a instrução 2 precisa do valor $A$ computado na instrução 1.

```
1 A = B + C
2 D = A + 2
3 E = A * 3
```

Figura 2.7: Dependência verdadeira ou dependência de fluxo.

2. **Anti-dependência (*WAR - Write After Read*):** No código apresentado na figura 2.8 a instrução presente na linha 1 usa o valor de $B$ antes da instrução da linha 2 atribuir um novo valor a $B$. Devido a isso essa ordem deve ser mantida para que o valor de $B$ utilizado seja o valor antigo, não o computado na linha 2.

```
1 A = B + C
2 B = D *2
```

Figura 2.8: Anti-dependência.

3. **Dependência de Saída (*WAW - Write After Write*):** No exemplo mostrado na figura 2.9, a instrução 1 e a instrução 3 estão atribuindo valor a variável $A$. Dependendo da ordem de execução das instruções, o valor resultante de $A$ pode ser errado.

```
1 A = B + C
2 D = A + 2
3 A = E * F
```

Figura 2.9: Dependência de saída.

4. **Dependência de Controle ou Dependência Procedural:** Quando ocorrem desvios condicionais em um código, como pode ser visto no exemplo presente

na figura 2.10 (a), o valor de $A$ utilizado pela instrução presente na linha 3 tanto pode ser o que foi gerado pela instrução da linha 1 ou pela instrução 2, dependendo do valor de $X$. Considerando as dependências de dados dentro de laços, como pode ser visto no exemplo na figura 2.10(b) existe uma dependência de dados entre a instrução da linha 2 e a instrução da linha 1. Porém essa dependência ocorre na mesma iteração do laço, visto que o valor do elemento $A$ produzido na instrução 1 será utilizado na instrução 2. O contrário pode ser visto no exemplo da figura 2.10(c) a dependência entre a instrução 2 e 1 permanece, porém, para qualquer iteração $i$ a instrução 2 utilizará o valor do elemento de $A$ produzido na iteração anterior. O que impedirá a paralelização desse laço.

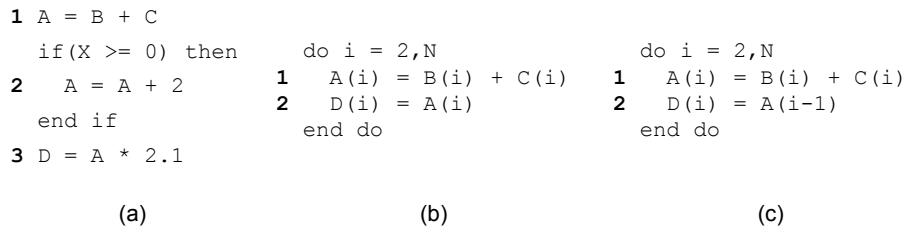


Figura 2.10: Dependência de controle.

Na análise criada nessa dissertaçnao, são analisados dois tipos de dependência, a dependência entre registradores e dependência de memória. Caso não haja nenhuma das dependências citadas o laço é considerado como paralelo.

**Dependência de registradores** Primeiramente é feito a verificação se há alguma dependência no laço, para isto o pressuposto é que variáveis de indução *phi* sempre podem ser reescritas em função da *threadID* [4]. Após é verificado se não há valores produzidos dentro do laço que são utilizados fora do laço. Se assim for, a ordem da iteração deve ser preservada.

**Dependência de memória** Primeiro é verificado se existe uma dependência confusa, ou seja, não é possível identificar como as instruções interagem. Não é possível paralelizar laços que contenham este tipo de instrução. Nossa análise não considera dependências entre *loads*, visto que são falsas dependências. É considerado que não há nenhuma dependência em um nível quando a distância dependência é conhecido como sendo zero.

[4] *Threads* podem ser definidas como uma linha ou encadeamento de instruções, essas instruções são executadas concorrencialmente. A *treadID* é o identificador de cada *thread*.

## 2.5 Divergências

Divergência de dados e de controle podem comprometer um possível ganho de performance que as técnicas aqui propostas podem obter, devido a isto foram implementados duas análises que estimem o impacto que a divergência de memória, e de controle causam na qualidade do código para a GPU que é gerado através da ferramenta proposta. Nessa seção é detalhado como as divergências citadas ocorrem e são referênciados trabalhos presentes na literatura que as classificam.

### 2.5.1 Divergência de Controle

Para explicar como divergências de dados e controle 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*), e são acessadas por um identificador (*threadID*). Esta organização interna faz com que instruções condicionais se tornem fonte de potencial perda de desempenho, como exemplificado na figura 2.11, num fenômeno conhecido como *divergência*. Neste trabalho, é utilizado uma técnica capaz de avaliar o impacto de dois tipos de divergência: de controle e de dados.

De acordo com Sampaio *et al.* [Sampaio et al., 2013] a divergência de dados ocorre sempre que o mesmo nome da variável é mapeado para valores distintos por diferentes *threads*. Neste caso, é dito que a variável é divergente, caso contrário, a variável é considerada uniforme. Causada pela divergência de dados a divergência de controle ocorre quando *threads* em um *warp* seguem caminhos distintos após o processamento da mesma instrução de desvio. Se a condição de desvio é controlada por dados divergentes, então ela pode ser verdade para algumas *threads*, e falso para outras. O código de ambos os caminhos, após a instrução de desvio, é sequencial. Dado que cada *warp* tem acesso a apenas uma instrução de cada vez algumas threads ficam ociosas enquanto outras estão executando.

### 2.5.2 Divergência de memória.

Também causada pela divergência de dados, este termo foi definido por Meng *et al.* [Meng et al., 2010] e eles explicam que ela ocorre sempre que uma instrução de carga

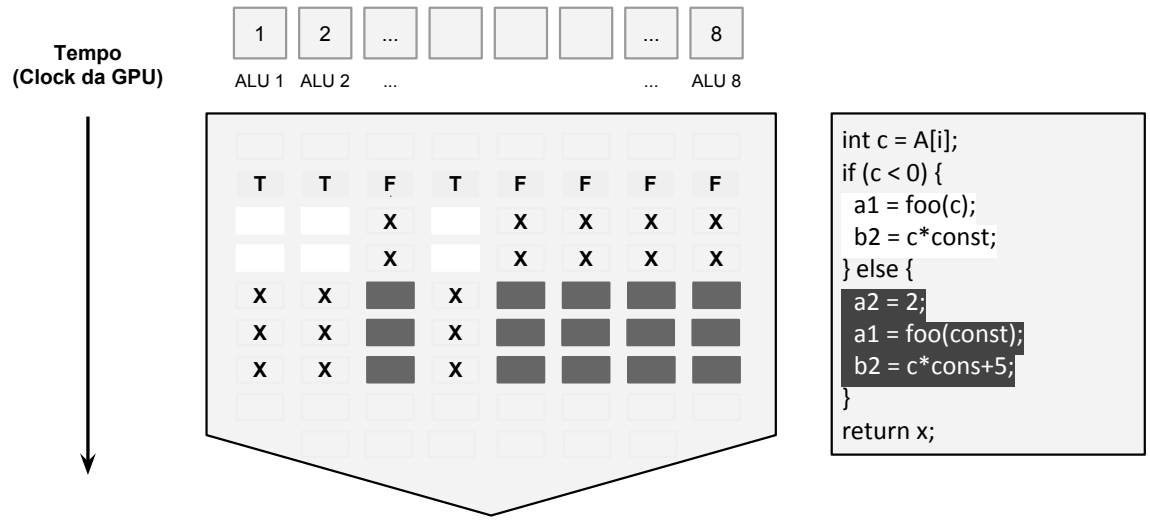


Figura 2.11: *Warp* com 8 *threads*: um conjunto de ALUs (assinaladas por X) deve permanecer ocioso sempre que suas *threads* não cumprirem a condição do bloco *if-else* no programa à direita. Este fato pode causar perda de desempenho.

ou de armazenamento de dados marcam endereços de dados divergentes provocando que threads em um *warp* acessem posições de memória com uma localidade ruim. Geralmente cada *thread* solicita uma posição de memória diferente para cada acesso de dados da entrada. Devido a isto a GPU possui uma grande largura de banda de memória, possibilitando realizar a transferência de uma grande quantidade de dados de uma única vez. Porém se o acesso a memória solicitado por uma única instrução pode ser separado em mais de uma largura de banda de comunicação de memória, então mais de um acesso a memória é necessário para processar os dados de uma única instrução, o que pode provocar muitas transferências de memória.

Na Figura 2.12 é exemplificado como ocorre um acesso á memória coalesced: todos os dados estão na memória global e esse acesso seria quando todas as *threads* em um *warp* irão acessar dados sequenciais. Na representação do acesso *non-coalesced* ou divergente os dados são acessados de forma randômica ou não sequencial.

Conforme Amilkanthwar *et al.* [Amilkanthwar & Balachandran, 2013] unir os acessos à memória é bom não só para o desempenho da aplicação, mas também para diminuir o consumo de energia. Maior quantidade de calor é dissipada quando o *hardware* precisa fazer uma busca de dados na memória local repedidas vezes, devido a isto maior quantidade de calor será dissipada ocasionando um gasto maior de energia. Várias pesquisas foram feitas para amenizar o efeito de divergências em uma aplicação. Essa dissertação utiliza duas análises de divergências (chamadas de divergência de memória e divergência de controle) para identificar laços que possuem muitas instruções

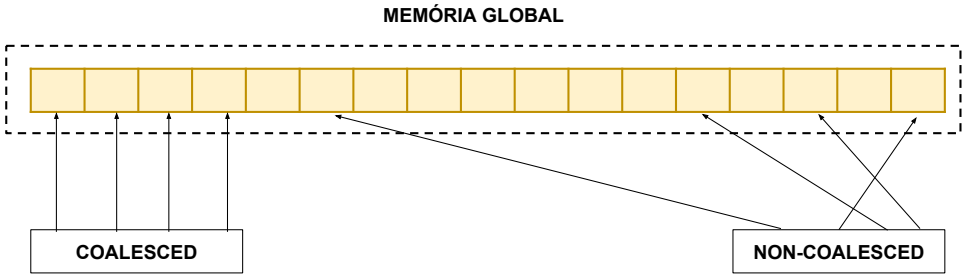


Figura 2.12: Representação de como é um acesso à memória *coalesced* e um acesso *non-coalesced* ou divergente.

divergentes e decidir se aquele laço deve ou não ser enviado para a GPU.

## 2.6 Inferência de tamanho de arranjos

Sempre que um laço é anotado para ser 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 término de cada laço à *Análise de Evolução Escalar* [Bachmann et al., 1994] 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 2.13 mostra o resultado obtido ao se aplicar esta análise a um programa de exemplo.

A Figura 2.13 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 está sendo usado a implementação de Alves *et al.* [Alves et al., 2012]. Entretanto, o uso desse tipo de análise para permitir a cópia de dados correta e automática entre dispositivos

```
1  for (int i = 0; i < n; ++i) {
2    arr[i-1] = i+1;
3    arr[i*2] = i;
4  }
```
**(a)**

```
1   int iMin = 0, iMax = n-1;
2   int idxMin = min(iMin-1, iMin*2);
3   int idxMax = max(iMax-1, iMax*2);
4
5   #pragma acc data pcopy(arr[idxMin:idxMax])
6   #pragma acc kernels
7
8   for (int i = 0; i < n; ++i) {
9     arr[i-1] = i+1;
10    arr[i*2] = i;
11  }
```
**(b)**

Figura 2.13: Inferência de tamanho de arranjos. (a) Programa original; (b) Programa automaticamente anotado.

é uma contribuição dessa dissertação.

## 2.7 Conclusão

Nesse capítulo foi especificado alguns termos necessários para o entendimento do anotador proposto. Para a compreensão completa da solução proposta, alguns conceitos como: blocos básicos, grafo de fluxo de controle, árvore de dominância dentre outros explicados nessa seção, são necessários. Nesta seção também foram apresentados outras ferramentas disponíveis que executam a mesma função ou uma função similar a algumas análises utilizadas no anotador. Até o presente momento não é de conhecimento da autora dessa dissertação ferramentas que executam a mesma função de forma automática e apenas com análises estáticas.

# Capítulo 3

# Solução Proposta

Neste capítulo é apresentada a solução proposta nessa dissertação. Inicialmente com o intuito de comparar a solução implementada são apresentadas algumas soluções similares encontradas na literatura. Também é explicado a solução preliminar desenvolvida pela autora dessa dissertação em conjunto com outros autores. Essa solução foi a motivação para o desenvolvimento do anotador por isso a importância de se ter uma seção para o detalhamento da mesma. Na terceira seção é feito um *overview* da solução proposta. Na quarta seção é feito uma explicação detalhada de como foi desenvolvido cada análise presente na ferramenta. Ainda nessa seção é explicado como o anotador faz a junção de todas as análises apresentadas. Para concluir é feito uma avaliação da solução proposta, onde é apontado pontos de melhoria da ferramenta.

## 3.1 Soluções Encontradas

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. O objetivo deste trabalho é decidir, em tempo de compilação, em que parte de um *hardware* heterogêneo os diferentes laços de um programa devem ser executados. Com o intuito de otimizar programas paralelos, existe uma vasta literatura descrevendo diferentes tentativas de reimplementar algoritmos tradicionais em GPUs. Exemplos de tais algoritmos envolvem sequenciamento genético Sandes & de Melo [2010], ordenação Cederman & Tsigas [2009], roteamento Mu et al. [2010], entre outros. Ao contrário desses trabalhos, a técnica de compilação aqui implementada gera código para placas gráficas sem nenhuma intervenção do usuário. Em outras palavras, a abordagem proposta neste artigo é uma forma automática de reimplementar em OpenACC para GPU código originalmente feito para uma CPU. A implementação

automática de código dá, ao desenvolvedor, uma forma rápida de prototipar ideias na placa gráfica.

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 [Lee & Vetter, 2012] 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. No trabalho aqui apresentado, é retirado do programador o ônus de lidar com estas otimizações manuais, inserindo diretivas de paralelização de forma completamente automática, e passando para o compilador a tarefa de otimização.

Lee *et al.* [Lee et al., 2009] 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. Porém o objetivo dessa dissertação 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. Amni *et al.* [Amini et al., 2011] desenvolveram uma técnica para melhorar a cópia de dados entre CPU e GPU reescrevendo código de forma automática. Contudo, ao contrário do presente trabalho, eles não decidem se computação deve ser executada na GPU. Tal decisão é tomada pelo desenvolvedor de programas. Kao e Hsu [Kao & Hsu, 2015] propõem uma técnica de programação para arquiteturas heterogêneas utilizando um perfilador capaz de determinar quando é benéfico enviar parte de um programa para um acelerador. Ao contrário dessa abordagem, a técnica aqui proposta é completamente estática e independente de conjuntos específicos de valores de entrada.

Várias linguagens de programação oferecem sintaxe e semântica para processamento paralelo. *Haleto* [Ragan-Kelley et al., 2013], por exemplo, é 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 distribuídas nos diferentes dispositivos de processamento disponíveis. Ao contrário dessas abordagens, as técnicas aqui propostas são completamente automáticas, ou seja não é necessário nenhuma intervenção do usuário(a).

## 3.2 Estudos Preliminares

Uma das motivações para o problema a ser resolvido nessa pesquisa de mestrado vem de um trabalho que também foi realizado, durante esse mestrado, para colocação automática de computação em *hardware* heterogêneo [do Couto Teixeira et al., 2015a]. A ferramenta desenvolvida tem como objetivo determinar, com um mínimo de intervenção do usuário, em qual processador cada parte de uma aplicação paralela deve ser executada, como pode ser visto na Figura 3.1. Etino, a ferramenta desenvolvida, agrupa, em um mesmo pacote, três ferramentas bem conhecidas, a saber: o compilador LLVM, o perfilador *aprof* e o tradutor de código ipmacc. Além de ligar tais aplicações, a construção de Etino demandou a implementação de uma análise de complexidade assintótica, e um anotador de código. A iteração entre esses vários produtos de *software* será explicada no restante da presente seção.

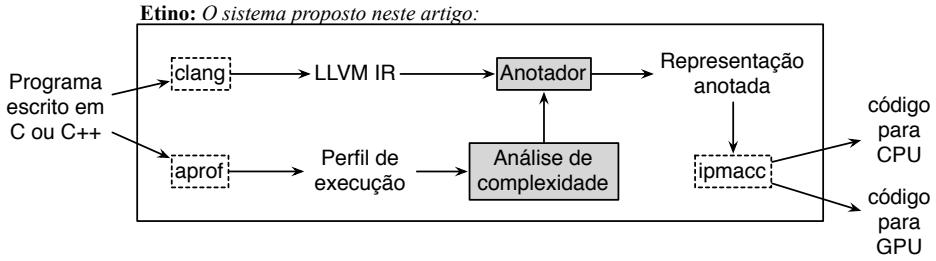


Figura 3.1: Visão geral do sistema de compilação proposto neste trabalho.

**Perfilamento sensível à entrada.** A fim de determinar o comportamento assintótico de uma aplicação, Etino produz um perfil de sua execução. Tal perfil é criado por meio da ferramenta *aprof*. Aprof retorna, para cada função de um programa, um gráfico que relata o tamanho da entrada lida com o número de instruções necessário para processar aquela entrada. O tamanho da entrada de um programa é definido como a quantidade de posições de memória que ele lê sem antes escrevê-las. Tal métrica é chamada RMS, sigla de *Read Memory Size* [Coppa et al., 2012].

Uma vez que aprof usa a memória lida como medida de tamanho de entrada, essa ferramenta é robusta o suficiente para determinar a complexidade de funções que usam estruturas de dados esparsas, tais como listas encadeadas e grafos. Essa abordagem, na opinião dos autores desse artigo, é superior às alternativas puramente estáticas de inferência de complexidade [Gawlitza & Monniaux, 2012; Gulavani & Gulwani, 2008; Gulwani et al., 2009] que foram recentemente introduzidas. A principal desvantagem das abordagens puramente estáticas é o seu curto alcance: elas dependem de entradas de dados regulares e produzem resultados muito conservadores para programas cujo

fluxo de controle pode atravessar diversos caminhos. Em face desse tipo de desafio, elas assumem que o comportamento do programa é sempre o pior caso, ainda que tal situação ocorra raramente. Além disso, a experiência dos autores com Loopus, uma ferramenta baseada nas técnicas de Gulwani *et al.* [Gulavani & Gulwani, 2008; Gulwani et al., 2009], revela que análises de complexidade puramente estáticas tendem a retornar resultados para uma quantidade relativamente pequena de laços em uma aplicação.

**Análise de Complexidade e Anotação de Código.** Uma vez produzido um perfil de execução para um programa, Etino passa à fase de análise de complexidade. Nesta fase analisam-se pares $(I \times T)$, onde $I$ é o tamanho da entrada, em RMS, e $T$ é o tempo de execução, medido como o número operações executadas. Enfatiza-se que ambas grandezas são determinísticas, isto é, um mesmo programa produz sempre o mesmo par $(I \times T)$, para a mesma entrada. A partir de um conjunto de pares, Etino procura encontrar uma curva que melhor se adeque àqueles pontos. Etino não utiliza qualquer heurística elaborada: funções cuja complexidade seja super-linear são marcadas para execução na CPU. Considera-se como super-linear qualquer função cujo coeficiente de regressão linear $(R)$ seja inferior a 0.9, e cuja reta de integração possua inclinação superior a 1.0.

A decisão de onde enviar cada computação é feita via anotações OpenACC. Etino utiliza uma transformação de código, implementada como um passe LLVM, para inserir tais anotações. Somente laços paralelos são considerados nesta fase. Um laço paralelo é marcado com a diretiva `#pragma acc loop independent`. O usuário de Etino deve indicar quais laços são paralelos: a ferramenta ainda não consegue detectar dependências entre iterações de laços. Cada laço paralelo tem sua complexidade analisada, conforme descrito anteriormente, e aqueles considerados promissores são anotados com a diretiva `#pragma acc kernels`. Etino usa informações de depuração para encontrar os cabeçalhos de laços onde inserir anotações. LLVM adiciona essas informações a sua representação intermediária quando o programa é compilado com a extensão `-g`.

**Geração de Código.** A geração de código é feita via ipmacc, um compilador de diretivas OpenACC desenvolvido na Universidade de Vitória, Canadá[1]. Ipmacc é um compilador fonte-a-fonte, que, conforme pode ser visto na Figura 3.1, produz dois tipos de código: funções escritas em C e funções escritas em C para CUDA. O mecanismo de geração de código adotado por ipmacc é simples. Laços que são marcados com

[1]Nesse trabalho foi usado a versão de ipmacc de 15 de Março de 2015, disponível neste endereço: `https://github.com/lashgar/ipmacc`

a diretiva `kernel` são transformados em funções CUDA. Essa transformação atribui uma *thread* para cada iteração daquele laço. Assim, é vital que não existam dependências entre iterações diferentes, ou o programa resultante dessa transformação pode ficar semanticamente errado. A Figura 3.2 ilustra essa transformação. O compilador ipmacc é capaz de traduzir o código visto na parte (a) da figura para o código visto na parte (b).

```
void saxpy(int n, float alpha, float *x, float *y) {
 #pragma acc data copyin(x[0:l]) copy(y[0:l]) {
  #pragma acc kernels {
   #pragma acc loop independent {
    for (int i = 0; i < n; i++) {
     y[i] = alpha*x[i] + y[i];
    }
   }
  }
 }
}
```

(a)

```
__global__ void
saxpy(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];
 }
}
```

(b)

Figura 3.2: (a) Programa escrito com diretivas OpenACC. (b) Programa que ipmacc produz para a entrada vista na parte (a) desta figura.

## 3.3 Este trabalho em um exemplo

O anotador automático proposto neste trabalho utiliza três análises, conforme já mencionado na seção 1: uma análise de paralelismo [Alves et al., 2012], uma análise de divergências [Sampaio et al., 2014] e uma análise de tamanho de arranjos [de Assis Costa et al., 2013]. 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 da ferramenta proposta, será utilizado o exemplo da Figura 3.3-a. Inicialmente, é verificado quais laços podem ser paralelizados, após essa verificação é analisado se os mesmos 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 inclui no corpo nenhuma instrução condicional e, consequentemente, não apresenta divergências de controle. 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 3.3: (a) Código sequencial de entrada; (b) Código de saída, automaticamente paralelizado pela ferramenta aqui apresentada.

Para que uma operação seja executada em um acelerador, seus dados de entrada devem estar presentes na memória interna do mesmo. Dessa forma é necessário definir o conjunto de dados que devem ser movidos para a GPU. Isso 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. Foi automatizado 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 nessa seção. A figura 3.3-b apresenta o programa de exemplo após ser automaticamente anotado pelo anotador proposto. 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.

## 3.4 O algoritmo de paralelização

A solução proposta é composta por três análises estáticas de código, executadas a nível de laços do programa, como pode ser visto na figura 3.4. 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 a ferramenta selecionar 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 três análises descritas a seguir foram implementadas sobre o compilador LLVM.

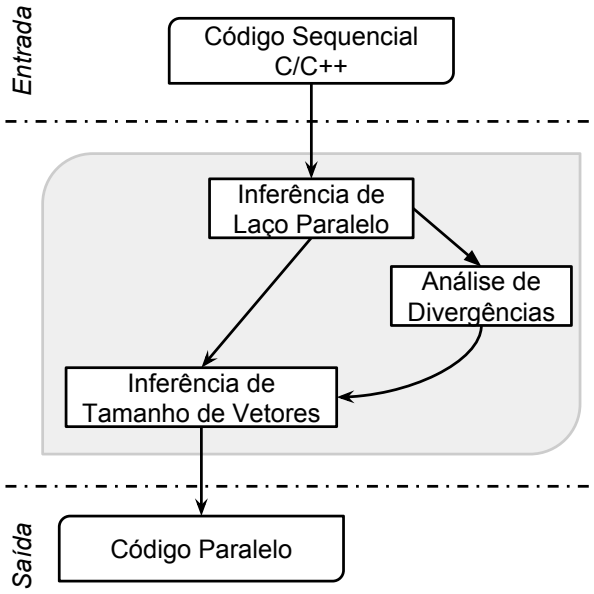


Figura 3.4: Fluxo de funcionamento das análises aqui propostas.

### 3.4.1 Análise de paralelismo

É a primeira análise a ser executada no anotador. As demais análises são independentes porém, na ferramenta aqui proposta, elas são executadas apenas em laços paralelos. Esta análise é responsável por identificar quais laços podem ser paralelizados, ou seja, quais laços não possuem nenhum tipo de dependência e ao ser paralelizado não irá gerar um resultado diferente do esperado. A seguir é detalhado cada passo realizado por esse algoritmo em ordem de execução, lembrando que a análise criada é feita a nível de laço:

1. Primeiro é verificado se possui alguma dependência de memória para cada par de instruções presentes na função. Nessa etapa é identificado se as duas instruções são de leitura ou escrita em memória, após essa verificação uma série de condições são analisadas:

    - Apenas dependências verdadeiras, ou seja, são ignoradas quando o par de instrução é um *load*, visto que este tipo de dependência não gera erro ao ser paralelizada.

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

Figura 3.5: Exemplo de utilização da análise de paralelismo.

- Na classe *Dependence* do LLVM existe uma função chamada *isConfused*, essa função retorna verdadeiro para toda dependência confusa [2].
- Após essa verificação, a função insere, em uma estrutura de dados, todos os laços que contenham pelo menos uma das duas instruções analisadas.

2. Após a verificação em memória, é verificado se possui alguma dependência entre registradores:

   - Nessa etapa é verificado se existe alguma dependência que é transportada entre iterações do laço. A suposição é que o passo constante do laço, ou a variável de indução $\phi$ sempre podem ser reescritas como uma função de *threadId*.
   - Após é verificado se algum valor produzido dentro do laço é utilizado em alguma instrução fora do laço. Caso essa condição seja satisfeita a ordem de cada iteração do laço deve ser preservada. Logo, este laço não poderá ser paralelizado.

No exemplo mostrado na figura 3.5 pode-se verificar dois casos onde a análise indica corretamente se o laço pode ser paralelizado. A primeira instrução do laço (linha 3) não possui nenhuma dependência e pode ser paralelizada. Porém a segunda instrução, presente na linha 4, possui uma dependência de controle: a matriz $a$ depende do valor de $b$ obtido na iteração anterior. Devido a essa dependência, apenas o laço mais externo será paralelizado, ou seja , o laço cuja variável de indução é o $i$.

### 3.4.2 Análise de Divergência de Controle

Essa análise é uma adaptação do algoritmo proposto por Sampaio *et al.* [Sampaio et al., 2014], e hoje disponível no compilador LLVM[3]. Nota-se, contudo, que este trabalho

[2]Dependência confusa é quando o compilador não consegue fazer uma análise de dependência entre as instruções, dessa forma ele assume a pior das hipóteses, ou seja, existe uma dependência.

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

descreve a primeira tentativa de se utilizar a análise de divergências para melhorar modelos de custo que tentam prever o desempenho relativo de computação executando na CPU ou na GPU.

A primeira etapa executada nessa análise é identificar instruções e/ou variáveis fontes de divergências. Como essa análise foi criada primeiramente para o uso em uma representação NVPTX[4] foram criadas funções para representações X86, arquitetura utilizada para o funcionamento do anotador. Na alteração feita foi considerado que: **toda variável de indução de um laço identificado como paralelo é uma fonte de divergência**. Esta alteração foi necessária visto que ao ser inserido as diretivas para paralelização do laço a variável de indução do mesmo será a *threadId* no *kernel* da função CUDA criada.

Após identificar todas as fontes de divergência da função analisada é feita a propagação da divergência, ou seja, são identificadas todas as instruções divergentes cuja fonte seja a variável de indução do laço. A primeira verificação realizada é se uma determinada instrução condicional ( *branch*) $T$ é divergente: se todos os nós $\phi$ presentes nos blocos básicos **pós dominados** pelo bloco $T$. Essa regra está exemplificada na figura 3.6. Como a instrução presente na linha 1 da Figura 3.6(a) é dependente de uma fonte de divergência($tid$), como pode ser visto na árvore de dominância 3.6(c) do grafo de fluxo mostrado em 3.6(b), logo a instrução na linha 4 será divergente.

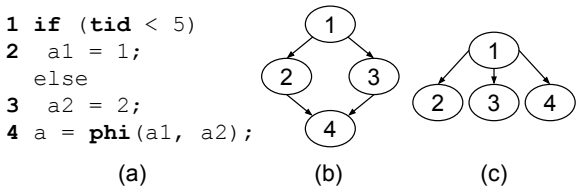


Figura 3.6: Primeira regra: (a) Código analisado; (b) Grafo de Fluxo; (c)Árvore de Dominância.

A segunda verificação realizada é: se um valor definido dentro do laço é usado fora do mesmo, este valor é dependente de sincronização, ou dependente de controle, na condição de saída do laço que o domina. Como pode ser visto na figura 3.7, dado um condicional $T$, é verificado todas as variáveis definidas na região de influência porém usada fora da mesma. Todas essas instruções são dependentes de sincronização de $T$. Para acelerar o processo de pesquisa, ao invés de buscar todos os blocos básicos da região de influência, é pesquisado todos os dominadores de $T$ até que esteja fora da região de influência.

[4]O LLVM possui um *back-end* para programação em GPUs chamado NVPTX `http://llvm.org/docs/NVPTXUsage.html`. PTX é uma linguagem intermediária produzida pela compilação de

```
1 int i = 0;
2 do {
3   i++;
4   if (foo(i)) ... // uniform
5 } while (i < tid);
6 if (bar(i)) ...   // divergent
```

Figura 3.7: Exemplo de análise feita para a segunda regra da análise de divergências de controle.

Após a identificação de todas instruções divergentes, é feita a computação do quão divergente é o laço. É verificado a porcentagem de divergência de um laço, e, baseado nos testes realizados, laços com mais de 80% de instruções divergentes não devem ser enviados para a GPU. No exemplo mostrado na figura 3.8, devido à instrução condicional presente na linha 5 e pelas regras explicadas anteriormente, a análise retornou que 23% das instruções presentes nesse laço são divergentes. Dessa forma este laço pode ser paralelizado e, possivelmente, não terá perda de desempenho na GPU.

```
1 for (i = 0; i < M; i++)
2   for (j = 0; j < N; j++)
3     for (k = 0; k < O; k++) {
4       a[i][j] = k + M * N;
5       if (i > 10)
6         a[k][j] = i + j;
7       a[j][k] = i * j;
8     }
```

| # INSTRUÇÕES | 21 |
|---|---|
| # DIVERGENTES | 5 |
| % DIVERGENTES | 23 |

Figura 3.8: Exemplo de uso da análise de Divergência de Controle: Programa de entrada e retorno produzido pela execução da análise modificada.

### 3.4.3 Análise de Divergência de Memória

Esta análise foi criada com o intuito de identificar acessos de memória *uncoalesced*, ou seja, divergência de memória. Essa análise já foi implementada por alguns pesquisadores mas não com a mesma finalidade em que está inserida neste projeto. Para identificação se em algum laço possui divergência de memória também foi considerado a variável de indução do laço ser a *threadId* da função em CUDA criada. Da mesma forma que a análise de divergência de controle, só serão analisados laços paralelos.

um programa em CUDA com o NVCC.

A primeira verificação, realizada no algoritmo proposto, é se os valores de determinado tipo são analisáveis com a classe ScalarEvolution[5]. Este teste inclui principalmente tipos inteiros, e na análise proposta inclui ponteiros em que se tem acesso a informações do destino específico. Após a primeira verificação é calculado o valor de saída para a variável, definida dentro do laço analisado, verificando o que manterá o valor no laço pai. Para isso é utilizado uma função da classe *ScalarEvolution* que retorna uma expressão para o valor especificado.

Após obter a expressão da variável, é verificado se essa expressão representa o seguinte formato: $A + B * x$ onde $A$ e $B$ são valores invariantes do laço. De forma conservadora estas variáveis já são marcadas como divergentes. Nesta etapa são verificados os operandos da expressão, se algum deles for a variável de indução do laço e os acessos não são agrupados, ou possuem acesso sequencial, também é marcado essa variável como divergente.

```
1  void bar() {
2    long a[M][M];
3    long i, j, k;
4    for (i = 0; i < M; i++)
5      for (j = 0; j < N; j++)
6        for (k = 0; k < O; k++) {
7          a[i][j] = k + M * N;
8          if (i > 10) {
9            a[k][j] = i + j;
10           printf ("a[k][j] = %ld\n", a[k][j]);
11         }
12         a[j][k] = i * j;
13       }
14 }
```

| | |
|---|---|
| #GEP's | 24 |
| #LOAD's | 3 |
| #STORE's | 9 |
| #Memória | 16 |
| #Uncoalesced | 4 |

Figura 3.9: Exemplo de uso da análise de Divergência de Memória: Programa de entrada e retorno produzido pela execução da análise.

A figura 3.9 mostra o resultado produzido pela análise de divergência de memória. Nesse resultado não foi considerado se o laço pode ser paralelizado ou não, então todos foram considerados como paralelos. Dessa forma todos os quatro acessos a memória foram considerados divergentes o que está correto. Na linha 7 do programa, se o laço cuja variável de indução é o $i$ for paralelizado este acesso será divergente. Na linha 12 do programa, se o laço cuja variável de indução é o $j$ for paralelizado este acesso será divergente. Da mesma forma acontece com o laço $k$. Ao ser marcado apenas o laço mais externo como paralelo o retorno é apenas um acesso *uncoalesced*. O que comprova a análise ser conservadora e correta.

[5] http://llvm.org/docs/doxygen/html/classllvm_1_1SCEV.html

### 3.4.4 Análise de Inferência do 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. Para automatizar todo o processo de paralelização foi criado essa análise para identificação dos limites dos arranjos. Em comparação ao Etino, solução preliminar proposta, esta análise foi crucial devido à remoção de envio de dados desnecessários à GPU. Para tanto, foram combinadas expressões condicionais que controlam o término de cada laço. Estes valores são então substituídos em cada expressão de indexação de memória encontrada ao longo do aninhamento, definindo assim seus limites. Estes últimos são 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.

```
1  for (int i = 0; i < n; ++i) {
2    arr[i-1] = i+1;
3    arr[i*2] = i;
4  }
```
**(a)**

```
1   int iMin = 0, iMax = n-1;
2   int idxMin = min(iMin-1, iMin*2);
3   int idxMax = max(iMax-1, iMax*2);
4
5   #pragma acc data pcopy(arr[idxMin:idxMax])
6   #pragma acc kernels
7
8   for (int i = 0; i < n; ++i) {
9     arr[i-1] = i+1;
10    arr[i*2] = i;
11  }
```
**(b)**

Figura 3.10: Resultado da Análise de inferência de tamanho de arranjos. (a) Programa original; (b) Programa automaticamente anotado.

A figura 3.10 mostra o resultado obtido ao se aplicar esta análise a um programa de exemplo. Como pode ser visto é identificado a variável de indexação do laço, no exemplo $i$, e os limites de acesso do vetor *pointer* [6]. Dessa forma apenas os valores acessados dentro do laço serão enviados para o *hardware* paralelo.

## 3.5 O anotador

O objetivo do presente trabalho é criar um anotador automático e apenas com análises estáticas. Como foi exemplificado na seção 3.3 e através da figura 3.4 o anotador possui três análises: análise de paralelismo, análise de inferência do tamanho de arranjos e

[6] No exemplo apresentado, a função *min* retornaria o valor $-1$ como limite inferior de acesso ao arranjo. Devido a isso foi implementado uma verificação, e apenas valores positivos são considerados, caso o cálculo retorne um valor negativo, será apontado como limite inferior o valor zero.

análise de divergências; sendo a última subdividida em duas (divergência de controle e divergência de memória ). Nessa seção será exemplificado como foi realizada a junção de todas as ferramentas listadas.

Inicialmente é executada a análise de paralelismo, para o seu correto funcionamento algumas otimizações são necessárias. Essas otimizações podem alterar muito o *byte code* gerado, dessa forma foi necessário a criação de uma função para armazenar quais laços encontrados são paralelos. A integridade do *byte code* gerado é essencial para o funcionamento da análise que infere o tamanho de arranjos. Após a coleta de todos os laços paralelos, foi criado uma análise responsável por inserir metadados[7] nos laços do *byte code* original, ou seja, sem as otimizações.

Ainda sobre o código otimizado são aplicadas as análises de divergência de controle e memória. As duas análises são executadas individualmente, nesta etapa são identificados quais laços possuem quantidade de divergência suficiente para comprometer o desempenho do código ao ser enviado para a GPU. Caso alguma das análises aponte para um laço divergente (80% ou mais das instruções e/ou acessos de memória divergente), o mesmo também é marcado, através de metadados como divergente. Lembrando que esta "marcação" é realizada no *byte code* original.

Após a execução das análises citadas anteriormente, é executado a análise responsável por inferir o tamanho dos arranjos. Se esta análise for executada no código otimizado a quantidade de acessos à memória analisável é reduzido. Devido a isto ela é aplicada no *byte code* original com poucas otimizações. Ela lê os metadados inseridos e é aplicada somente a laços paralelos e não divergentes. Na figura 3.11 é mostrado o fluxo detalhado de funcionamento do anotador.

## 3.6 Avaliação da solução proposta

Nesse capítulo foi apresentado a solução proposta nessa dissertação. Foi feito uma pesquisa de ferramentas similares disponíveis e foi apresentado a solução preliminar que foi a motivação desse trabalho. O anotador é capaz através das análises descritas identificar quais os laços podem ser paralelizados, quais possivelmente terão um ganho de desempenho ao ser enviado para a GPU e a partir de um código em C/C++ criar um código anotado com diretivas OpenACC compilável em qualquer compilador de OpenACC.

Foi criado um *script* para junção de todas as análises onde é possível com apenas um comando executar o anotador proposto. O anotador se mostrou eficaz para o que

[7]Metadados do inglês *Metadata* representa uma informação opcional sobre uma instrução, ou módulo, que pode ser descartada sem interferir na corretude do programa.`http://llvm.org/`

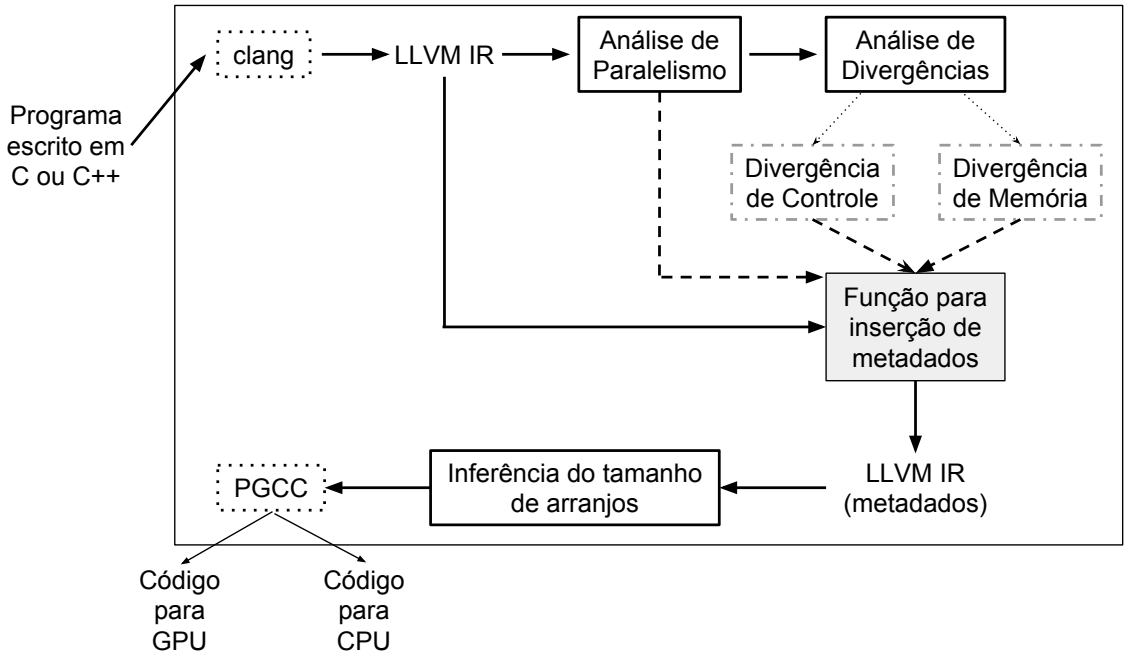


Figura 3.11: Fluxo detalhado de funcionamento do anotador

foi proposto. É totalmente estático, ou seja, todas as análises e alterações no código são realizadas durante a compilação do programa. É totalmente automático, como foi informado anteriormente com apenas um comando o programador(a) é capaz de executar a ferramenta.

# Capítulo 4

# Experimentos

O intuito dessa seção é demonstrar a efetividade do arcabouço de paralelização de código proposto nesta dissertação. Nessa seção é abordado sobre os testes realizados para verificação da ferramenta. Primeiro é detalhado o ambiente de testes utilizado: é descrito o *hardware* e cada *benchmark*. Na terceira seção são demonstrados através de gráficos e tabelas os resultados obtidos com o anotador. Esta seção de resultados é separada em resultados estáticos e tempos de execução, visto que além dos *speedups* alcançados outras benefícios podem ser retirados do trabalho apresentado. Para finalizar é realizado uma avaliação dos resultados obtidos.

## 4.1 Ambiente de Testes

Nesta seção é descrito o ambiente utilizado para a realização dos testes experimentais. Para o mesmo foi utilizado um servidor, disponível para os alunos do laboratório de compiladores, com as seguintes configurações:
**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).

## 4.2 Benchmarks

Nesta seção é descrito o *software* utilizado para a realização dos testes experimentais. São listados todos os *softwares* utilizados para o desenvolvimento da ferramenta e a

execução de cada experimento.

**Software** : Os testes mostrados nesta seção foram realizados sobre diversos *benchmarks*, incluindo Parboil, Rodinia e OMPSpec. Dada a grande quantidade de *benchmarks* disponíveis, é mostrado números condensados para a maior parte deles; entretanto, é detalhado o tempo de execução para todos os benchmarks disponíveis em Polybench[1]. Os tempos de execução para os outros benchmarks são semelhantes. PolyBench [Liu et al., 2013] 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* [Grauer-Gray et al., 2012]. Para a geração de código paralelo foi utilizado o compilador PGCC (*Portland Group C Compiler*) [2] 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 [3].

## 4.3 Resultados

Devido a ferramenta possuir várias análises que podem ser executadas de maneira independente sem perda de precisão, alguns resultados serão apresentados para a ferramenta como um todo e para algumas análises distintas. Nessa seção separamos em duas subseções: resultados estáticos e tempos de execução. Essa separação foi realizada para o melhor entendimento e aproveitamento dos resultados obtidos.

### 4.3.1 Resultados Estáticos

O objetivo da ferramenta é paralelizar o maior número de laços e enviar o mínimo de dados possíveis para a GPU. O anotador é executado de forma automática e suas análises são totalmente estáticas, ou seja, são realizadas em tempo de compilação. Com as análises disponíveis no anotador, é possível coletar vários dados que demonstram a eficiência da mesma.

Para inserir diretivas OpenACC, o anotador 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 no capítulo 3. A tabela 4.1 mostra que

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

foi possível analisar 98% dos acessos a memória presentes em cada um dos benchmarks do Polybench, o que maximiza o número de laços que será possível anotar com o arcabouço proposto. Nos *benchmarks* analisados é possível anotar 234 laços, ou seja, 95% dos laços presentes. Note que esta 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.

| | **Laços** | | **Memória** | | **Instruções** | |
|---|---|---|---|---|---|---|
| **Nome** | **Total** | **Anotáveis** | **Acessos** | **Analisáveis** | **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** |

Tabela 4.1: Dados estáticos coletados com o anotador automático.

Além da memória analisada e laços anotados, foi coletado a quantidade de instruções divergentes. Este dado é importante para a identificação de uma possível perda de desempenho ao se enviar um laço para a GPU, como explicado na seção 3. Ainda na tabela 4.1 pode-se ver que aproximadamente 26% das instruções encontradas são divergentes. Este número pode ser pequeno olhando o somatório geral, porém o anotador 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 será enviado o mesmo para a GPU.

Afim de demonstrar a capacidade da análise proposta de lidar com programas reais, ela foi utilizada em vários outros benchmarks além de Polybench. Foram selecionadas algumas funções de *benchmarks* para avaliar aplicações paralelas. Essas estão listadas a seguir:

**OMPSpec** [Saito et al., 2003] foi criado para mensurar a performance de aplicações baseadas em OpenMP para processamento de aplicações paralelas de memória compartilhada. O pacote inclui 14 aplicações científicas e de engenharia, incluindo computações de dinâmica de fluidos computacional, modelagem molecular e manipulação de imagens.

**Parboil** [Stratton et al., 2012] é uma coleção de aplicações para estudo de performance e computação da taxa de transferência entre arquiteturas heterogêneas. O pacote inclui aplicações comerciais e científicas, incluindo processamento de imagem, simulação biomolecular, dinâmica dos fluidos e astronomia.

**Rodinia** [Che et al., 2009] é um pacote de aplicações para arquiteturas heterogenias. Conforme Che *et al.* esse pacote de *benchmarks* foi criado para ajudar arquitetos a estudar plataformas emergentes como GPUs, Rodínia inclui aplicativos e *kernels* que têm como alvo a plataformas de vários núcleos de CPU e GPU. Os mesmos possuem aplicações de comunicação paralela, técnicas de sincronização e consumo de energia. Além dessas aplicações o Rodinia têm permitido a análise de limitações de largura de banda de memória e a consequente importância do *layout* de dados.

**MGBench** [do Couto Teixeira et al., 2015a] é uma coleção de 8 aplicações, criadas pela autora dessa dissertação em conjunto com outros alunos do laboratório de compiladores, para testar um anotador baseado em *profiling* de programas. Esse pacote inclui oito aplicações escritas em C com diretivas OpenACC.

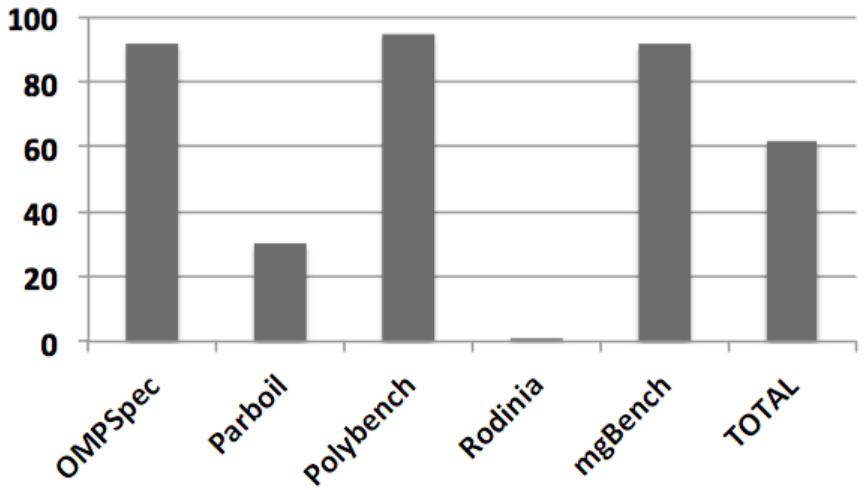


Figura 4.1: Gráfico: Porcentagem de laços Anotáveis.

Foi coletado os dados e os sumarizado nos gráficos das figuras 4.1 e 4.2. Como pode ser visto no gráfico da figura 4.1, a análise proposta é capaz de anotar aproximadamente 62% dos laços presentes em todos os *benchmarks* analisados[4]. Essa quantidade é realmente grande visto que a análise é totalmente estática e não depende dos dados de entrada.

Além da quantidade de laços analisáveis a inferência de tamanhos de arranjos é capaz de encontrar limites para mais de 80% dos acessos a memória presentes nas funções, como pode ser visto no gráfico da figura 4.2. 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 consegue-se 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 que a análise deixe de anotá-lo.

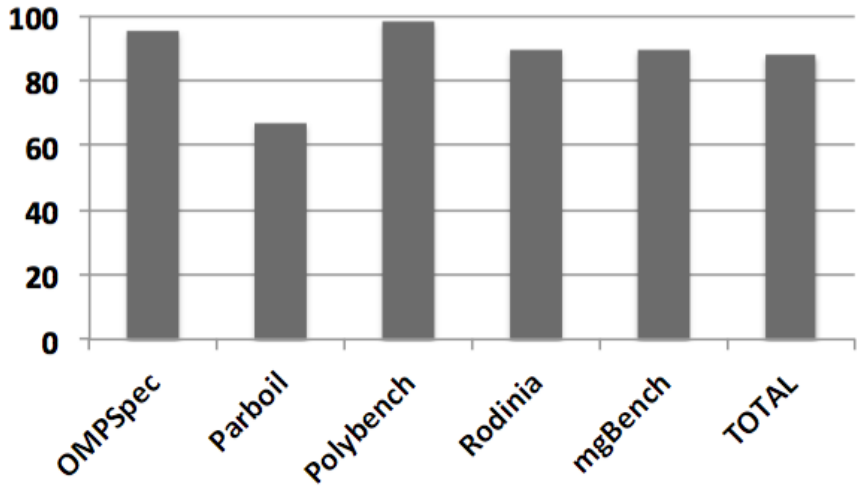


Figura 4.2: Gráfico: Porcentagem de memória analisável.

### 4.3.2 Tempo de Execução

A figura 4.2 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 4.2 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

[4] 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 analisar os acessos à memória presentes.

| Name | Device | MINI | SMALL | MEDIUM | LARGE | X 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 |

Tabela 4.2: Tempo de execução de programas paralelizados automaticamente, com entradas de cinco diferentes tamanhos.

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.

A Figura 4.3 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.* [Gawlitza & Monniaux, 2012] e Gulavani *et al.* [Gulavani & Gulwani, 2008] 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 [do Couto Teixeira et al., 2015a], porém, naquele caso, usou-se um perfilador para inferir a complexidade assimptótica dos algoritmos paralelizáveis.

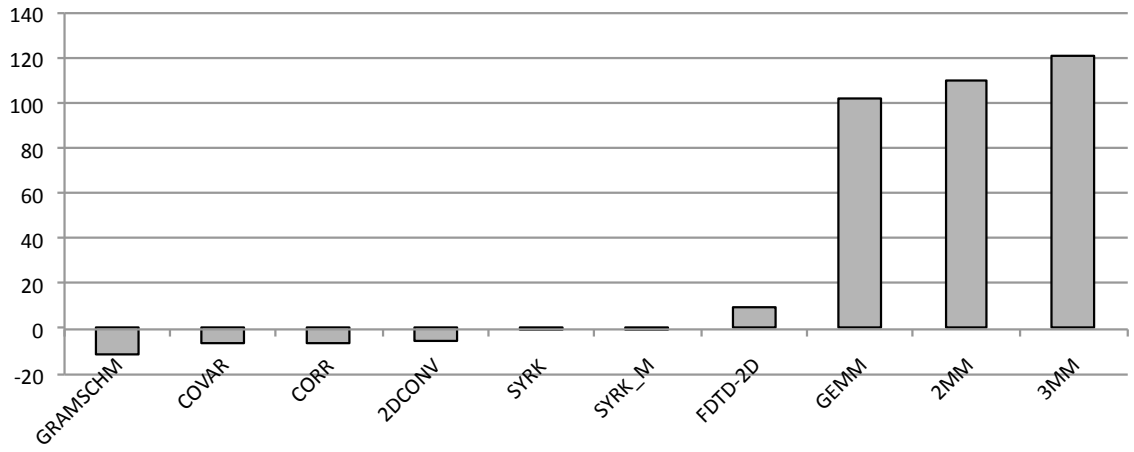


Figura 4.3: Gráfico: Speedup relativo entre os diferentes programas de Polybench que executaram por mais de um segundo com as maiores entradas disponíveis.

O maior *speedup* observado dentre tais programas, como pode ser visto no gráfico da figura 4.3 e na tabela 4.2, foi de 121x na execução da multiplicação de matrizes tridimensionais 3MM (293.513secs na CPU vs 2.424secs na GPU). Para detalhar melhor o resultado foi criado o gráfico 4.4. Nele é possível verificar a evolução do tempo de execução[5]do algoritmo. Para matrizes com dimensão acima 512 já se começa a ter um ganho de desempenho ao ser enviado para GPU. Este resultado já era esperado, visto que conforme Etino já havia demonstrado multiplicação de matrizes tem um ganho ao ser paralelizada, pois o custo computacional do algoritmo é superior ao custo de envio dos dados.

O maior *slowdown* observado dentre os programas testados, foi de 11x em GRAMSCHM (22.426secs na CPU vs 266 251.832secs na GPU), como pode ser visto no gráfico da figura 4.3 e na tabela 4.2. Para detalhar melhor o resultado foi criado

[5]O tempo de execução está representado em $lnt$ onde $t$ é o tempo de execução em segundos.

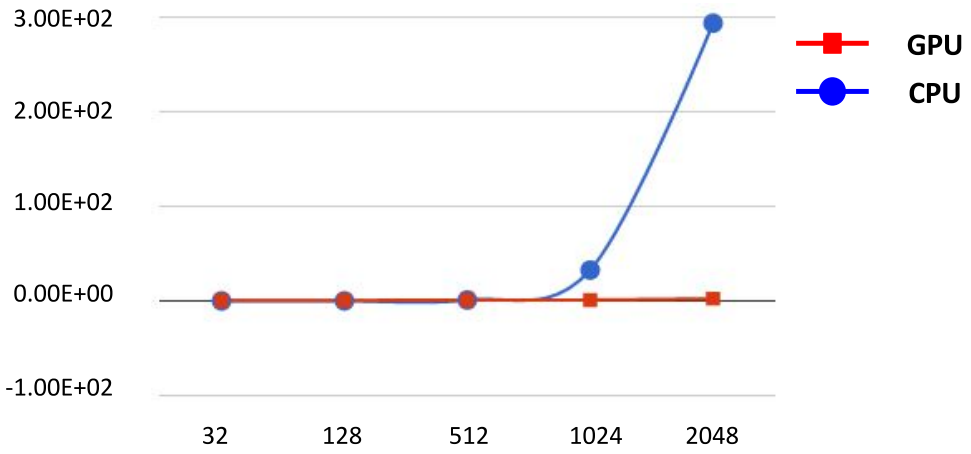


Figura 4.4: Gráfico: Tempo de execução da multiplicação de matrizes tridimensionais 3MM. Eixo x: Tempo de Execução; Eixo y: Tamanho da entrada.

o gráfico[5] 4.5. A aplicação *Gramschm* ou decomposição *Gram-Schmidt* é um método para normalização ortogonal de um conjunto de vetores em um espaço de produto interno. No resultado apresentado mesmo para valores grandes não se alcançou um ganho de desempenho na GPU. Este benchmark possui divergência de memória nas funções marcadas como paralelas. Porém a análise proposta não foi capaz de detectar todos os acessos uncoalesced, devido a forma de acesso ao vetor. Contudo após a modificação do benchmark a análise detectou na função principal 7 dos 10 acessos da função principal são divergentes.

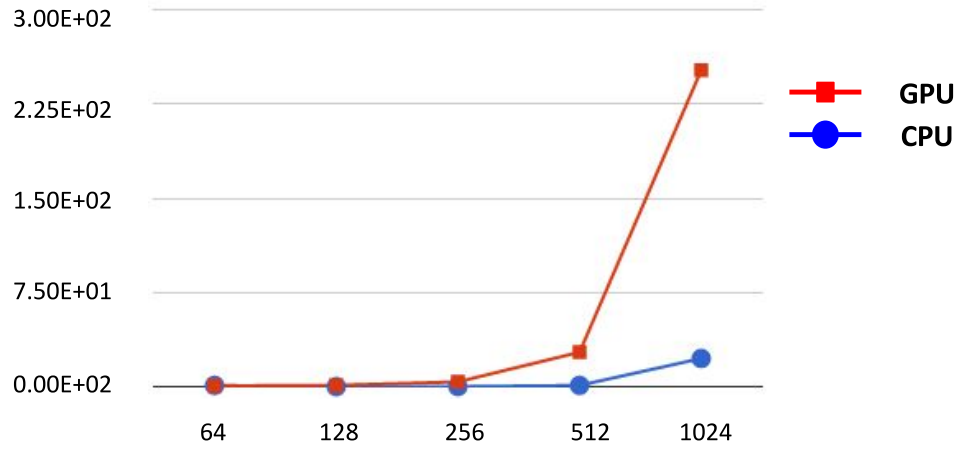


Figura 4.5: Gráfico: Tempo de execução da decomposição *Gram-Schimidt* Gramschm. Eixo x: Tempo de Execução; Eixo y: Tamanho da entrada.

**Impacto de divergências:** Como foi explicado anteriormente divergência de memória e de controle tem uma influência negativa no desempenho do programa ao ser executado

na GPU. Nos testes feitos com os *benchmarks* do *Polybench* pôde-se verificar que em alguns casos não se obtiveram o resultado esperado ao ser enviado o laço para execução na GPU. A Figura 4.5 exemplifica o impacto da divergência de memória em um programa.

Para determinar a porcentagem de divergência que é capaz de reduzir o desempenho do programa ao ser enviado para a GPU, foram executados testes em alguns benchmarks. A divergência de controle e a divergência de memória foram analisadas separadamente. Para se determinar este impacto foram ignoradas as demais análises e para identificação do laço a ser paralelizado, foi considerado o laço mais externo. A seguir são apresentados dois testes realizados. No primeiro teste foi considerado a divergência de controle e no segundo a divergência de memória.

1. Na Figura 4.6, é mostrado o *benchmark* Convolução 3D disponível no pacote do Polybench. Utilizando a ferramenta não foi possível identificar laços na função primcipal como paralelos. Para executar os testes e garantir a confiabilidade dos mesmos, foi realizada a análise manual e como os laços não possuem dependências os mesmos foram anotados como paralelos. Nesses laços foi executada a análise de divergências e a mesma foi capaz de identificar 85% de instruções divergentes. Como pode ser visto no gráfico mesmo para matrizes com tamanhos acima de 1024 o tempo de execução na GPU é muito superior ao tempo de execução na CPU. Logo, conclui-se que divergiencia de controle possui um impac†o negativo no tempo de execução do programa ao ser enviado para a GPU [6].

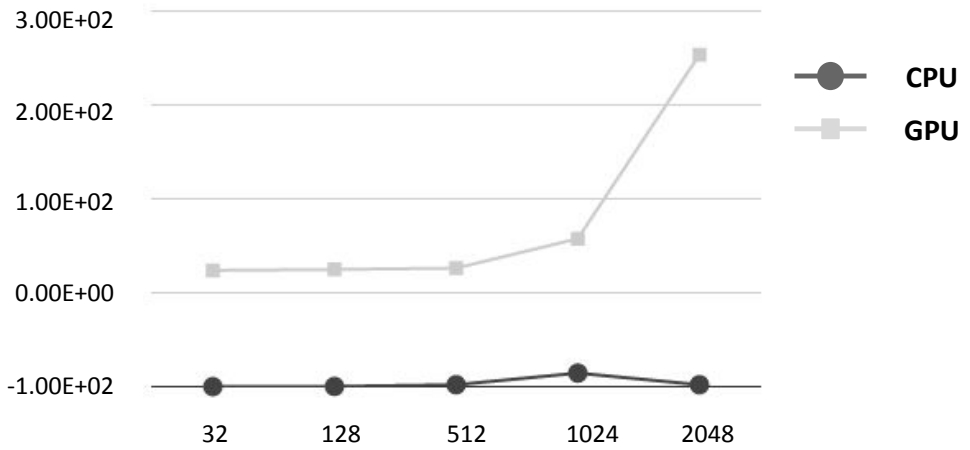


Figura 4.6: Gráfico: Tempo de execução da Convolução 3D *3DConv*. Eixo x: Tamanho da entrada; Eixo y: Tempo de execução.

[6]Foram realizados outros testes para definir que a porcentagem de 80% seria a considerada limite para o não envio do laço para a GPU

2. Na Figura 4.7, é mostrado o *benchmark K-Nearest*, ou K vizinhos mais próximos, o qual está disponível no pacote MGBench. No arquivo original devido a forma de acesso do vetor não é possível identificar que o acesso é não agrupado (*uncoalesced*). Devido a isto foi alterado a forma de acesso aos arranjos utilizados e a análise proposta foi capaz de identificar **70%** de divergência na função principal. No gráfico apresentado são mostrados três tempos de execução: (*i*) tempo de execução na CPU; (*ii*) tempo de execução na GPU; e (*iii*) o tempo de excução na GPU do programa modificado. Nesse último caso foi alterado a ordem dos laços dessa forma a análise de divergências retornou apenas 30% de acessos divergentes e como pode ser visto reduziu consideravelmente o tempo de execução.

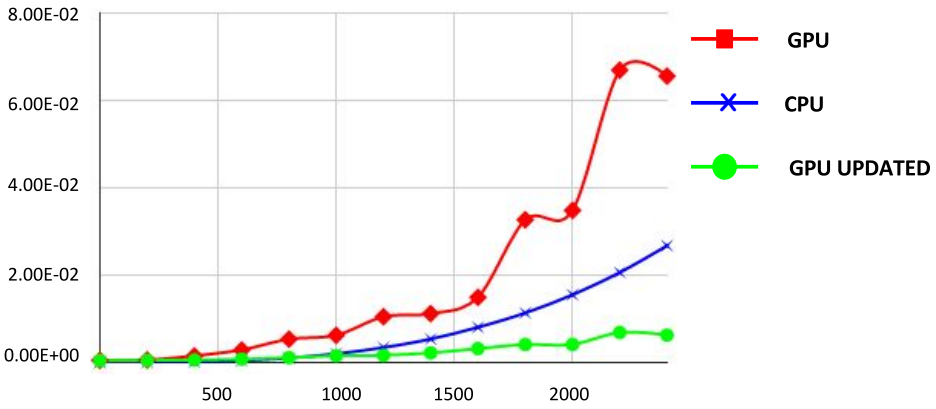


Figura 4.7: Gráfico: Tempo de execução do algoritmo que encontra os k vizinhos mais próximos *K-Nearest* Eixo x: Tamanho da entrada; Eixo y: Tempo de execução.

## 4.4 Discussão dos Resultados Experimentais

Nessa seção serão discutidos alguns tópicos pertinentes ao anotador proposto e os resultados obtidos.

***Benchmarks*** Atualmente existem diversos *benchmarks* específicos para problemas de paralelização de código. Nesse trabalho algumas análises foram executadas em grandes *benchmarks* como o Rodinia por exemplo, porém a ferramenta como um todo não foi testada no mesmo. Alguns aspectos foram decisivos para a escolha dos benchmarks utilizados, como por exemplo, o compilador utilizado possui uma limitação para o envio de vetores para a GPU. Apenas vetores unidimensionais são permitidos devido a isso, algumas adaptações foram feitas na análise que infere o tamanho dos arranjos, o que

impossibilitou o uso de adversos *benchmarks* publicamente disponíveis. Como uma verificação da ferramenta proposta e como um trabalho futuro, será realizado testes em *benchmarks* maiores como o SPEC.

**Número de laços paralelos** Devido a análise de paralelismo ser conservadora, o número de laços paralelos encontrados é ainda muito pequeno. Não é possível fazer muito com relação a isto, porém este fator reduziu o ganho de desempenho previsto para os *benchmarks* testados. Uma alternativa, utilizada em alguns testes realizados é deixar ao encargo do compilador de diretivas decidir se o laço pode ser paralelizado. Esta função já está disponível em diversos compiladores de diretivas disponíveis.

**Divergências** Divergências de controle e memória afetam o desempenho de um programa na GPU. Porém a análise de divergência de memória proposta ainda não é completa, ou seja, ela ainda não retorna todos os acessos divergentes. Ela possui varias restrições enquanto à forma de acesso ao arranjo e devido a isto os resultados não foram o esperado. Como foi exemplificado ela é capaz de medir o impacto no desempenho ao ser enviado um programa para a GPU, porêm ela ainda não está funcionando da melhor forma em conjunto com a ferramenta.

**Conclusão** Esta seção detalhou os resultados obtidos com o anotador proposto nesse trabalho. O processo de paralelização é totalmente estático e automático: é decidido quais 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 dessa seção mostram que a ferramenta proposta 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 foi possível paralelizar automaticamente 95% dos laços encontrados. Como consequência, foi observado *speedups* de até 121x. Resultado este surpreendente, visto que o uso da ferramenta não prevê encargos ao programador.

# Capítulo 5

# Conclusão

Este trabalho apresentou técnicas para decidir em que processadores, GPU ou CPU , cada parte paralela de um programa deve ser executada. Foi realizado uma pesquisa bibliográfica a respeito dos temas e conceitos abordados neste trabalho e um levantamento de trabalhos similares ao proposto. O anotador implementado é composto por três etapas: primeiro é verificado quais os laços são paralelos; após é identificado divergências de controle e memória e para finalizar é feito uma inferência do tamanho de arranjos e são inseridas diretivas no código fonte original.

De forma resumida, o anotador proposto é composto por um conjunto de análises estáticas capazes de anotar código C ou C++ sequencial com diretivas OpenACC, de forma totalmente automática, possibilitando sua execução em unidades de processamento gráfico. Essa abordagem é uma extensão da ferramenta Etino, lhe adicionando a capacidade de paralelizar código sem a necessidade de um perfilador. Além de ser capaz de identificar quais laços são paralelizáveis e quais poderão ter um desempenho melhor na GPU.

A técnica ora proposta foi implementada sobre o compilador LLVM e ela foi utilizada para paralelizar automaticamente diversos programas, obtendo ganhos de performance expressivos. Neste trabalho, foi utilizada 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 do *hardware* heterogêneo, a autora dessa dissertação acredita que ferramentas como a que foi proposta neste trabalho, capaz de facilitar significativamente a tarefa do programador, serão cada vez mais importantes na indústria de desenvolvimento de *software*.

## 5.1 Limitações

Algumas limitações foram encontradas no decorrer na implementação deste trabalho. Foi verificado que devido à análise que infere quais os laços podem ser paralelizados ser conservadora, ou seja, livre de falsos negativos, vários laços, paralelizáveis, não foram paralelizados. Devido a isto, e como pode ser visto nos testes apresentados no capítulo 4, alguns programas que teoricamente teriam um desempenho melhor na GPU, não apresentaram tal melhoria nos valores obtidos. Este problema afetou todas as análises presentes no anotador, visto que todas as análises apenas verificam laços paralelizáveis, reduzindo assim significativamente o resultado.

Outra limitação quanto às análises implementadas, foi a quantidade de acessos a memória *uncoalesced* pela análise de divergência de memória. Como essa análise não é completa, ou seja, retorna falsos negativos (ela ainda não é capaz de encontrar todos os acessos divergentes), apesar de não retornar falsos positivos. Isto não é um problema em si, visto que a ferramenta é totalmente estática.

Uma limitação com relação ao uso da ferramenta se dá à linguagem dos programas fontes de entrada e a linguagem do programa gerado. Atualmente a ferramenta só está disponível para programas escritos em C ou C++ e o código gerado é anotado apenas com diretivas OpenACC.

## 5.2 Trabalhos Futuros

A partir da pesquisa proposta nessa dissertação, diversos trabalhos podem ser criados. Além do uso das análises propostas para outro fim, algumas melhorias no anotador podem aprimorar substancialmente os resultados obtidos. O primeiro trabalho que já está sendo desenvolvido é a escolha do sistema de anotações a ser utilizado no programa de saída. Outra melhoria seria a substituição ou o *upgrade* da análise que identifica laços paralelos por outras análises disponíveis na indústria e na academia para ser capaz de identificar uma quantidade maior de laços, o que acarretará em um ganho significativo no desempenho dos programas gerados.

Em relação ao envio de todos os laços paralelos para o *hardware* paralelo, com apenas a restrição com relação à presença de divergência de controle e de memória. Outro trabalho a ser realizado é, uma espécie de integração do anotador proposto e a solução prévia, o Etino. A ideia seria identificar a complexidade de cada função, ou cada região, é inferir se a mesma obterá ganho de performance ao ser enviado para a GPU. A ferramenta Etino foi capaz de identificar que funções super lineares devem ser enviadas para a GPU, se for criado uma análise de complexidade estática ( livre do

uso de um perfilador), e associá-la às análises propostas neste trabalho. Os programas paralelizados terão um desempenho sempre melhor na GPU e a análise continuará estática e totalmente automática.

# Referências Bibliográficas


Aho, A. V.; Sethi, R. & Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

Allen, F. E. (1970). Control flow analysis. Em *Proceedings of a Symposium on Compiler Optimization*, pp. 1--19, New York, NY, USA. ACM.

Alves, P. R. O.; de Assis Costa, I. R.; Pereira, F. M. Q. & Figueiredo, E. L. (2012). Parameter based constant propagation. Em *SBLP*, pp. 46--60. SBC.

Amilkanthwar, M. & Balachandran, S. (2013). Cupl: A compile-time uncoalesced memory access pattern locator for cuda. Em *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pp. 459--460.

Amini, M.; Coelho, F.; Irigoin, F. & eryell, R. K. (2011). Static compilation analysis for host-accelerator communication optim ization. Em *LCPC*, pp. 237--251. Springer.

Bachmann, O.; Wang, P. S. & Zima, E. V. (1994). Chains of recurrences - a method to expedite the evaluation of closed -form functions. Em *In ISSAC*, pp. 242--249. ACM.

Bai, T.; Ding, C. & Li, P. (2015). Assessing safe task parallelism in spec 2006 int. Em *Cluster, Cloud and Grid Computing (CCGrid)*, pp. 402–411.

Cederman, D. & Tsigas, P. (2009). GPU-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics*, 14(1):4--24.

Che, S.; Boyer, M.; Meng, J.; Tarjan, D.; my W. Sheaffer, J.; Lee, S.-H. & Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. Em *IISWC*, pp. 44–54. IEEE.

Coppa, E.; Demetrescu, C. & Finocchi, I. (2012). Input-sensitive profiling. Em *PLDI*, pp. 89--98. ACM.

Coutinho, B.; Sampaio, D.; Pereira, F. M. Q. & Jr., W. M. (2011). Divergence analysis and optimizations. Em *PACT*, pp. 320--329. IEEE.

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

de Assis Costa, I. R.; Alves, P. R. O.; Santos, H. N. & Pereira, F. M. Q. (2013). Just-in-time value specialization. Em *CGO*, pp. 1--11. ACM.

do Couto Teixeira, D.; Andrade, K.; Souza, G. & Pereira, F. (2015a). Colocação automática de computação em hardware heterogêneo. Em *SBLP*. SBC.

do Couto Teixeira, D.; Andrade, K.; Souza, G. & Pereira, F. (2015b). Etino: Colocação automática de computação em hardware heterogêneo. Em *CBSoft Tools*. SBC.

Garland, M. (2008). Parallel computing experiences with CUDA. *IEEE Micro*, 28:13--27.

Gawlitza, T. M. & Monniaux, D. (2012). Invariant generation through strategy iteration in succinctly represented control flow graphs. *Logical Methods in Computer Science*, 8(3).

Ghike, S.; Gran, R.; Garzar án, M. J. & Padua, D. A. (2014). Directive-based compilers for gpus. Em Brodman, J. C. & Tu, P., editores, *LCPC*, volume 8967 of *Lecture Notes in Computer Science*, pp. 19--35. Springer.

Grauer-Gray, S.; Xu, L.; Searles, R.; Ayalasomayajula, S. & Cavazos, J. (2012). Auto-tuning a high-level language targeted to gpu codes. Em *InPar*, pp. 1–10. IEEE.

Gulavani, B. & Gulwani, S. (2008). A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. Em *CAV*, volume 5123 of *LNCS*, pp. 370–384. Springer.

Gulwani, S.; Mehra, K. K. & Chilimbi, T. (2009). Speed: Precise and efficient static estimation of program computational complexity. Em *POPL*, pp. 127--139. ACM.

Harris, M. J.; Coombe, G.; Scheuermann, T. & Lastra, A. (2002). Physically-based visual simulation on graphics hardware. Em *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 109--118. Eurographics Association.

Hennessy, J. L. & Patterson, D. A. (2011). *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5th edição.

Jaeger, J.; Carribault, P. & Pérache, M. (2015). Fine-grain data management directory for openmp 4.0 and openacc. *Concurrency and Computation: Practice and Experience*, 27(6):1528--1539.

Jalby, W.; Wong, D. C.; Kuck, D. J.; Acquaviva, J. T. & Beyler, J. C. (2012). Measuring computer performance. Em *High-Performance Scientific Computing - Algorithms and Applications.*, pp. 75--95. Springer Publishing Company, Incorporated.

Kao, C.-C. & Hsu, W.-C. (2015). An adaptive heterogeneous runtime framework for irregular application s. *J. Signal Process. Syst.*

Lattner, C. & Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. Em *CGO*, pp. 75–88. IEEE.

Lee, S.; Min, S.-J. & Eigenmann, R. (2009). Openmp to gpgpu: a compiler framework for automatic translation and opt imization. Em *PPoPP*, pp. 101--110. ACM.

Lee, S. & Vetter, J. S. (2012). Early evaluation of directive-based gpu programming models for produc tive exascale computing. Em *In CHPCNSA*, SC '12. IEEE.

Lee, S. & Vetter, J. S. (2014). Openarc: open accelerator research compiler for directive-based, efficient heterogeneous computing. Em *HPDC*, pp. 115--120. ACM.

Liu, D.; Yin, S.; Liu, L. & Wei, S. (2013). Polyhedral model based mapping optimization of loop nests for cgras. Em *DAC*, pp. 19:1–19:8, New York, NY, USA. ACM.

Meenderinck, C. & Juurlink, B. H. H. (2011). Nexus: Hardware support for task-based programming. Em *DSD*, pp. 442--445. Springer.

Meng, J.; Tarjan, D. & Skadron, K. (2010). Dynamic warp subdivision for integrated branch and memory divergence tolerance. Em Seznec, A.; Weiser, U. C. & Ronen, R., editores, *ISCA*, pp. 235--246. ACM.

Mu, S.; Zhang, X.; Zhang, N.; Lu, J.; Deng, Y. S. & Zhang, S. (2010). IP routing processing with graphic processors. Em *DATE*, pp. 93–98. IEEE.

Nickolls, J. & Dally, W. J. (2010). The GPU computing era. *IEEE Micro*, 30:56--69.

Planas, J.; Badia, R. M.; Ayguadé, E. & Labarta, J. (2015). Ssmart: Smart scheduling of multi-architecture tasks on heterogeneous systems. Em *In WACCPD*, pp. 1:1--1:11. ACM.

Prabhu, T.; Ramalingam, S.; Might, M. & Hall, M. (2011). EigenCFA: Accelerating flow analysis with GPUs. Em *POPL*. ACM.

Ragan-Kelley, J.; Barnes, C.; Adams, A.; P aris, S.; Durand, F. & Amarasinghe, S. (2013). Halide: A language and compiler for optimizing parallelism, locality , and recomputation in image processing pipelines. Em *PLDI*, pp. 519--530. ACM.

Saito, H.; Gaertner, G.; Jones, W.; Eigenmann, R. d.; Iwashita, H.; Lieberman, R.; van Waveren, M. & Whit ney, B. (2003). Large system performance of spec omp benchmark suites. *Int. J. Parallel Program.*, 31:197--209.

Sampaio, D.; de Souza, R. M.; Collange, S. & Pereira, F. M. Q. (2013). Divergence analysis. *ACM Trans. Program. Lang. Syst.*, 35(4):13.

Sampaio, D.; de Souza, R. M.; Collange, S. & Pereira, F. M. Q. (2014). Divergence analysis. To appear.

Sandes, E. F. O. & de Melo, A. C. M. (2010). Cudalign: using gpu to accelerate the comparison of megabase genomic sequences. Em *PPoPP*, pp. 137--146. ACM.

Standard, O. (2013). The openacc programming interface. Relatório técnico, CAPs.

Stratton, J. A.; Rodrigues, C.; Sung, I.-J.; Ob eid, N.; Chang, L.-W.; Anssari, N.; Liu, G. D. & Hwu, W.-m. (2012). Parboil: A revised benchmark suite for scientific and commercial thro ughput computing. *In CRHPC*.

Wienke, S.; Springer, P. L.; Terboven, C. & an Mey, D. (2012). OpenACC - first experiences with real-world applications. Em *Euro-Par*, pp. 859--870. Springer.

Zhang, M.; Biswas, S. & Bond, M. D. (2016). Relaxed dependence tracking for parallel runtime support. Em *In CC*, pp. 45--55. ACM.

Zhang, Y.; Cohen, J. & Owens, J. D. (2010). Fast tridiagonal solvers on the gpu. Em *PPoPP*, pp. 127--136. ACM.

# Apêndice A

# Glossário

**ALU** Do inglês *Arithmetic Logic Unit*, ou Unidade Lógica aritmética é um circuito digital usado para executar operações lógicas e aritméticas.

**CBSoft** Congresso Brasileiro de *Software*.

**CPU** Do Inglês *Central Processing Unit* ou Unidade Central de Processamento.

**CUDA** Sigla para *Compute Unified Device Architecture*, é uma extensão para a linguagem de programação C, a qual possibilita o uso de computação paralela.

**GPGPU** Sigla para *General Purpose Graphics Processing Unit* ou Unidade de Processamento Gráfico de Propósito Geral.

**GPU** Sigla para *Graphics Processing Unit* ou Unidade de Processamento Gráfico.

**HMPP** Sigla para *Hybrid Multicore Parallel Programming* ou Programação Paralela MultiCore Híbrida.

**IP** Sigla para *Internet Protocol* é um número que todo computador (ou roteador) recebe quando se conecta à Internet.

**LLVM** Sigla para *Low Level Virtual Machine* é uma infraestrutura de compilador escrita em C++, desenvolvida para otimizar em tempos de compilação, ligação e execução de programas escritos em linguagens de programação variadas.

**Metadados** Metadados do inglês *Metadata* representa uma informação opcional sobre uma instrução, ou módulo, que pode ser descartada sem interferir na corretude do programa `http://llvm.org/`.

**NVCC** Do inglês *NVidia CUDA Compiler* é um compilador proprietário criado pela NVidia para o uso de CUDA.

**NVPTX** O LLVM possui um *back-end* para programação em GPUs chamado NVPTX `http://llvm.org/docs/NVPTXUsage.html`.

**PGCC** Um compilador fechado desenvolvido pelo *Portland Group* (PGI). Sigla para *Portland Group C Compiler*.

**PTX** É uma linguagem intermediária produzida pela compilação de um programa em CUDA com o NVCC.

**RMS** Sigla para *Read Memory Size* ou o tamanho da entrada lida. De acordo com [Coppa et al., 2012] o tamanho da entrada de um programa é definido como a quantidade de posições de memória que ele lê sem antes escrevê-las.

**SBLP** Simpósio Brasileiro de Linguagem de Programação.

**SIMD** Sigla para *Single Instruction Multiple Data*, traduzindo única instrução múltiplos dados. O SIMD é um tipo de arquitetura utilizado em processadores paralelos.

**SPEC** Sigla para *Standard Performance Evaluation Corporation* , seus *benchmarks* são amplamente utilizados para avaliar desempenho de sistemas.

**Warp** Warp, termo criado pela NVIDIA, é um grupo de *threads*.