UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUAN BODNER DO ROSÁRIO

FERRAMENTA DE COMPILAÇÃO PARA A GERAÇÃO DE CÓDIGO PARA UM *RUNTIME* DE *OFFLOADING* AUTOMÁTICO

MONOGRAFIA

CAMPO MOURÃO

2019

LUAN BODNER DO ROSÁRIO

FERRAMENTA DE COMPILAÇÃO PARA A GERAÇÃO DE CÓDIGO PARA UM *RUNTIME* DE *OFFLOADING* AUTOMÁTICO

Trabalho de Conclusão de Curso de Graduação apresentado à disciplina de Trabalho de Conclusão de Curso II, do Curso de Bacharelado em Ciência da Computação do Departamento Acadêmico de Computação da Universidade Tecnológica Federal do Paraná, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Rogério Aparecido Gonçalves

CAMPO MOURÃO
2019

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Campo Mourão
Curso de Bacharelado em Ciência da Computação

UTFPR UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

ATA DE DEFESA DO TRABALHO DE CONCLUSÃO DE CURSO

Às **15:50** do dia **27 de junho de 2019** foi realizada na sala **E104** da UTFPR-CM a sessão pública da defesa do Trabalho de Conclusão do Curso de Bacharelado em Ciência da Computação do(a) acadêmico(a) **Luan Bodner do Rosário** com o título **FERRAMENTA DE COMPILAÇÃO PARA A GERAÇÃO DE CÓDIGO PARA UM RUNTIME DE OFFLOADING AUTOMÁTICO**. Estavam presentes, além do(a) acadêmico(a), os membros da banca examinadora composta por: **Prof. Dr. Rogério Aparecido Gonçalves** (orientador), **Prof. Dr. Diego Bertolini Gonçalves** e **Prof. Ms. Paulo Cesar Gonçalves**. Inicialmente, o(a) acadêmico(a) fez a apresentação do seu trabalho, sendo, em seguida, arguido(a) pela banca examinadora. Após as arguições, sem a presença do(a) acadêmico(a), a banca examinadora o(a) considerou ____________ na disciplina de Trabalho de Conclusão de Curso **2** e atribuiu, em consenso, a nota _____ (______________). Este resultado foi comunicado ao(à) acadêmico(a) e aos presentes na sessão pública. A banca examinadora também comunicou ao acadêmico(a) que este resultado fica condicionado à entrega da versão final dentro dos padrões e da documentação exigida pela UTFPR ao professor Responsável do TCC no prazo de **onze dias**. Em seguida foi encerrada a sessão e, para constar, foi lavrada a presente Ata que segue assinada pelos membros da banca examinadora, após lida e considerada conforme.

Observações: ____

Campo Mourão, **27 de junho de 2019**

Prof. Dr. Diego Bertolini Gonçalves
Membro 1

Prof. Ms. Paulo Cesar Gonçalves
Membro 2

Prof. Dr. Rogério Aparecido Gonçalves
Orientador

A ata de defesa assinada encontra-se na coordenação do curso.

Agradecimentos

Agradeço à minha família pelo suporte durante todo este processo, me proporcionando as condições para que eu possa concluir mais essa etapa.

Agradeço ao meu orientador Rogério Gonçalves, que me passou o conhecimento necessário para estudar essa área da computação e me auxiliou durante o desenvolvimento deste trabalho.

Por fim agradeço à todos os meus professores, amigos e colegas que influenciaram minhas escolhas e minha vida acadêmica direta ou indiretamente.

Muito obrigado à todos!

Resumo

ROSÁRIO, Luan Bodner do. Ferramenta de Compilação para a Geração de Código para um *Runtime* de *Offloading* Automático. 2019. 79. f. Monografia (Curso de Bacharelado em Ciência da Computação), Universidade Tecnológica Federal do Paraná. Campo Mourão, 2019.

Computação Heterogênea é uma área da Computação que desperta grande interesse tanto no setor comercial quanto no acadêmico. Essa área engloba o conhecimento sobre nós computacionais compostos por dispositivos aceleradores(`GPUs`) e processadores *multicore*. Apesar de populares, poucas aplicações são capazes de utilizar todo o poder computacional disponibilizado por essas plataformas heterogêneas.
Diante desse cenário, um ambiente de execução havia sido desenvolvido para tentar solucionar este problema. Este ambiente toma como entrada funções com versões destinadas à `CPU` e à `GPU` e é capaz de controlar a execução das *threads* `OpenMP` coletando contadores de desempenho dentro do processador. Com essas informações a decisão de *offloading* de código é tomada em tempo de execução.
Neste projeto foi desenvolvida uma ferramenta de compilação com o propósito de trabalhar em conjunto com este ambiente de execução, sendo capaz de detectar regiões demarcadas do código como paralelizáveis, partindo de um código que inicialmente seria executado sequencialmente. Após a detecção, a ferramenta gera versões do código que pode ser executado tanto em `CPU` *multicore* usando chamadas de funções `OpenMP` quanto em `GPUs` por meio da plataforma de processamento paralelo *Compute Unified Device Architecture* (CUDA).
Para comprovar as funcionalidades desta ferramenta, foram produzidos um conjunto de provas de conceito que mostram passo à passo o funcionamento do compilador, as transformações feitas durante o processo e a saída produzida pelo ambiente de execução com o propósito de mostrar a viabilidade de um sistema capaz de transformar código legado e explorar a arquitetura heterogênea de destino com poucas modificações no código original.
Palavras-chaves: ferramenta de compilação, computação heterogênea, *multicore*, dispositivos aceleradores, *offloading*, ambiente de execução

Abstract

ROSÁRIO, Luan Bodner do. . 2019. 79. f. Monograph (Undergradute Program in Computer Science), Federal University of Technology – Paraná. Campo Mourão, PR, Brazil, 2019.

Heterogeneous computing is an area in computer science that has become of great interest both comercially and academically. This area encompasses our understanding of computer nodes formed by both accelerating devices, such as GPUs and multi core processors. Although popular, very few applications are able to use all of the computing power available in these heterogeneous platforms.
With these questions in mind, an execution enviroment was developed to solve this problem. This enviroment takes as input blocks of code (functions) with two different versions: one version destined to the GPU and another version destined to the CPU. This enviroment is capable of controlling the threads in the CPU and use them to collect data to decide which device is best suited for a specific function inside of the application.
In this project, we attempted to develop a compiler tool with the purpose of working with this execution enviroment. The tool is able to detect if a demarcated region of code is parallel, starting from a code originally sequencial. After that, the tool generates code for both CPU by means of function calls to the `OpenMP` library and GPU by means of `CUDA`.
To show that the compiler tool is functional, we produced a set of conceptual proofs that show the step-by-step of the tool by explaining what happened during the compilation process, the transformations made during each step and finally the output generated by the enviroment, proving that the system created and developed here is viable for improving the efficiency of code with very little changes in the original source code.
Keywords: compiler tool, heterogeneous computing, multi core, accelerating devices, offloading, execution environment

Lista de figuras

2.1 Evolução das *Application Programming Interfaces* (APIs) disponíveis para *Unidade de Processamento Gráficos* (GPUs). Fonte: (BRODTKORB et al., 2012) . . . 18
2.2 Comunicação *Central Processing Unit* (CPU)-GPU. Baseado em (GONçALVES, 2014) . . . 19
2.3 Arquitetura Fermi. Baseado em (BRODTKORB et al., 2012) . . . 20
2.4 *Streaming Processor* da Arquitetura Fermi. Baseado em (BRODTKORB et al., 2012) . . . 21
2.5 Paralelismo Dinâmico. Fonte: (NVIDIA, 2012) . . . 22
2.6 Arquitetura SMX. Baseado em (NVIDIA, 2012) . . . 23
2.7 *Chip* completo Kepler. Baseado em (NVIDIA, 2012) . . . 24
2.8 Maxwell GM107. Baseado em (NVIDIA, 2014) . . . 25
2.9 Maxwell *Streaming Multiprocessor Maxwell* (SMM). Baseado em (NVIDIA, 2014) . . . 26
2.10 Modelo Clássico do Fluxo de Execução de um *kernel*. Baseado em (GONçALVES, 2014) . . . 27
2.11 *Grid* com 64 *threads*. Baseado em (STRINGHINI et al., 2012) . . . 30
2.12 Linearização para o endereço do vetor. . . . 30
2.13 Fluxo de execução de uma região paralela. . . . 31
2.14 Grafo de Fluxo de Controle (GFC) com região paralelizável destacada. Fonte (GONçALVES, 2014). . . . 38

3.1 Execução do Código *Open Multi-Processing* (OpenMP). Baseado em (GONçALVES, 2016) . . . 44
3.2 Arquitetura do ambiente de execução. Fonte: (GONçALVES, 2016) . . . 45

4.1 *Overview* da arquitetura da ferramenta de compilação. . . . 49
4.2 Código não canônico. . . . 54
4.3 Código canônico. . . . 54
4.4 Representação gráfica do *Static Control Part* (SCoP) do Código 4.6. . . . 55
4.5 Código 4.7 otimizado pelo PoLLy. . . . 56

4.6 Função (`foo_WITH_GPU_for.cond1.preheader`) e sua função mãe. 57

5.1 Lista de *threads* na *cpu* geradas pelo programa. 66
5.2 Lista de *threads* na `GPU` geradas pelo programa. 67
5.3 Região paralelizável detectada na função `add_matrix`. 68
5.4 Região paralelizável detectada na função `mm31`. 70
5.5 Lista de *threads* na `CPU` geradas pela multiplicação de matrizes. 71
5.6 Lista de *threads* na `GPU` geradas pela multiplicação de matrizes. 72

Lista de tabelas

2.1 Diferença entre *Schedule* estático e *schedule* dinâmico. Fonte: (STRINGHINI et al., 2012) 32

3.1 Lista de eventos capturados pelo *Performance Application Programming Interface* (PAPI) 47

3.2 Taxas de largura de banda e latência para os tipos de alocação de memória na CPU 48

3.3 Taxas de largura de banda e latência para os tipos de alocação de memória na GPU 48

Siglas

- API: *Application Programming Interface*
- CPU: *Central Processing Unit*
- CUDA: *Compute Unified Device Architecture*
- FPGA: *Field Programmable Gate Array*
- GFC: Grafo de Fluxo de Controle
- GPC: *Graphic Processing Cluster*
- GPU: *Unidade de Processamento Gráfico*
- GPUs: *Unidades de Processamento Gráfico*
- LLVM: *Low Level Virtual Machine*
- LLVM-IR: *Low Level Virtual Machine Intermediate Representation*
- MIMD: *Multiple Instruction Multiple Data*
- MISD: *Multiple Instruction Single Data*
- OpenCL: *Open Computing Language*
- OpenGL: *Open Graphics Library*
- OpenMP: *Open Multi-Processing*
- PAPI: *Performance Application Programming Interface*
- PPCG: *Polyhedral Parallel Code Generation*
- SCoP: *Static Control Part*
- SFU: *Special Function Unit*
- SIMD: *Single Instruction Multiple Data*
- SIMT: *Single Instruction Multiple Threads*
- SISD: *Single Instruction Single Data*
- SM: *Streaming Multiprocessor*
- SMM: *Streaming Multiprocessor Maxwell*
- SMX: *Streaming Multiprocessor Kepler*
- SPMD: *Single Program Muliple Data*

Sumário

1 Introdução **11**
1.1 Problema de Pesquisa 12
1.2 Proposta e Metodologia 12
1.3 Objetivos 13
1.4 Principais Contribuições 13
1.5 Organização do Texto 14

2 Referencial Teórico **15**
2.1 Terminologia 15
2.2 Unidades de Processamento Gráfico (`GPU`) 17
2.3 Modelo de Programação - `CPU` e `GPU` 24
2.3.1 `CUDA`: *Threads* na `GPU` 25
2.3.2 `OpenMP` : Paralelismo em `CPUs` *multicore* 29
2.4 Ferramentas Relacionadas 32
2.4.1 `LLVM` 33
2.4.2 `PoLLy` 35
2.4.3 `PPCG` 39
2.4.4 `KernelGen` 42

3 Ambiente de Execução **43**
3.1 Funcionamento do Ambiente 43
3.2 Coleta de Métricas 46

4 Desenvolvimento **49**
4.1 Pré-processamento 50
4.2 Detecção de regiões paralelas 53
4.2.1 Reorganização (canonicalização) do código 53
4.2.2 Detecção da área paralelizável 53
4.3 Extração do código em funções externas 54
4.3.1 Encapsulamento do código paralelo em uma função distinta 55
4.3.2 Criação de um novo módulo para a função 57
4.4 Geração de código para `CPU` e `GPU` 58

4.4.1 Geração do código destinado à CPU 58
4.4.2 Geração do código para a GPU 60
4.5 Reorganização do código e compilação final 62
4.5.1 Reorganização do código 62
4.5.2 Geração dos executáveis 62
4.6 Conclusões 63

5 Experimentos e Resultados **64**
5.1 Casos de Teste e Experimentos 64
5.1.1 Adição de Matrizes 65
5.1.2 Multiplicação de Matrizes 69
5.2 Testes com o Ambiente de Execução 72
5.3 Problemas de Desenvolvimento 73

6 Conclusões **75**
6.1 Trabalhos Futuros 76

Referências **77**

Capítulo 1

Introdução

Computação Heterogênea é a área da Computação que engloba o conhecimento e a utilização de nós computacionais que apresentam em sua composição elementos de processamento com arquiteturas distintas. Dentre esses elementos heterogêneos, temos as `GPUs`, os dispositivos de arquiteturas reconfiguráveis como os ***Field Programmable Gate Arrays* (FPGAs)** (BACON et al., 2013) e os arranjos de coprocessadores, como o `Xeon Phi` da Intel (INTEL, 2017). Esses elementos sendo utilizados em conjunto com `CPUs` *multicore* proporcionam aos usuários e aplicações um potencial ganho de desempenho (STRINGHINI et al., 2012).

Quanto aos dispositivos aceleradores comercializados para o grande público, as `GPUs` fabricadas pela NVIDIA podem ser consideradas os mais populares (BRODTKORB et al., 2012), sendo o foco de desenvolvedores de *software* e *hardware* no mundo todo.

Além do sucesso comercial, `GPUs` são utilizadas por uma grande parte dos supercomputadores presentes na lista dos computadores mais poderosos do mundo atualmente, a `top500.org` (TOP500, 2019). Por exemplo, o maior supercomputador americano, o *Summit*, possui uma arquitetura heterogênea que teoricamente é capaz de realizar operações à 148.6 *petaflops* (TOP500, 2019).

Os supercomputadores exploram o paralelismo dessas arquiteturas heterogêneas na tentativa de extrair o máximo do poder de processamento disponível. Esses dados sugerem que o uso de arquiteturas híbridas exploradas por aplicações capazes de utilizar o paralelismo da máquina de forma eficiente são a base para pesquisas em Computação Paralela (GONçALVES, 2014).

1.1. Problema de Pesquisa

Para que o poder computacional dessas plataformas heterogêneas seja utilizado de forma que ocorram melhorias na performance, os dispositivos aceleradores e a CPU devem ser utilizados em conjunto explorando o paralelismo do código quando possível, de forma que as tarefas de uma aplicação sejam executadas tanto em CPU quanto em GPU paralelamente.

Apesar da importância de *softwares* que utilizem esses nós computacionais de forma eficiente, sendo que o problema a ser resolvido seja adequado para a arquitetura em que ele está sendo executado de forma que a performance não seja comprometida ou limitada, são bastante raros, visto que tais arquiteturas não dão suporte direto para desenvolvedores (GONçALVES, 2014).

Dado este problema, é necessário que, para extrairmos o máximo da heterogeneidade dos dispositivos modernos, os programas existentes para arquiteturas simples sejam completamente reescritos para que o paralelismo do dispositivo seja explorado e a divisão de trabalho entre CPU e GPU aconteça (GONçALVES, 2014). Por si só, reescrever programas inteiros já seria extremamente trabalhoso, mas é necessário levar em consideração também a complexidade de desenvolvimento de novos programas para estas plataformas, que requerem o aprendizado de novas linguagens e a utilização de *kits* de desenvolvimentos específicos da plataforma, o que não é um trabalho trivial (GONçALVES, 2014).

Para que o desenvolvimento de novas aplicações e o uso de aplicações de código legado seja facilitado, é necessário remover o envolvimento de desenvolvedores o máximo possível, evitando a necessidade de aprendizado de novos paradigmas e diminuindo os riscos existentes no desenvolvimento de um *software* desta complexidade (GROSSER, 2016). Isso pode ser alcançado por meio de ferramentas que automaticamente paralelizem código para plataformas heterogêneas, fazendo uso dos processadores *multicore* e GPUs.

1.2. Proposta e Metodologia

Para resolver o problema de complexidade de desenvolvimento, Gonçalves (2014) propôs uma ferramenta que deve ser capaz de transformar um código inicialmente sequencial em paralelo e fazer a divisão de trabalho entre GPU e processador. Esta aplicação é dividida em duas partes: uma **ferramenta de compilação**, responsável pela geração de código que possa ser utilizado em um **ambiente de execução**, que por sua vez utiliza o código transformado e faz a sua distribuição entre os dispositivos do nó computacional. Em Gonçalves (2016) o ambiente de execução foi desenvolvido, porém a ferramenta de compilação não foi desenvolvida, sendo que as transformações de código foram feitas por meio de *scripts* e o código de entrada, que consiste em código OpenMP (CHAPMAN et al., 2007) e código que será executado na GPU (GONçALVES, 2014), foram preparados manualmente.

Para suprir esta necessidade, este trabalho foca no desenvolvimento de uma ferramenta de compilação na tentativa de prover uma entrada para este ambiente de execução. Inicialmente, foi feito um estudo na literatura existente sobre as ferramentas disponíveis para que elas possam ser utilizadas como base de desenvolvimento e modelo para a ferramenta aqui produzida. Após este processo, foi determinado que a ferramenta `PoLLy` era a mais adequada para servir como base direta para este projeto.

As transformações no código que precisam ser feitas para a geração da saída necessária para que ela seja utilizada pelo *runtime* foram definidos em Gonçalves (2014), por meio do uso de *scripts*. Esses passos serviram como base para a definição do modelo de desenvolvimento da ferramenta em si. A partir deste modelo foram utilizadas as funções e variáveis de ambiente determinadas dentro do projeto `PoLLy` para analisar e transformar o código de forma equivalente aos *scripts*.

1.3. Objetivos

Os principais objetivos que buscamos alcançar com o desenvolvimento deste trabalho e, por consequência, da ferramenta desenvolvida são:

- Desenvolver um estudo dentro da área de Computação Heterogênea;
- Facilitar o processo de desenvolvimento de aplicações para plataformas heterogêneas;
- Criação de uma ferramenta que pode ser expandida no futuro e é capaz de gerar código tanto para a `CPU` quanto a `GPU` simultaneamente;

1.4. Principais Contribuições

A contribuição maior da ferramenta aqui desenvolvida foi criar um *software* capaz de fazer geração de código que possa explorar o poder de processamento tanto de `CPUs` quanto de `GPUs`, removendo a necessidade do programador se envolver diretamente com os *kits* de desenvolvimento.

Anteriormente ao desenvolvimento da ferramenta, a intenção era de que o desenvolvedor não precisasse modificar nenhuma linha de código, porém foi necessário adicionar um conjunto de diretivas que auxiliassem na detecção de regiões paralelas. Essas diretivas são mínimas e não requerem que o programador conheça nenhuma parte do funcionamento do compilador.

Essa ferramenta pode, como citado anteriormente, pode ser expandida no futuro de forma que outros ambientes de execução ou novas diretivas possam ser adicionadas, melhorando o suporte para a linguagem C e criando suporte para novas linguagens.

1.5. Organização do Texto

Neste capítulo, introduzimos nosso problema de pesquisa, a ferramenta que foi desenvolvida, bem como os objetivos e contribuições deste trabalho. No Capítulo 2 apresentamos os conceitos relevantes para o desenvolvimento do trabalho. No Capítulo 3 apresentamos os detalhes do ambiente de execução em maior profundidade. O Capítulo 4 é destinado para a explicação sobre o processo e desenvolvimento da ferramenta de compilação. No Capítulo 5 mostramos os resultados alcançados pelo compilador e por fim no Capítulo 6 apresentamos algumas conclusões obtidas com a implementação e execução de experimentos.

Capítulo 2

Referencial Teórico

O modelo arquitetural dos processadores modernos possui características limitantes quanto à evolução do poder de processamento desses dispositivos. Tanto a quantidade de transistores utilizados para memória *cache* quanto o limite de frequência imposto pelo material utilizado na fabricação impedem a melhoria contínua e sustentável da performance desses processadores (BRODTKORB et al., 2010).

Arquiteturas heterogêneas fornecem uma alternativa à esse modelo, combinando diversos tipos de dispositivos aceleradores com `CPUs` *multicore*. Esses dispositivos são capazes de aumentar a capacidade de processamento de forma mais sustentável, utilizando aceleradores que diminuem a complexidade dos núcleos de uma `CPU`, mas são capazes de melhorar drasticamente a performance de algoritmos que se beneficiam do paralelismo do dispositivo em que está sendo executado (BRODTKORB et al., 2010; GONçALVES, 2014).

Como o trabalho aqui desenvolvido baseia-se nos trabalhos desenvolvidos em Gonçalves (2016) e (GONçALVES, 2014), que por sua vez focaram sobre as `GPUs` fabricadas pela NVIDIA, neste Capítulo, faremos um estudo sobre estes dispositivos, explorando os detalhes de arquiteturas como a `Fermi`, `Maxwell`, `Kepler` e `Pascal`. O estudo dessas arquiteturas se faz necessário dado que determinados detalhes do funcionamento das `GPUs` precisam entendidos para que o modelo de programação paralelo para esses dispositivos faça sentido. Após a análise dos modelos arquiteturais, discutiremos o modelo de programação paralela tanto em nível de `CPU` quanto de `GPU`.

2.1. Terminologia

A taxonomia de Flynn (FLYNN, 1972) é utilizada para descrever e categorizar as diferentes organizações heterogêneas a partir de uma visão macroscópica, dado que não é feita nenhuma referência a um sistema específico de forma detalhada, das possíveis interações entre dados e

instruções dentro do sistema (FLYNN, 1972).

Embora essa taxonomia não abranja todas as arquiteturas existentes, os termos aqui descritos ainda são amplamente utilizados para descrever os diferentes tipos de paralelismo encontrados atualmente (GONçALVES, 2014).

- ***Single Instruction Single Data*** **(SISD)**: Esse tipo de arquitetura não possui paralelismo, já que trabalha com um único fluxo de instruções manipulando apenas um fluxo de dado sequencial;
- ***Single Instruction Multiple Data*** **(SIMD)**: Arquitetura que utiliza seus núcleos para executar o mesmo programa, ou seja, existe apenas um fluxo de instruções em diferentes fluxos de dados (vetorização de instruções, por exemplo);
- ***Multiple Instruction Single Data*** **(MISD)**: Máquinas que são capazes de fazer a execução de diferentes fluxos de instruções (vários programas) mas não apresentam paralelismo de fluxo de dados, sendo que os programas são executados sequencialmente;
- ***Multiple Instruction Multiple Data*** **(MIMD)**: Similar ao MISD, essas máquinas são capazes de executar diversos programas, mas neste caso também são capazes de executar diversos fluxos de dados.

Outra forma de classificar essas arquiteturas é por meio da análise de como o compartilhamento de memória é feito. Essa forma de classificação está diretamente ligada ao modelo de programação utilizado para o desenvolvimento nessas plataformas (STRINGHINI et al., 2012):

- **Multi-processador**: Refere-se às arquiteturas/ sistemas que possuem diversos processadores em uma única maquina dividindo memória e outros recursos;
- ***Multicore***: Similar ao multi-processador no fato de que recursos são compartilhados entre múltiplos dispositivos, mas neste caso um processador único possui diversos núcleos que compartilham *last level cache*, o que torna a comunicação ainda mais barata;
- **Paralelismo em nível de instrução**: Quando um processador consegue fazer a execução de várias instruções em paralelo. Processadores superescalares exploram o paralelismo nesse nível utilizando escalonamento dinâmico de instruções e execução fora de ordem.
- ***Multi-thread***: Este tipo de paralelismo é exemplificado quando um *core* é capaz de executar diferentes contextos, resolvendo tarefas menores dentro de um programa de forma simultânea ou intercalada à outros fluxos de execução;

Por fim, existe ainda classificações considerando o nível de abstração:

- ***Single Program Muliple Data*** **(SPMD)** : Similar ao modelo SIMD, porém utiliza aplicações com paralelismo de dados e *threads*;

- ***Single Instruction Multiple Threads* (SIMT)** : Modelo utilizado pela NVIDIA, gerenciando a execução de várias *threads* no acelerador. Essas *threads* executam as mesmas instruções dentro do código lançado para o acelerador.

2.2. Unidades de Processamento Gráfico (GPU)

Nesta seção, estudaremos o principal dispositivo acelerador disponível, tanto para o uso pessoal quanto acadêmico que podem alcançar níveis de performance muito superiores às CPUs, as GPUs (STRINGHINI et al., 2012).

Inicialmente, a utilização de GPUs era voltada unicamente para operações gráficas. O dispositivo era responsável pela renderização de objetos geométricos de um ambiente virtual 3D para imagens 2D. Cada *pixel* dessa imagem 2D eram processados pela GPU que, de forma paralela, definia quais cores seriam utilizadas em cada um desses elementos. Desenvolver aplicações gráficas para essas plataformas era extremamente difícil, devido à existência de *bugs* no compilador e a deficiência de ferramentas para *debug* (BRODTKORB et al., 2012).

Com o passar do tempo, pesquisas feitas sobre o *hardware* levaram à novas linguagens de alto nível que abstraíram as funções gráficas, facilitando o desenvolvimento de aplicações. Essas pesquisas também fizeram com o que os desenvolvedores percebessem o potencial de processamento dessas GPUs e que esses dispositivos poderiam ser utilizados em conjunto com CPUs para melhorar o processamento de certos algoritmos, como o de Monte Carlo, que se beneficiam com o uso de muitos núcleos de baixa frequência executando ao mesmo tempo (GONçALVES, 2014; BRODTKORB et al., 2012, 2010; KAUER, 2013).

A partir dessas descobertas, empresas desenvolvedoras de GPUs criaram linguagens que permitiram programação não gráfica para esses dispositivos (BRODTKORB et al., 2012).

A Figura 2.1 mostra a linha do tempo e as diferentes APIs e linguagens criadas, a partir da *Open Graphics Library* (OpenGL) e DirectX, focadas na função gráfica do *hardware*, até as linguagens mais genéricas, como a CUDA (NVIDIA, 2017a) e *Open Computing Language* (OpenCL) (KHRONOS, 2017).

Com o aumento das possibilidades para o desenvolvimento de aplicações de alto desempenho nessas plataformas, dispositivos aceleradores começaram a ser utilizados por uma grande parte dos supercomputadores presentes na lista dos computadores mais poderosos do mundo, a top500.org (TOP500, 2019). Os supercomputadores *Summit* e *Sierra* ocupam respectivamente o primeiro e segundo lugar no *ranking*, sendo que ambos foram construídos pela IBM e tiram seu poder computacional de nove CPUs e NVIDIA V100 *Unidades de Processamento Gráficos* (GPUss).

Em terceiro lugar está o supercomputador Chinês *Sunway TaihuLight*, que utiliza sua própria implementação do *OpenACC 2.0* (diretivas de compilação para aceleradores) para auxiliar na paralelização do código e explorar a arquitetura utilizada (OPENACC,

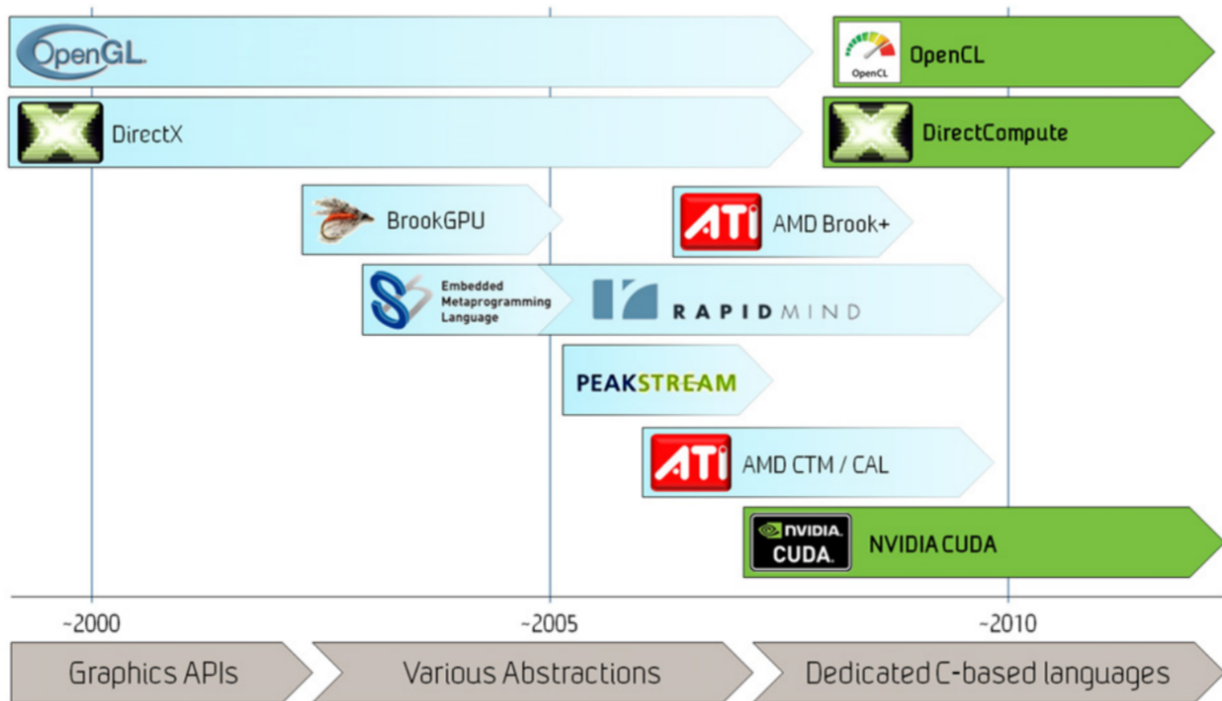

Figura 2.1. Evolução das APIs disponíveis para GPUs. Fonte: (BRODTKORB et al., 2012)

2016). Esses dados sugerem que o uso de arquiteturas híbridas exploradas por aplicações capazes de utilizar o paralelismo da máquina de forma eficiente são a base para pesquisas em Computação Paralela (GONçALVES, 2014).

Já no âmbito comercial, existem três grandes fabricantes de GPUs. Eles são a Intel (dominante apenas no mercado de dispositivos integrados de baixa performance), AMD e NVIDIA, desenvolvedores de dispositivos de alto rendimento. A NVIDIA domina tanto no aspecto comercial quanto no acadêmico (BRODTKORB et al., 2012). De acordo com os resultados fiscais do terceiro trimestre do ano fiscal de 2017 publicados pela NVIDIA, a receita total mostra um crescimento de 40% em comparação com o segundo trimestre. Neste trimestre também foi marcado a receita recorde de $2.00 bilhões de dólares, cerca de 54% a mais do que o ano anterior, sendo este crescimento liderado pela nova linha de GPUs, utilizada em consoles e Realidade Virtual (NVIDIA, 2016a).

Esses fabricantes, por meio dos *kits* de desenvolvimento e APIs que disponibilizam para os desenvolvedores, definem o modelo de programação e como essas GPUs funcionam dentro de um sistema heterogêneo, especificando como é feita a comunicação entre CPU-GPU e como a GPU utiliza a sua memória dedicada.

No esquema clássico de comunicação entre CPU e GPU, o *host* é responsável pela execução do programa principal e inicializa as *threads* na GPU, fazendo a execução de *kernels* (funções especiais extraídas de um programa principal e enviadas para que sejam executadas em um acelerador, criando um altíssimo grau de paralelismo entre esses dois dispositivos) (GONçALVES, 2014). Essa comunicação é feita a partir de um barramento *PCI Express*,

como mostra a Figura 2.2.

A organização dos *cores* das GPUs varia conforme a arquitetura utilizada na construção desses dispositivos. Como as GPUs da NVIDIA são o foco do nosso estudo e é necessário entender a organização desses dispositivos para que o modelo de programação seja compreendido, algumas arquiteturas lançadas pela da NVIDIA são apresentadas em maior detalhe. A arquitetura Fermi, introduzida comercialmente na série *GTX 400*, A Kepler, introduzida comercialmente na série *GTX 600*, a arquitetura Maxwell, da série *GTX 750* e por fim a arquitetura Pascal, utilizada primariamente na *GTX 10 Series*.

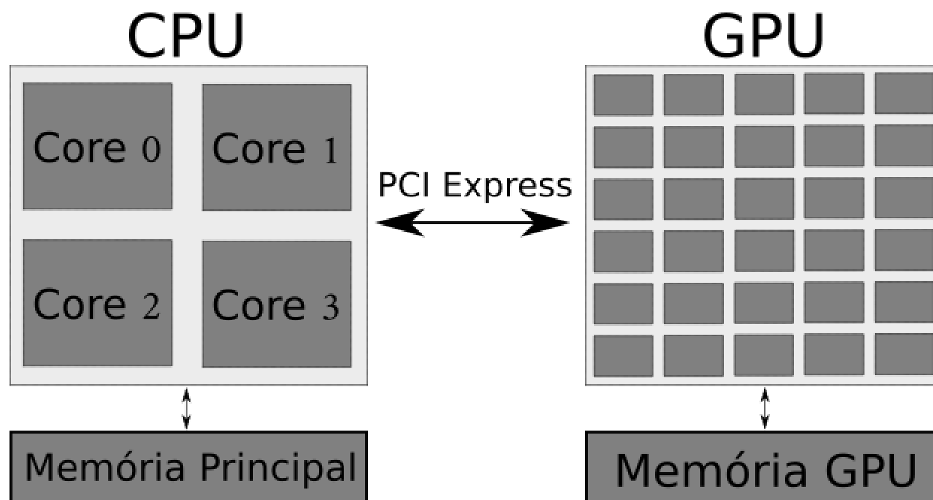

Figura 2.2. Comunicação CPU-GPU. Baseado em (GONçALVES, 2014)

Arquitetura Fermi

Os dispositivos da arquitetura Fermi e GPUs em geral seguem um modelo de execução diferente das CPUs, já que são compostas por centenas de núcleos de processamento mais simples se comparados com uma CPU *multicore*, onde os transistores são utilizados primariamente para computação e não para memórias *cache* e unidades de controle. (BRODTKORB et al., 2012; GONçALVES, 2014).

Os *cores* na arquitetura Fermi são organizados em 16 *Streaming Multiprocessors* (SMs). Cada um desses SMs possuem 32 núcleos de processamento, somando em um total de 512 núcleos. Além disso, a arquitetura Fermi possui dois níveis de memória *cache*. O primeiro nível (L1) é a memória compartilhada pelos 32 núcleos dentro do SM. O segundo nível (L2) é o mais alto, sendo compartilhada por todos os 16 SMs.

Existe ainda uma memória compartilhada, que divide espaço fisicamente com a memória *cache* L1, podendo ser explicitamente configurada e variando de tamanho entre 16KB e 48KB (somando no total 64KB). Esta memória pode ser utilizada para acelerar a execução de um programa. Por fim, essa arquitetura ainda pode possuir uma memória global DRAM de até 6GB (GONçALVES, 2014).

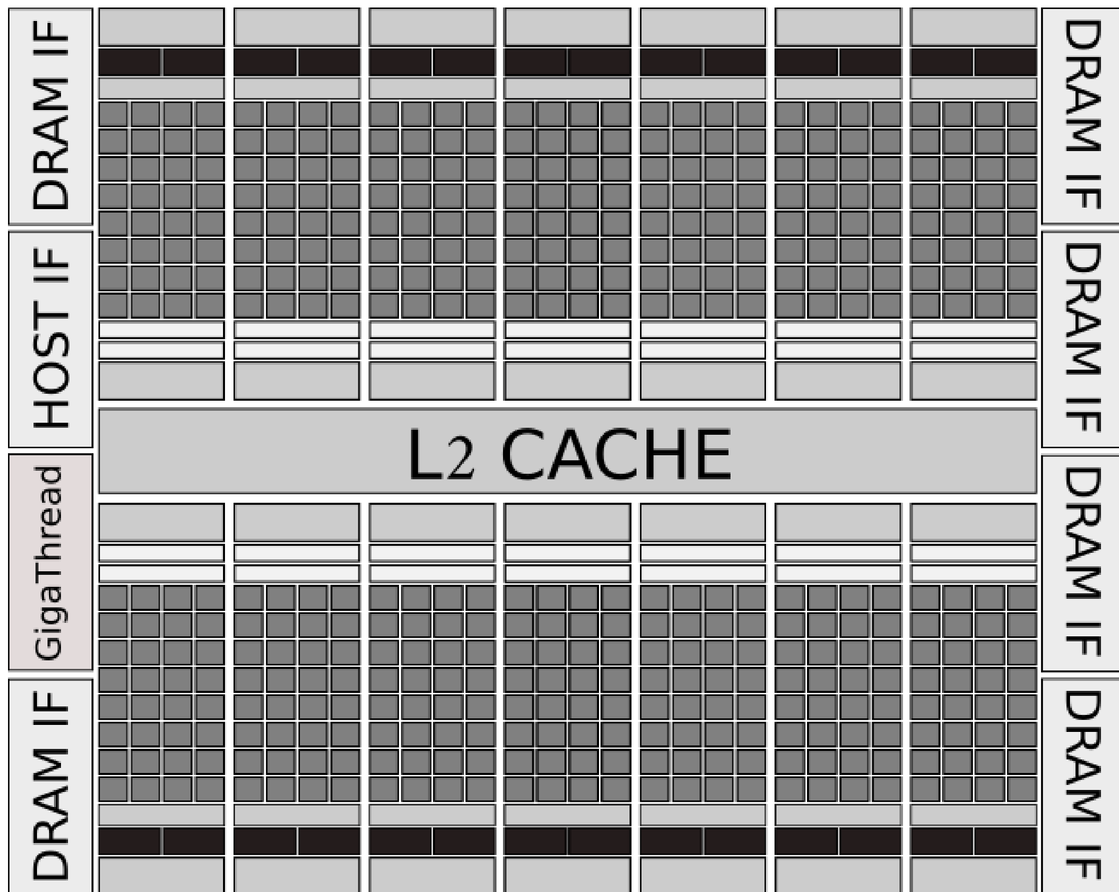

Figura 2.3. Arquitetura `Fermi`. Baseado em (BRODTKORB et al., 2012)

A Figura 2.3 mostra uma versão simplificada dos componentes dentro da `GPU` (SM, núcleos, memória *cache* e etc.). Já a Figura 2.4 mostra os detalhes dentro de um SM. Cada SM possuiu dois *warps* (unidade de escalonamento de *threads*), duas *Dispatch Units*, quatro *Special Function Units* (SFUs), responsável pela execução de instruções como seno, coseno, raiz quadrada e interpolação (BRODTKORB et al., 2012). Os 32 núcleos são organizados em dois grupos de 16 *cores* capazes de executarem instruções simples inteiras ou de ponto flutuante e 16 unidade de *load/store*.

Arquitetura `Kepler`

A arquitetura `Kepler` (NVIDIA, 2012), introduziu uma série de novas tecnologias utilizadas para simplificar o *design* de programas que exploram o paralelismo e para auxiliar no uso de `GPUs` tanto em supercomputadores quanto em computadores pessoais.

Entre às inovações dessa arquitetura em comparação com a `Fermi`, temos a introdução do paralelismo dinâmico, o que torna a `GPU` capaz de gerar o seu próprio trabalho. Isso significa que a `GPU` possui maior autonomia em relação a `CPU`, dado que o controle do fluxo do trabalho (disparar a execução de novas *threads*, sincronizar os resultados, escalonamento) pode ser feito no lado da `GPU`, agora capaz de chamar *kernels* dentro de outros *kernels*,

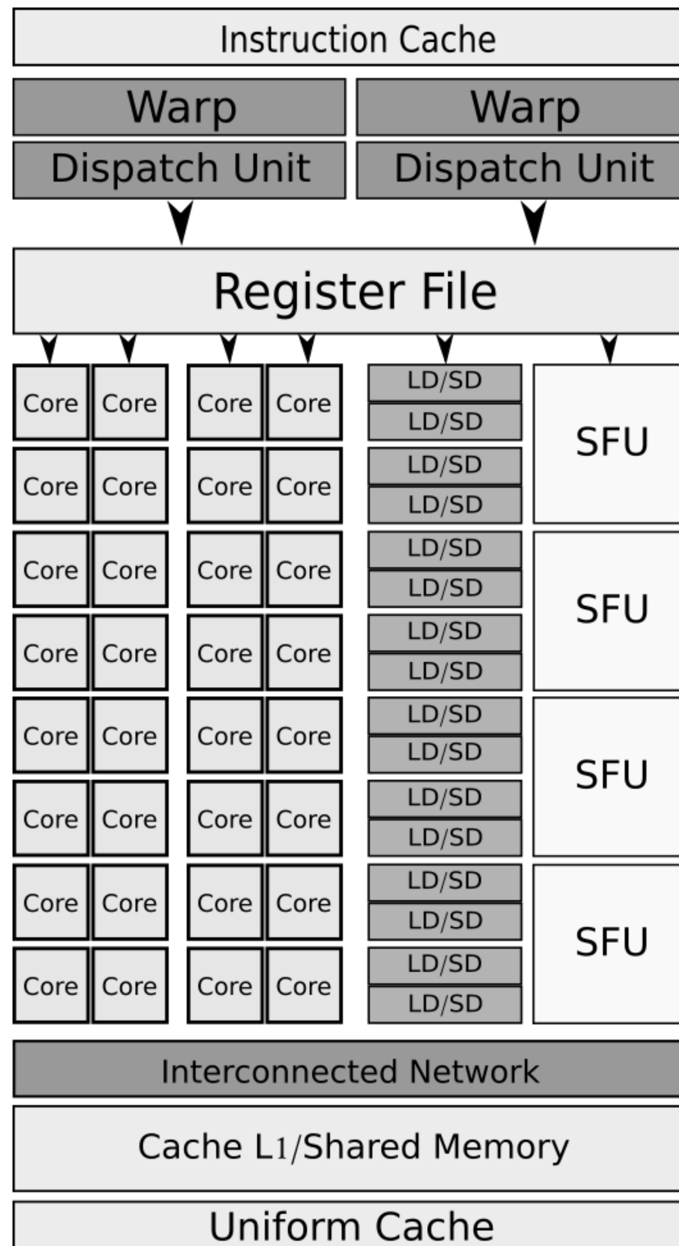

Figura 2.4. *Streaming Processor* da Arquitetura `Fermi`. Baseado em (BRODTKORB et al., 2012)

permitindo a execução de programas diretamente da GPU (GONçALVES, 2014; NVIDIA, 2012), sem a necessidade de retornar o controle de fluxo de execução para a `CPU`.

Como exemplificado na Figura 2.5, a `GPU Kepler` permite que a `CPU` fique livre para executar outras tarefas, onde a `Fermi` é completamente dependente da `CPU` para executar *kernels* e deve retornar o controle de execução para a `CPU` após cada chamada.

Outra inovação da `Kepler` é a tecnologia *Hyper-Q* que possibilita diversos núcleos do processador enviarem trabalho (fazer a chamada de *kernels*) simultaneamente para a `GPU`. Essa arquitetura permite que 32 chamadas simultâneas sejam feitas, enquanto a arquitetura `Fermi` permitia apenas uma chamada (NVIDIA, 2012; GONçALVES, 2014).

Quanto a organização arquitetural, cada *Streaming Multiprocessor Kepler* (SMX) da

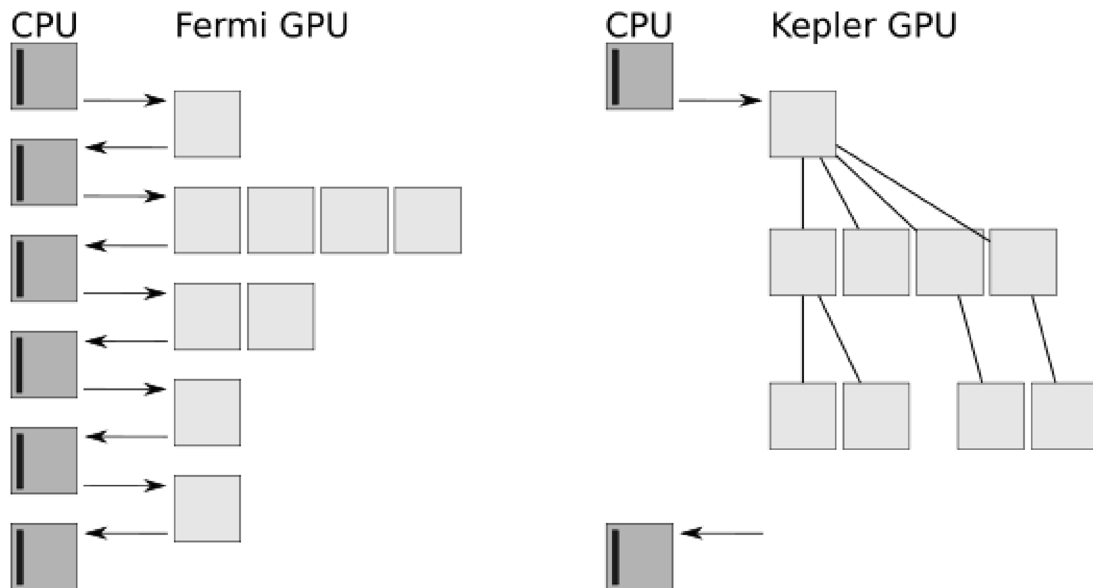

Figura 2.5. Paralelismo Dinâmico. Fonte: (NVIDIA, 2012)

`GPU Kepler` é dividido em 192 *cores* de precisão simples, 64 unidades de precisão dupla, 32 SFUs e mais 32 unidades *load/store* como visto na Figura 2.6. No total, a `Kepler GK110` pode possuir um conjunto de até 15 SMXs, como mostrado na Figura 2.7, que dá uma visão geral do *chip* completo (NVIDIA, 2012; GONçALVES, 2014).

Na Figura 2.6 também é possível visualizar que a arquitetura `Kepler` possui quatro *warps*, enquanto a `Fermi` possui apenas dois. Essa organização permite que 4 *warps* de 32 *threads* sejam escalonadas e executadas concorrentemente nos SMXs, ao contrario dos dois escalonados na `Fermi`.

Arquitetura `Maxwell`

`GPUs` da Arquitetura `Maxwell` introduziram um novo modelo que melhorou o particionamento lógico da arquitetura, balanceamento da carga de trabalho e nas taxas de energia gasta pela placa (NVIDIA, 2014). O escalonamento de tarefas foi modificado de forma que *stalls* fossem evitados, diminuindo a quantidade de energia gasta na execução de uma instrução (GONçALVES, 2014).

A `GPU GM107` apresenta cinco SMMs organizados dentro de um *Graphic Processing Cluster* (GPC). A `Maxwell GM107` possui um GPC, com dois controladores de memória de 64-bits cada, como exemplificado na Figura 2.8, sendo essa a configuração utilizada nas placas *GTX750 Ti* (NVIDIA, 2014).

Apesar dos SMMs serem menos poderosos que os SMXs da arquitetura `Kepler` (cerca de 90% do rendimento), a organização dessa placa permite que muito mais núcleos (cerca de 1,7 vezes a quantidade de núcleos da `Kepler`) sejam utilizados devido ao tamanho de cada *core* (GONçALVES, 2014). Os detalhes desta arquitetura são destacados na Figura 2.9.

Figura 2.6. Arquitetura SMX. Baseado em (NVIDIA, 2012)

Arquitetura `Pascal`

A arquitetura `Pascal` (NVIDIA, 2016c, 2016b) é a arquitetura considerada a sucessora da arquitetura `Maxwell`, lançado pela NVIDIA em 2016. Este modelo foi introduzido no `Tesla P100` (NVIDIA, 2016c) da NVIDIA, utilizando a `GPU Pascal GP100`, sendo que uma versão modificada, a `GP104` (NVIDIA, 2016b), é utilizada nos dispositivos aceleradores `GTX 10 Series`.

Essa arquitetura é composta por 7.2 bilhões de transistores, incluindo 2560 núcleos

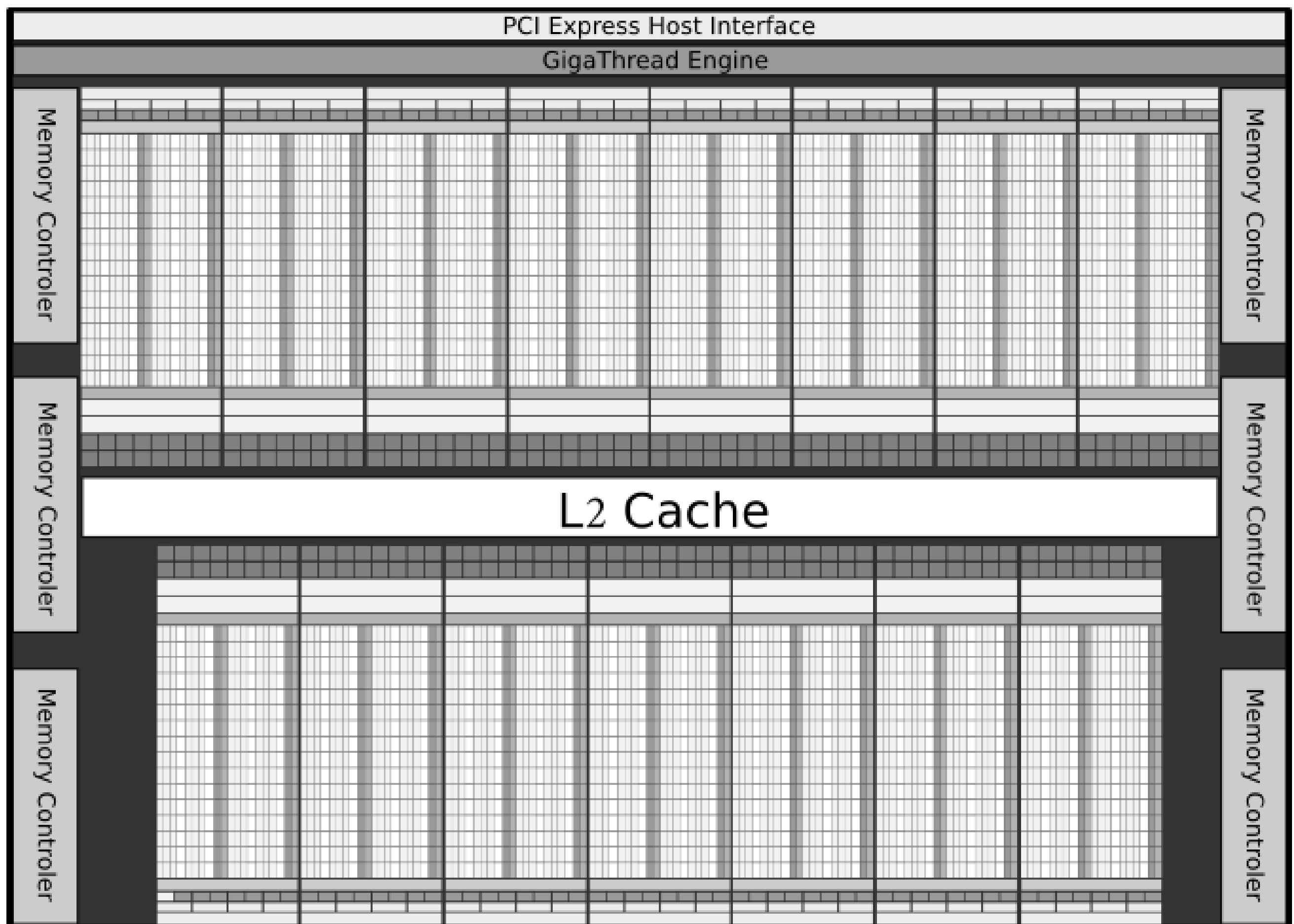

Figura 2.7. *Chip* completo `Kepler`. Baseado em (NVIDIA, 2012)

de precisão simples. Esses núcleos estão organizados em 4 GPCs. Cada GPC possui 5 SMs, que por sua vez contêm 128 núcleos, 256 *kb* de registro e 96 *kb* de memória compartilhada.

Essa arquitetura apresenta inovações quanto a memória (`GDDR5X`), capaz de transferências de até 10 *Gpbs* e ao motor de projeção, que agora é capaz de gerar múltiplas projeções de uma única *stream*, sendo que gráficos podem ser gerados em um campo de visão muito maior mas mantendo a perspectiva correta.

2.3. Modelo de Programação - `CPU` e `GPU`

O modelo de programação para uma `CPU` tradicional difere do modelo de programação para `GPUs` e arquiteturas heterogêneas em geral, uma vez que o *hardware* é extremamente diferente (BRODTKORB et al., 2010, 2012). Devido a essas diferenças, essas plataformas exigem que os programadores aprendam um novo paradigma com novos conceitos, bibliotecas e até mesmo uma nova linguagem de programação completa. Além disso, o desenvolvimento para arquiteturas heterogêneas deve ser feito de maneira mais cuidadosa, já que o *hardware* é menos robusto para aplicações menos otimizadas (GONçALVES, 2014; BRODTKORB et al., 2010).

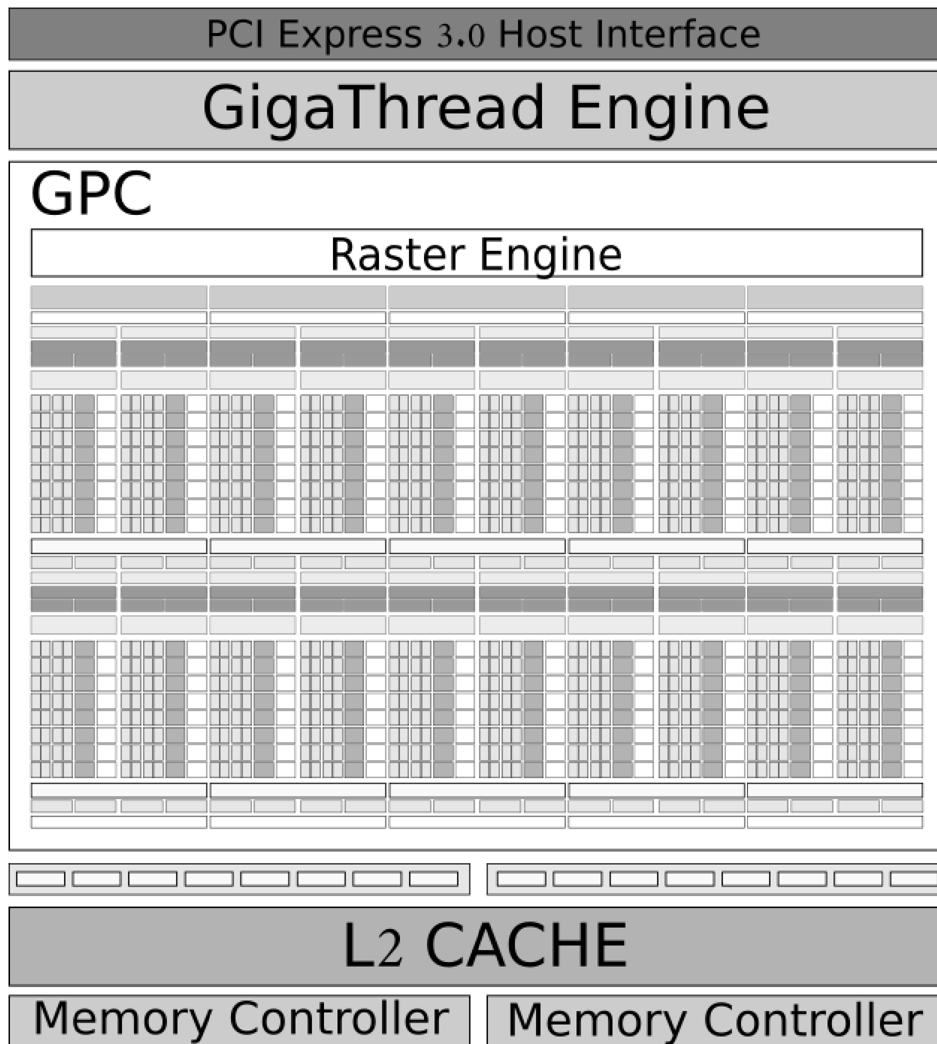

Figura 2.8. `Maxwell GM107`. Baseado em (NVIDIA, 2014)

O modelo de programação e execução utilizado pelas placas da NVIDIA é suportado pela plataforma `CUDA`, sendo portanto o modelo que exploraremos neste trabalho.

2.3.1. `CUDA`: *Threads* na `GPU`

`CUDA`, também conhecida como Arquitetura de Computação Paralela de Propósito Geral, é uma plataforma/tecnologia desenvolvida pela NVIDIA que oferece uma série de instruções, bibliotecas e ferramentas de compilação permitindo que aplicações paralelas sejam desenvolvidas e executadas em `GPUs` (STRINGHINI et al., 2012; BRODTKORB et al., 2010).

Além de oferecer uma extensão para as linguagens C/C++, `CUDA` oferece suporte para linguagens como FORTRAN, `OpenCL`, DirectCompute e para abordagens baseadas em diretivas como o padrão OpenACC (OPENACC, 2017).

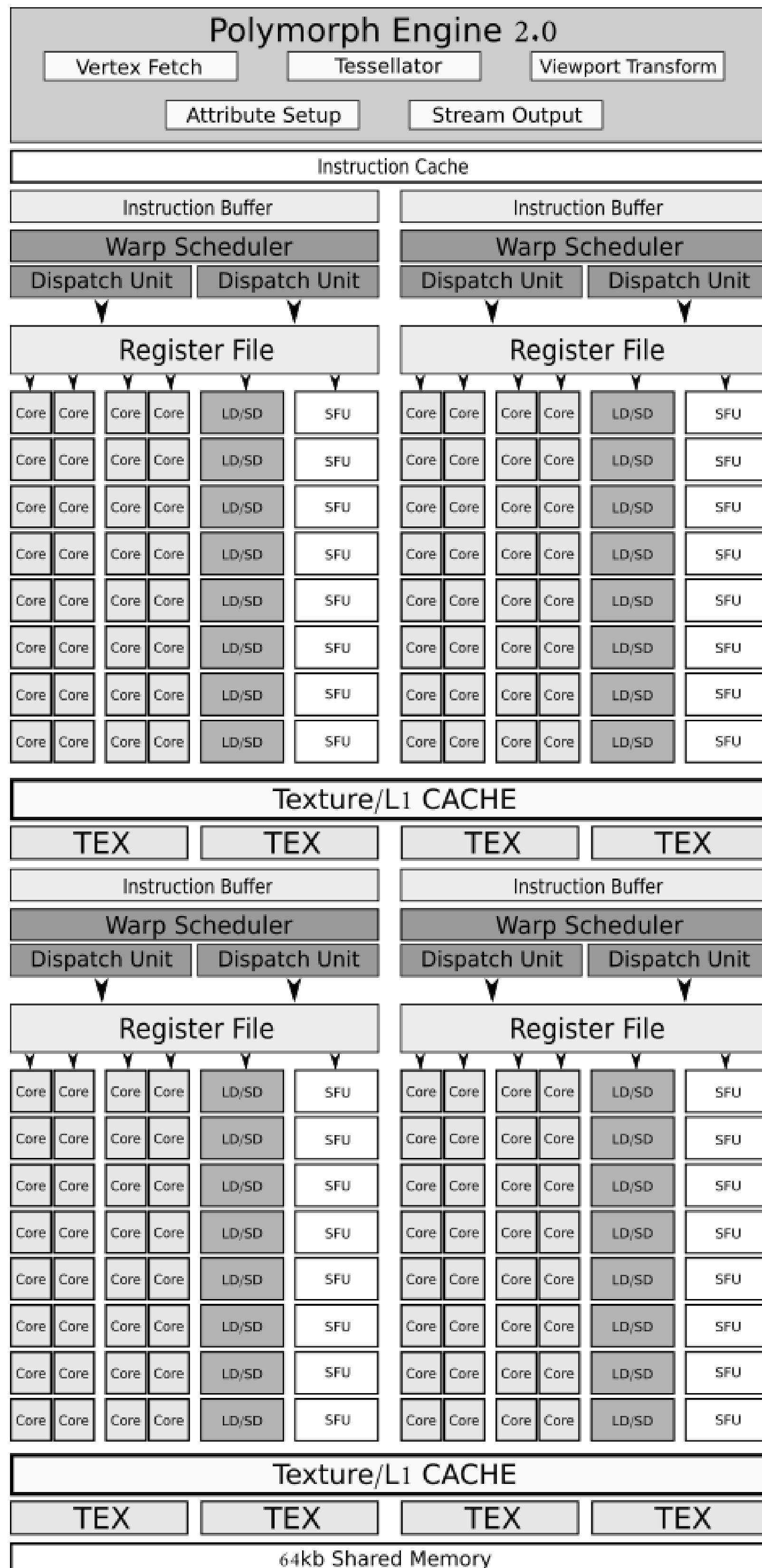

Figura 2.9. `Maxwell` SMM. Baseado em (NVIDIA, 2014)

Modelo de Programação

O modelo de programação `CUDA` assume que a `GPU` trabalha em conjunto com a `CPU`. A `CPU` controla o fluxo de execução do programa, fazendo chamadas de funções (*kernels*) para serem

executadas pelas *threads* do dispositivo acelerador (NVIDIA, 2017b; STRINGHINI et al., 2012; GONçALVES, 2014). O fluxo de comunicação entre CPU e GPU é exemplificado na Figura 2.10.

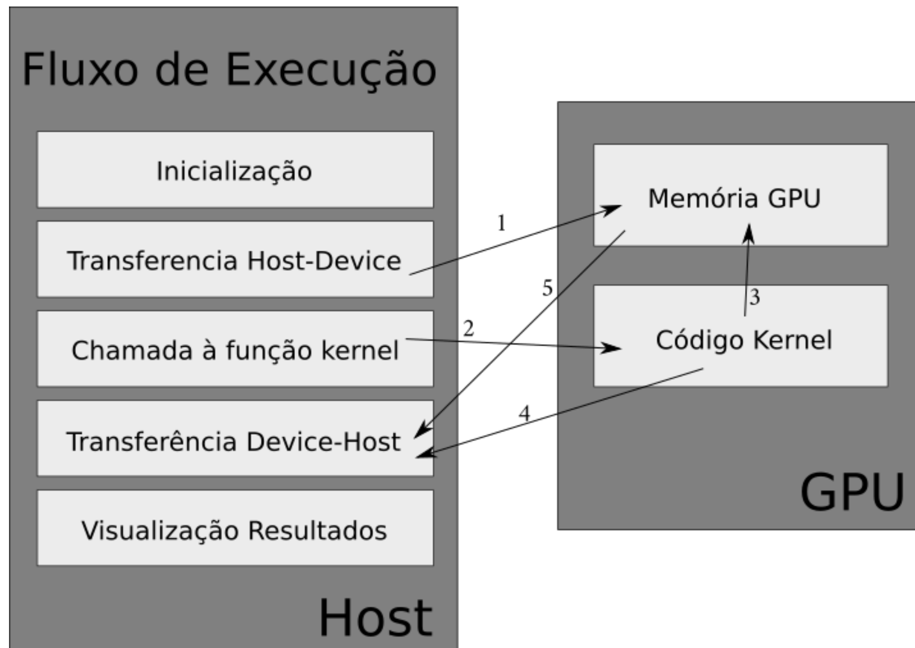

Figura 2.10. Modelo Clássico do Fluxo de Execução de um *kernel*. Baseado em (GONçALVES, 2014)

Uma função que será executada na GPU deve ser explicitamente definida como um *kernel*. Para isso, utiliza-se o modificador ___*global*___ na definição da função (NVIDIA, 2017b). As *threads* da GPU por sua vez executam o mesmo conjunto de instruções definidos dentro do *kernel* (SIMT).

Na invocação do *kernel*, a quantidade de *threads* devem ser explicitamente definidas pelo programador por meio da notação <<< *Dg, Db, Ns, S* >>> entre o nome da função e a lista de parâmetros (NVIDIA, 2017b), onde os parâmetros para a plataforma CUDA são:

- ***Dg*** é do tipo *dim3* (um vetor de inteiros com 3 dimensões: x, y e z), que especifica a dimensão e tamanho de um *grid*;
- ***Db*** também é do tipo *dim3*, que especifica a quantidade de *threads* por *block*;
- ***Ns*** é do tipo *size_t* que especifica a quantidade de *bytes* extras que serão alocados em memória compartilhada para cada *block*. Este parâmetro é opcional, assumindo o valor 0 caso não especificado;
- ***S*** é do tipo *cudaStream_t*, que especifica qual é a *stream* associada. Uma *stream* em CUDA é a sequencia de operações que serão executadas em ordem de invocação pelo *host*. Operações dentro de uma *stream* serão executadas em ordem, mas varias *streams* podem ser executadas concorrentemente. Esse valor também é opcional, assumindo o valor da *stream* padrão, caso omitido.

O conjunto de *threads*, *blocks* e *grids* criados nessa invocação são organizados de uma forma hierárquica. As *threads* são organizadas em um ou mais *blocks*, que por sua vez são organizados em um *grid* (NVIDIA, 2017b). Cada um desses níveis hierárquicos possuem vetores tridimensionais com valores embutidos que identificam a posição dos *blocks* e *threads* dentro de seus níveis. Esse valores podem ser utilizados durante a execução do *kernel* para manipular o arranjo de *threads* criadas. Essas variáveis são:

- ***gridDim***: Indica a quantidade de *blocks* dentro de um eixo (x,y ou z) do *grid*;
- ***blockDim***: Indica a quantidade de *threads* dentro de um eixo do *block*;
- ***blockIdx***: Mostra a posição do *block* nos eixos do *grid*;
- ***threadIdx***: Posição do *thread* nos eixos do *block*.

O Código 2.1 mostra um exemplo básico de definição e invocação de um *kernel* utilizando um *block* com 8 *threads* divido em 8 *blocks* dentro do *grid*, totalizando 64 *threads*. Dentro do *kernel*, utiliza-se as variáveis embutidas para acessar os valores dentro de um vetor a partir da identificação da *thread*.

No Código 2.1 podemos observar também o modelo de memória utilizado pela GPU, visto que todos os dados que serão utilizados na execução do *kernel* devem ser copiados na memória da GPU previamente e copiados de volta para o *host* após o fim da execução. Essas operações são feitas a partir das funções *cudaFree*, *CudaMalloc*, e *cudaMemcpy*.

```
#include <stdio.h>
#include <stdlib.h>
// CUDA Kernel
__global__ void vecAdd(double *a, double *b, double v, int n)
{

    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < n)
        b[id] = a[id] + v;
}

int main( int argc, char* argv[] )
{

    int N = 64;
    // Vetores do Host
    double *h_a;
    double *h_b;

    // Vetores da GPU
    double *d_a;
    double *d_b;

    size_t size = N*sizeof(double);

    // Alocacao de memoria no Host
    h_a = (double*)malloc(size);
    h_b = (double*)malloc(size);
```

```
    // Alocacao de memoria na GPU
    cudaMalloc(&d_a, size);
    cudaMalloc(&d_b, size);

    int i;
    // Inicializacao do Vetor
    for( i = 0; i < N; i++ )
        h_a[i] = i;

    // Copiando a Memoria do Host para a GPU
    cudaMemcpy( d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, size, cudaMemcpyHostToDevice);

    // Execute the kernel
    vecAdd<<<8, 8>>>(d_a, d_b, 10, N);
    cudaDeviceSynchronize();

    // Copia a memoria de volta pro Host
    cudaMemcpy( h_b, d_b, size, cudaMemcpyDeviceToHost );

    // Free Host
    cudaFree(d_a);
    cudaFree(d_b);

    //Free GPU
    free(h_a);
    free(h_b);

    return 0;
}
```

Código-fonte 2.1. Chamada de *grid* 8x8. Fonte (STRINGHINI et al., 2012)

A Figura 2.11 mostra a organização hierárquica das *threads*. O valor da variável *id* na linha *4* é calculada a partir das variáveis embutidas, que linearizam o endereço das *threads*, conforme apresentado na Figura 2.12.

Modelo de Execução

Quando um *kernel* é invocado, o *grid* é enviado para o dispositivo que deve processá-lo. Os *blocks* são numerado e distribuídos para os SMs. Uma vez dentro do SM, grupos de até 32 *threads* de um *block* são organizadas em *warps*. Cada um desses *warps* são escalonados por *warp scheduler*, que mapeia as *threads* para os núcleos que farão a execução (NVIDIA, 2017b).

2.3.2. `OpenMP` : Paralelismo em `CPUs` *multicore*

O paralelismo na `CPU` já é algo que vem sendo explorado à algum tempo. Desde o padrão POSIX, que oferece um modelo de execução para a manipulação de *threads*, desenvolvedores já exploram as melhorias oferecidas pelo uso de *multi-threads*. A interface *pthreads* oferece uma

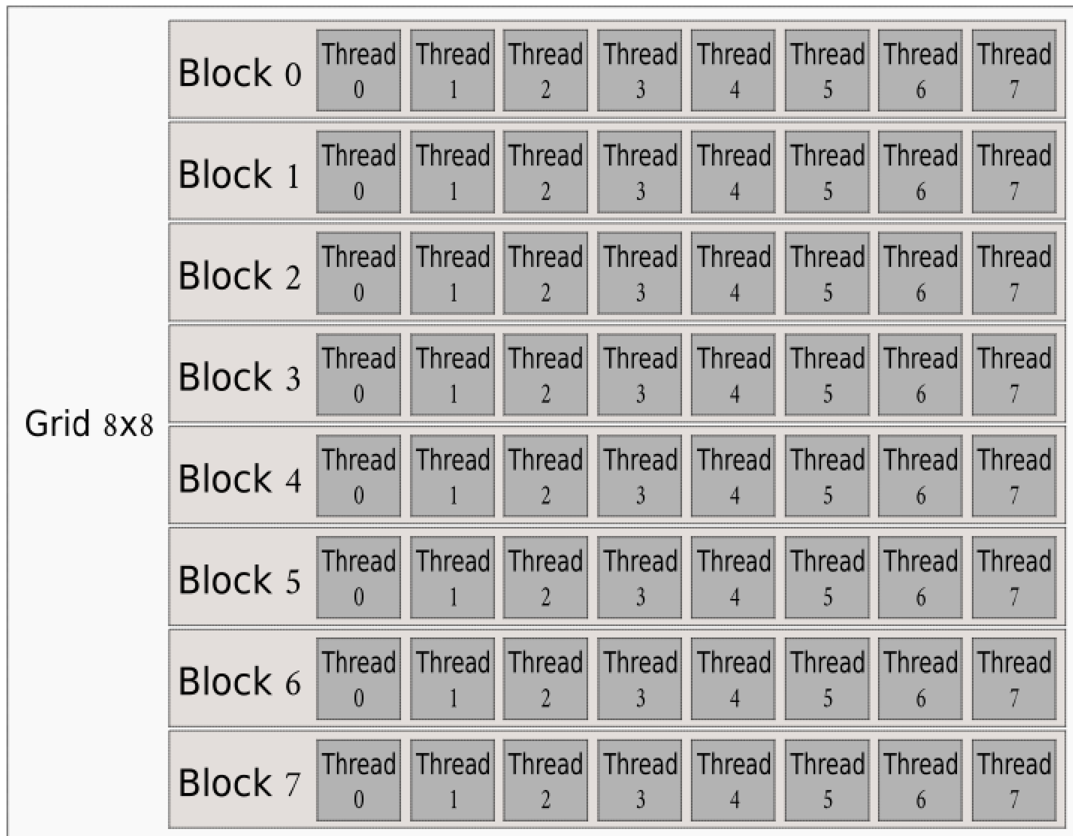

Figura 2.11. *Grid* com 64 *threads*. Baseado em (STRINGHINI et al., 2012)

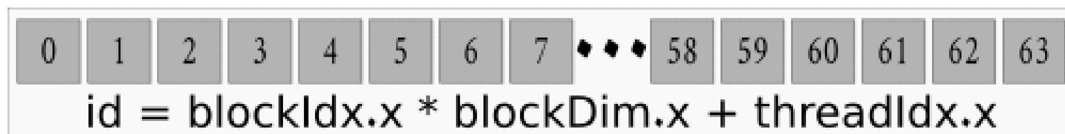

Figura 2.12. Linearização para o endereço do vetor.

implementação deste modelo, permitindo criação e sincronização de *threads* (STRINGHINI et al., 2012).

O padrão `OpenMP` também oferece um modelo que possibilita a programação paralela, disponibilizando um conjunto de diretivas de compilação, funções e variáveis de ambiente que podem ser utilizadas nas linguagens C/C++ e Fortran. Existem diversas implementações deste modelo, portanto é comum que os compiladores mais populares possuam opções de compilação que gerem código `OpenMP` (STRINGHINI et al., 2012). Entre esses compiladores, temos o `GCC` com a `libgomp` (LIBGOMP, 2016), o `Intel ICC` com a `libomp` (INTEL, 2016) e o *`Low Level Virtual Machine`* `(LLVM) clang` que suporta as duas bibliotecas.

Modelo de execução

No modelo de execução do `OpenMP`, as *threads* são criadas a partir de blocos de código definidos por uma anotação, conhecida como regiões paralelas. O conjunto de instruções dentro deste bloco é extraído em uma *outlined function* (STRINGHINI et al., 2012).

O programa resultante é inicialmente executado sequencialmente até que a região paralela é encontrada. Quando isso acontece, uma operação similar ao *fork* é feita e o conjunto de *threads* é criado para executar o código da *outline function* (STRINGHINI et al., 2012).

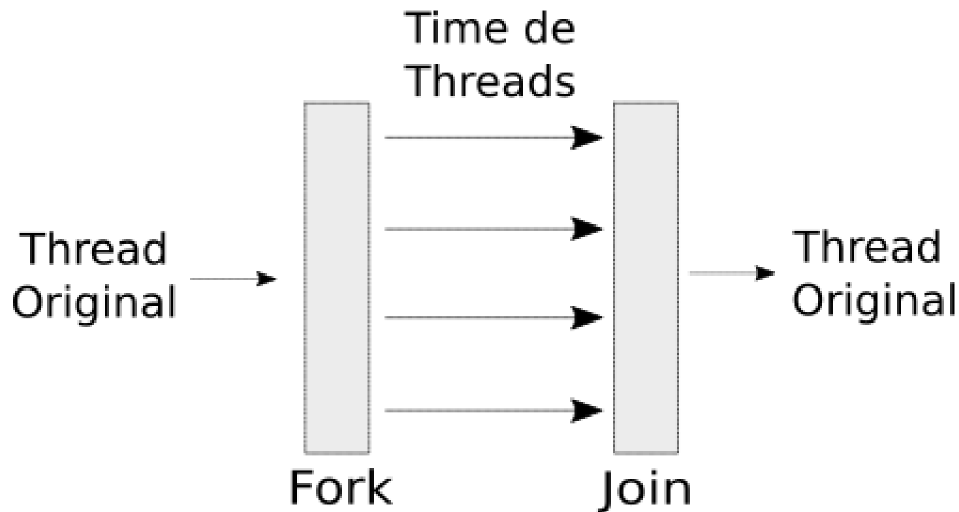

Figura 2.13. Fluxo de execução de uma região paralela.

Quando a execução dessa região termina, as *threads* chegam em uma barreira implícita, onde as *threads* são finalizadas e o programa é sincronizado, voltando a ser sequencial (STRINGHINI et al., 2012). A Figura 2.13 mostra o fluxo de execução de um programa anotado.

```
#include <omp.h>
#include <stdio.h>
#define SIZE 16
int main() {
    int A[SIZE], i;
    //inicia as threads que executarao o bloco "for"
    #pragma omp parallel for schedule(static, 2) num_threads(4)
    for(i = 0; i < SIZE; i++){
        A[i] = i * i;
        printf("Th%d[%2d] = %d\n", omp_get_thread_num(),i, A[i]);
    }
}
```

Código-fonte 2.2. Exemplo de diretivas `OpenMP`. Fonte (STRINGHINI et al., 2012)

O Código 2.2 mostra um exemplo das diretivas e funções disponibilizadas pela biblioteca *omp* para as linguagens C e C++. Nela, podemos ver que a diretiva *pragma omp parallel* cria uma região paralela e o time com as *threads* que serão utilizadas. O construtor *for* define que a região paralela executará um laço de repetição. A cláusula *num_threads* define o tamanho do time de *threads* que serão criados para executar o laço de acordo com o *schedule*.

O *schedule* apresenta dois parâmetros, o primeiro sendo o tipo (estático, dinâmico, auto e etc.) e o segundo o *chunk_size*, ou seja, a quantidade de iterações que uma *thread* irá

executar por vez (neste caso, cada *thread* irá executar *SIZE/4* iterações do laço).

Estático	**Dinâmico**
Th0[0] = 0	Th1[0] = 0
Th0[1] = 1	Th0[2] = 4
Th0[8] = 64	Th2[4] = 16
Th0[9] = 81	Th3[6] = 36
Th1[2] = 4	Th1[1] = 1
Th1[3] = 9	Th0[3] = 9
Th1[10] = 100	Th2[5] = 25
Th1[11] = 121	Th3[7] = 49
Th2[4] = 16	Th1[8] = 64
Th2[5] = 25	Th0[10] = 100
Th2[12] = 144	Th2[12] = 144
Th2[13] = 169	Th3[14] = 196
Th3[6] = 36	Th1[9] = 81
Th3[7] = 49	Th0[11] = 121
Th3[14] = 196	Th2[13] = 169
Th3[15] = 225	Th3[15] = 225

Tabela 2.1. Diferença entre *Schedule* estático e *schedule* dinâmico. Fonte: (STRINGHINI et al., 2012)

A Tabela 2.1 mostra como as *threads* se comportam dadas as diretivas do Código 2.2. Na primeira coluna, vemos que cada *thread* é executada 4 vezes e os índices estão separados em *chunks* (subconjuntos de iterações) de tamanho 2, como definido em *schedule*, funcionando num modelo *round-robin*.

Na segunda coluna, vemos um *schedule* dinâmico, os pacotes são divididos da mesma forma, porém a *thread* que irá processá-los não seguem a mesma ordem, já que a *thread* que estiver livre para executar pega o próximo *chunk* disponível.

2.4. Ferramentas Relacionadas

Nesta seção são apresentadas as ferramentas e projetos relacionados ao desenvolvimento do trabalho. Estas ferramentas são capazes de aplicar técnicas de paralelização automática em um código sequencial sem envolvimento direto do programador.

Entre as ferramentas apresentadas temos o o `PoLLy` (GROSSER et al., 2012) projeto do `LLVM` (LLVM, 2017; LATTNER; ADVE, 2004) e as ferramentas ***Polyhedral Parallel Code Generation*** `(PPCG)` (VERDOOLAEGE et al., 2013) e `KernelGen` (MIKUSHIN et al., 2013).

O processo de paralelização é feito com base no *Modelo Polyhedral*, que apresenta uma forma alternativa de representar certas regiões do código (como laços de repetição), conhecidas como SCoP, que podem ser paralelizadas, dependendo das relações de dependência entre as instruções.

Para que um laço de repetição seja analisado durante o processo de detecção de paralelismo, a variável de indução deve possuir um valor com limite inferior e superior, sendo incrementado por uma constante. As chamadas de funções dentro do laço não podem utilizar a variável de indução ou um elemento de um vetor como parâmetro e as condicionais devem comparar dois valores de duas expressões afins (GONçALVES, 2014).

2.4.1. LLVM

O Projeto LLVM (LLVM, 2017; LATTNER; ADVE, 2004) é uma infraestrutura de compiladores que visa disponibilizar métodos que possibilitem a análise e transformação de código para qualquer tipo de *software* de forma transparente. O projeto fornece uma estrutura modular que pode ser utilizada para a criação de novas ferramentas de compilação, possuindo ferramentas relacionadas à todas as fases do fluxo de compilação (*front end*, *middle end* e *backend*) (GONçALVES, 2014).

O LLVM utiliza uma representação intermediária, conhecida como ***Low Level Virtual Machine Intermediate Representation*** (LLVM-IR), como base para fazer transformações e análise de código durante todas as fases do projeto. O LLVM-IR provê informação de alto nível necessária para análise mesmo sendo baixo nível o suficiente para representar qualquer programa (LATTNER; ADVE, 2004).

É importante conhecer essa representação dado que ela é utilizada no PoLLy, que é uma ferramenta relacionada ao trabalho aqui desenvolvido e com diversas outras ferramentas que se baseiam no *Modelo Polyhedral*.

O LLVM-IR apresenta três formatos equivalentes: o formato binário *bitcode*, o formato composto pelas estruturas de dados em memória e por fim o texto legível com extensão (*.ll*). O Código 2.3 mostra um código escrito em linguagem C e a sua representação intermediária LLVM-IR é mostrada no Código 2.4.

```
int funcA(int x, int y){
    int result = x * y;
    return result;
}

int main(){
    int a,b,c;
    a = 5;
    b = 2;
    c = funcA(a,b);

    return 0;
}
```

Código-fonte 2.3. Código *main* na linguagem C. Fonte (GONçALVES, 2014)

```
; ModuleID = 'main.c'
```

```
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @funcA(i32 %x, i32 %y) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %result = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  store i32 %y, i32* %2, align 4
  %3 = load i32* %1, align 4
  %4 = load i32* %2, align 4
  %5 = mul nsw i32 %3, %4
  store i32 %5, i32* %result, align 4
  %6 = load i32* %result, align 4
  ret i32 %6
}

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  store i32 0, i32* %1
  store i32 5, i32* %a, align 4
  store i32 2, i32* %b, align 4
  %2 = load i32* %a, align 4
  %3 = load i32* %b, align 4
  %4 = call i32 @funcA(i32 %2, i32 %3)
  store i32 %4, i32* %c, align 4
  %5 = load i32* %c, align 4
  ret i32 %5
}

attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="
    true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="
    false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="
    false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"Debian clang version 3.5.0-10 (tags/RELEASE_350/final) (based on
   LLVM 3.5.0)"}
```

Código-fonte 2.4. Código *main* na representação intermediária LLVM-IR. Fonte (GONçALVES, 2014)

Entre as ferramentas presentes dentro do LLVM que podem ser utilizadas modularmente, nós temos:

- opt: O comando opt é o otimizador modular do projeto LLVM. Este comando recebe como entrada um código no formato binário, executa as otimizações especificadas, e tem como saída o código otimizado.

As otimizações que podem ser feitas por meio deste comando dependem de quais bibliotecas estiverem listadas como utilizáveis (TEAM, 2019b).

- `llc`: O comando `llc` é capaz de fazer a compilação de um código no formato `LLVM-IR` para linguagem `assembly` de uma arquitetura alvo. Esse código, agora na linguagem `assembly` da máquina, pode ser passada por um montador nativo e ser utilizado para gerar um executável (TEAM, 2019a).

2.4.2. `PoLLy`

O `PoLLy`(GROSSER et al., 2012) é um projeto do `LLVM` utilizado para a otimização da representação intermediária (`LLVM-IR`) de programas. Essa ferramenta é capaz de detectar e extrair regiões relevantes para a paralelização de aplicações automaticamente, evitando que o desenvolvedor tenha que modificar o código (GROSSER et al., 2012).

Dado que o `PoLLy` trabalha com o `LLVM-IR`, a linguagem de programação utilizada não é importante, sendo que construções como os iteradores do `C++`, aritmética de ponteiros e laços de repetições baseados em *gotos* são todas traduzidas para um mesmo código base (GROSSER et al., 2012), o `LLVM-IR`.

A arquitetura do `PoLLy` é dividida em *front*, *middle* e *back end*, cada um sendo responsável por um passo na detecção e análise do `LLVM-IR`(GROSSER et al., 2012).

O *front end* é responsável pela detecção dos SCoPs de uma entrada em uma forma canônica, ou seja, seguem o formato apresentado na Seção 2.4. Na segunda parte (*middle end*), o código é analisado e otimizado e, por fim, o *back end* é responsável pela geração do código a partir da representação otimizada, transformando as regiões detectadas em código paralelo em `LLVM-IR`.

Um SCoP em `LLVM-IR` é um subgrafo que forma uma região com apenas uma entrada e apenas uma saída dentro do GFC. Sendo essa região paralelizável ela poderia ser transformada em um *kernel*, composto pelo corpo do laço, e o domínio de iteração seria transferido para os *ids* das *threads*.

Para exemplificar o funcionamento desta ferramenta, utilizaremos o código 2.5 na linguagem `C` que será transformado para que o SCoPs sejam detectados.

```c
#include <stdio.h>
#define N 1024

float h_a[N];
float h_b[N];
float h_c[N];

int main(){
    int i;

    init_array (); //Abreviado
```

```
    for(i = 0; i < N; i++){
        h_c[i] = h_a[i] + h_b[i];
    }

    print_array(); //Abreviado
    check_result(); //Abreviado

    return 0;
}
```

Código-fonte 2.5. Adição de Vetores em C. Fonte (GONçALVES, 2014)

O Código 2.6 apresenta o código `LLVM-IR` gerado a partir do Código 2.5. Para que a detecção da região paralela seja feita, deve-se transformar o código para a forma canônica. Isso é feito removendo dependências desnecessárias entre os blocos de código, criando blocos independentes (GROSSER et al., 2012).

```
; ModuleID = 'vectoradd.c'
target datalayout ="e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64
    :64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
@h_a = common global [1024 x float] zeroinitializer , align 16
@h_b = common global [1024 x float] zeroinitializer , align 16
@h_c = common global [1024 x float] zeroinitializer , align 16

define void @init_array() #0 {
}
define void @print_array() #0 {
}
define void @check_result() #0 {
}

define i32 @main() #0 {
  %1 = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 0, i32* %1
  call void @init_array()
  store i32 0, i32* %i, align 4
  br label %2

; <label>:2       ; preds = %18, %0
  %3 = load i32* %i, align 4
  %4 = icmp slt i32 %3, 1024
  br i1 %4, label %5, label %21

; <label>:5       ; preds = %2
  %6 = load i32* %i, align 4
  %7 = sext i32 %6 to i64
  %8 = getelementptr inbounds [1024 x float]* @h_a, i32 0, i64 %7
  %9 = load float* %8, align 4
  %10 = load i32* %i, align 4
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds [1024 x float]* @h_b, i32 0, i64 %11
  %13 = load float* %12, align 4
  %14 = fadd float %9, %13
  %15 = load i32* %i, align 4
```

```
  %16 = sext i32 %15 to i64
  %17 = getelementptr inbounds [1024 x float]* @h_c, i32 0, i64 %16
  store float %14, float* %17, align 4
  br label %18

; <label>:18      ; preds = %5
  %19 = load i32* %i, align 4
  %20 = add nsw i32 %19, 1
  store i32 %20, i32* %i, align 4
  br label %2

; <label>:21      ; preds = %2
  call void @print_array()
  call void @check_result()
  ret i32 0
}
```

Código-fonte 2.6. Adição de Vetores em LLVM. Fonte (GONçALVES, 2014)

O Código 2.7 mostra os blocos independentes. Podemos perceber que, antes o laço de repetição estava dentro de um único bloco básico. A partir desta representação, é possível detectar o paralelismo da aplicação.

```
; ModuleID = 'vectoradd.preopt.ll'

target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64
    :64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
@h_a = common global [1024 x float] zeroinitializer, align 16
@h_b = common global [1024 x float] zeroinitializer, align 16
@h_c = common global [1024 x float] zeroinitializer, align 16
define void @init_array() #0 {
}
define void @print_array() #0 {
}
define void @check_result() #0 {
}
define i32 @main() #0 {

call void @init_array()
br label %1

; <label>:1                                  ; preds = %6, %0
  %indvar = phi i64 [ %indvar.next, %6 ], [ 0, %0 ]
  %exitcond = icmp ne i64 %indvar, 1024
  br i1 %exitcond, label %2, label %7

; <label>:2                                  ; preds = %1
  %scevgep2.moved.to. = getelementptr [1024 x float]* @h_a, i64 0, i64 %indvar
  %scevgep1.moved.to. = getelementptr [1024 x float]* @h_b, i64 0,  i64 %indvar
  %scevgep.moved.to. = getelementptr [1024 x float]* @h_c, i64 0,  i64 %indvar
  %3 = load float* %scevgep2.moved.to., align 4
  %4 = load float* %scevgep1.moved.to., align 4
  %5 = fadd float %3, %4
  store float %5, float* %scevgep.moved.to., align 4
```

```
  br label %6

; <label>:6                                        ; preds = %2
  %indvar.next = add i64 %indvar, 1
  br label %1

; <label>:7                                        ; preds = %1
  call void @print_array()
  call void @check_result()
  ret
}
```

Código-fonte 2.7. Representação intermediaria de blocos independentes. Fonte: (GONçALVES, 2014)

A Figura 2.14 mostra o GFC da `LLVM-IR` com blocos independentes, onde o primeiro bloco dentro da área destacada representa o cabeçalho do laço, o segundo o corpo do laço e o terceiro é o bloco com o incremento do laço de repetição.

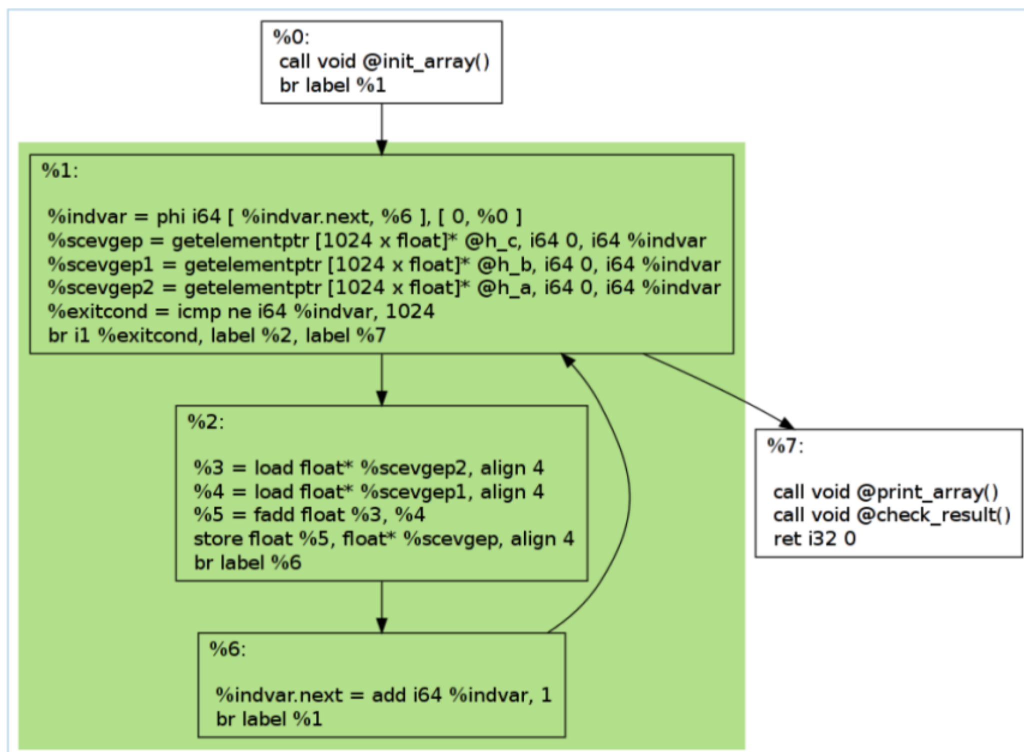

Figura 2.14. GFC com região paralelizável destacada. Fonte (GONçALVES, 2014).

O `PoLLy` é utilizado por diversas outras ferramentas, como o `KernelGen` (MIKUSHIN et al., 2013), que será apresentado neste trabalho e o `PoLLy-ACC` (GROSSER, 2016), que utiliza o `PoLLy` como um *front end* e o próprio `PPCG` (VERDOOLAEGE et al., 2013) para a geração de código(*back end*).

2.4.3. PPCG

O PPCG (VERDOOLAEGE et al., 2013) é um compilador *source-to-source* capaz de gerar código OpenMP, CUDA e OpenCL. Esse compilador utiliza o modelo *polyhedral* ou traduz regiões de código anotadas pelo programador, gerando código dividido em duas partes, uma para ser executada na CPU e outra na GPU (GONçALVES, 2014).

O primeiro passo na geração de código é a detecção de SCoPs, o que é feito utilizando a ferramenta pet (VERDOOLAEGE; GROSSER, 2012). O PPCG pode ser utilizado tanto na detecção automática quanto na análise de regiões de código delimitadas por anotações, como apresentado no Código 2.8.

```
#include <stdlib.h>

int main() {
  int i;
  int a[101];

  i = 0;
  #pragma scop
  for (i = 0; i < 100; ++i)
    a[i] = i;
  #pragma endscop

  if (a[100] != 100)
    return EXIT_FAILURE;

  return EXIT_SUCCESS;
}
```

Código-fonte 2.8. *iterator.c* anotado para o PPCG

O Código é gerado em duas partes, uma para ser executada no *host* e uma para ser executada no dispositivo acelerador. O comando para gerar o código CUDA é apresentado no Código 2.9.

```
ppcg --target=cuda iterator.c
```

Código-fonte 2.9. Comando para gerar código para GPU a partir da anotação.

As instruções dentro do *#pragma scop* são transformadas em *kernels* que são invocados no *host*. Esses *kernels* são mostrados no Código 2.10, onde *kernel0* executa as instruções dentro do laço de repetição, *kernel1* coloca o valor do limite do laço (100) na variável *i*, que vai ser copiada para o vetor *a* no *kernel2*.

```

#include "iteratorPragma_kernel.hu"
__global__ void kernel0(int *a)
{
  int b0 = blockIdx.x;
```

```
  int t0 = threadIdx.x;

  if (32 * b0 + t0 <= 99)
    a[32 * b0 + t0] = (32 * b0 + t0);
}
__global__ void kernel1(int *i)
{
  int private_i;

  {
    private_i = (100);
    *i = private_i;
  }
}
__global__ void kernel2(int *a, int *i)
{
  a[i[0]] = i[0];
}
```

Código-fonte 2.10. *Kernels* gerados pelo PPCG

O Código 2.11 mostra o código que vai ser chamado no *host*, controlando a transferência de memória e do fluxo de execução para a GPU.

```
#include <assert.h>
#include <stdio.h>
#include "iteratorPragma_kernel.hu"
#include <stdlib.h>

int main()
{
    int i;
    int a[101];

    i = 0;
    {
#define cudaCheckReturn(ret) \
  do { \
    cudaError_t cudaCheckReturn_e = (ret); \
    if (cudaCheckReturn_e != cudaSuccess) { \
      fprintf(stderr, "CUDA error: %s\n", cudaGetErrorString(cudaCheckReturn_e)); \
      fflush(stderr); \
    } \
    assert(cudaCheckReturn_e == cudaSuccess); \
  } while(0)
#define cudaCheckKernel() \
  do { \
    cudaCheckReturn(cudaGetLastError()); \
  } while(0)

      int *dev_a;
      int *dev_i;

      cudaCheckReturn(cudaMalloc((void **) &dev_a, (101) * sizeof(int)));
      cudaCheckReturn(cudaMalloc((void **) &dev_i, sizeof(int)));
```

```
    cudaCheckReturn(cudaMemcpy(dev_a, a, (101) * sizeof(int), cudaMemcpyHostToDevice));
    {
      dim3 k0_dimBlock(32);
      dim3 k0_dimGrid(4);
      kernel0 <<<k0_dimGrid, k0_dimBlock>>> (dev_a);
      cudaCheckKernel();
    }

    {
      dim3 k1_dimBlock;
      dim3 k1_dimGrid;
      kernel1 <<<k1_dimGrid, k1_dimBlock>>> (dev_i);
      cudaCheckKernel();
    }

    {
      dim3 k2_dimBlock;
      dim3 k2_dimGrid;
      kernel2 <<<k2_dimGrid, k2_dimBlock>>> (dev_a, dev_i);
      cudaCheckKernel();
    }

    cudaCheckReturn(cudaMemcpy(a, dev_a, (101) * sizeof(int), cudaMemcpyDeviceToHost));
    cudaCheckReturn(cudaFree(dev_a));
    cudaCheckReturn(cudaFree(dev_i));
  }
  if (a[100] != 100)
      return EXIT_FAILURE;

  return EXIT_SUCCESS;
}
```

Código-fonte 2.11. Código do *host* gerado pelo PPCG

Como citado anteriormente, a detecção de regiões paralelizáveis pode ser feita automaticamente. Para exemplificar essa funcionalidade e a geração de código com diretivas OpenMP, utilizaremos o Código 2.8, desconsiderando as anotações. O Código 2.12 apresenta o comando utilizado para a geração de código OpenMP.

```
ppcg --pet-autodetect --target=c --openmp iteratorAuto.c
```

Código-fonte 2.12. Comando para gerar código paralelo para a CPU a partir da detecção automática.

O Código 2.13 mostra o código OpenMP gerado a partir da detecção automática de uma região paralelizável. Nota-se que o *schedule* e *num_threads* não são definidos, sendo assumidos valores padrões para essas variáveis.

```
#include <stdlib.h>

int main()
{
    int i;
    int a[101];
```

```
    /* ppcg generated CPU code */

    {
      i = (100);
      #pragma omp parallel for
      for (int c0 = 0; c0 <= 99; c0 += 1)
        a[c0] = (c0);
      a[i] = i;
    }

    if (a[100] != 100)
        return EXIT_FAILURE;

    return EXIT_SUCCESS;
}
```

Código-fonte 2.13. Código C com diretivas OpenMP

2.4.4. KernelGen

O KernelGen (MIKUSHIN et al., 2013) é uma ferramenta que trabalha com a detecção automática de regiões paralelizáveis do código. Essa ferramenta tem como base o projeto LLVM (LATTNER; ADVE, 2004; LLVM, 2017) e o projeto PoLLy(GROSSER et al., 2012).

Ao contrário do PPCG, o KernelGen não usa anotações do código para criar kernels, sendo que a geração de código CUDA é necessariamente automática (MIKUSHIN et al., 2013).

Apesar deste detalhe, a maior diferença entre essa ferramenta e as demais apresentadas até então é o seu modelo de execução. No modelo tradicional, a CPU controla o fluxo de execução enviando *kernels* para a GPU. Porém, o KernelGen é centrado na GPU.

O *kernel* principal é lançado na GPU assim que a aplicação começa a ser executada, independente do paralelismo do código, ou seja, todas as regiões do código (seriais ou paralelas) são executadas na GPU (MIKUSHIN et al., 2013). Existem duas exceções neste caso: o lançamento de novos *kernels* na GPU e chamadas de sistema ou funções de entrada e saída.

Na chamada de novos *kernels*, são feitos *callbacks* para a CPU que retoma o controle da execução temporariamente e envia para o GPU o novo código para ser executado. As chamadas de funções do sistema funciona de forma similar (*callbacks*), dando controle da execução do programa até que a chamada seja completada.

Capítulo 3

Ambiente de Execução

A ferramenta de compilação desenvolvida nesse trabalho foi produzida com o propósito de trabalhar em conjunto com o **ambiente de execução** desenvolvido em Gonçalves (2016). Esse ambiente é responsável pela interceptação do código e pela análise das métricas coletadas durante a execução da aplicação. Além disso, o ambiente também é capaz de fazer o *offloading* do código `GPU` para a arquitetura adequada caso detecte a necessidade de fazê-lo.

Neste capítulo será discutido o funcionamento básico deste ambiente, incluindo suas bibliotecas relevantes para o desenvolvimento da ferramenta de compilação e suas responsabilidades neste contexto. Por fim também serão discutidas as métricas que precisam ser coletadas e como esse processo de coleta é executado.

3.1. Funcionamento do Ambiente

Como entrada para o ambiente de execução devemos considerar uma aplicação híbrida, ou seja, composta por blocos de código (no caso deste ambiente, funções na linguagem C) que podem ser executados tanto na `CPU` quanto na `GPU` sem que o resultado gerado por esses dispositivos sejam divergentes, independente de qual deles seja utilizado.

Em um primeiro momento, o código *multithread* destinado à `CPU` é executado, sendo que interceptações no código são feitas para contadores de desempenho de hardware possam ser coletados. A Figura 3.1 apresenta os diferentes cenários que podem ocorrer durante a execução de um programa.

Durante a execução, uma *thread* é escolhida para fazer as medições de desempenho e o restante das *threads* continuam a execução do programa. Caso a decisão seja pelo *offloading*, as *threads* da `CPU` param o funcionamento do programa e são terminadas pelo ambiente de execução. Desta forma, a *thread* principal do programa faz a transferência dos dados de memória e da responsabilidade da execução para a `GPU` (GONçALVES, 2016).

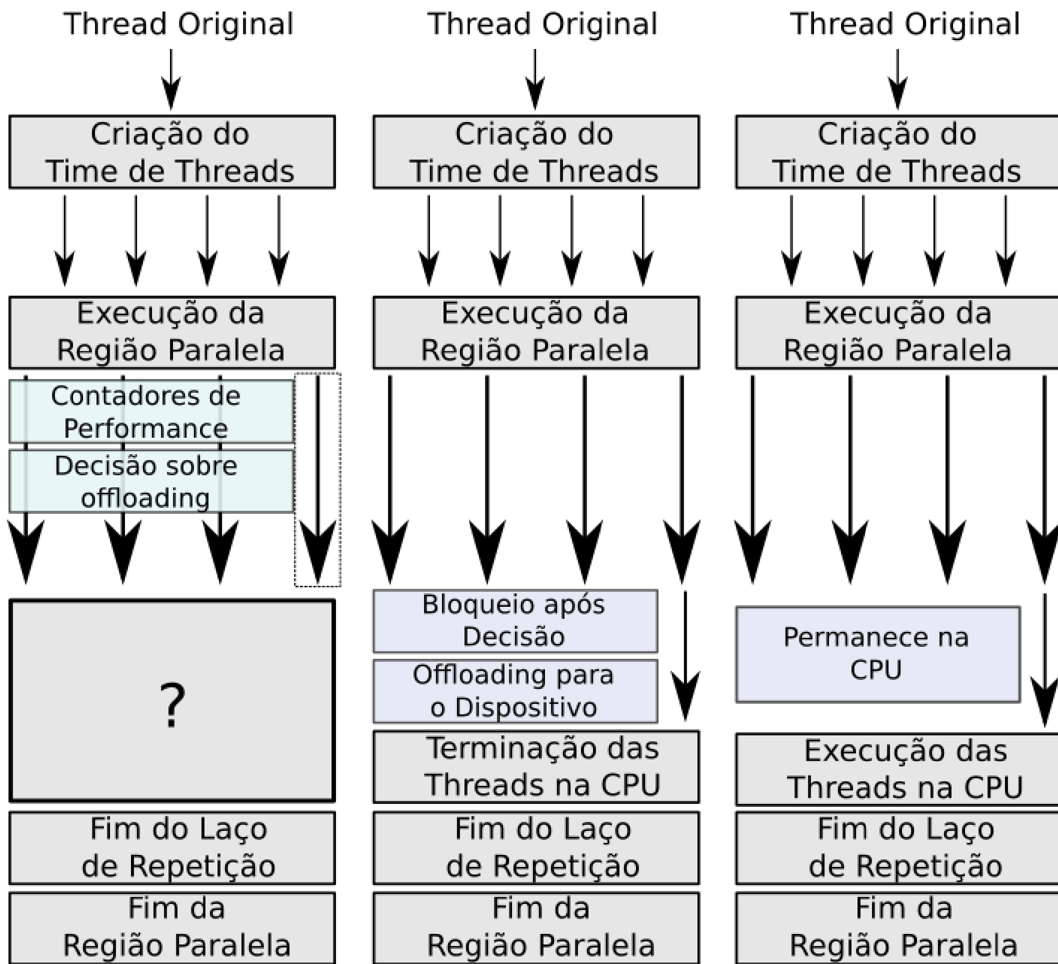

Figura 3.1. Execução do Código OpenMP. Baseado em (GONçALVES, 2016)

Caso a decisão seja pela execução na CPU, as *threads* continuam a existir, monitorando e executando o código. A visão geral do funcionamento e fluxo de execução do código observado pelo ambiente de execução é apresentado na Figura 3.2.

A biblioteca *libhookomp* implementa as funções que fazem a interceptação do código e extraem os dados para medição de desempenho. Esses dados são enviados para a biblioteca *librooflline*, que por sua vez faz o cálculo da *intensidade operacional*, baseando-se na quantidade de trabalho (quantidade de operações de ponto flutuante (W)) pela quantidade de bytes que foram movidos pela hierarquia de memória para executar essas instruções (Q), como mostrado na Equação 3.1.

$$I = \frac{W}{Q} \tag{3.1}$$

Para que o ambiente consiga fazer o *offloading* das funções destinadas para o dispositivo GPU é necessário construir uma tabela de funções alternativas que liste as funções e seus argumentos. Essa tabela por sua vez possui ponteiros para as funções na versão CUDA. Caso a decisão por *offloading* seja tomada, o ambiente de execução acessa essa tabela e lança a chamada da função *kernel* para a GPU por meio da reconstrução da chamada. Essa

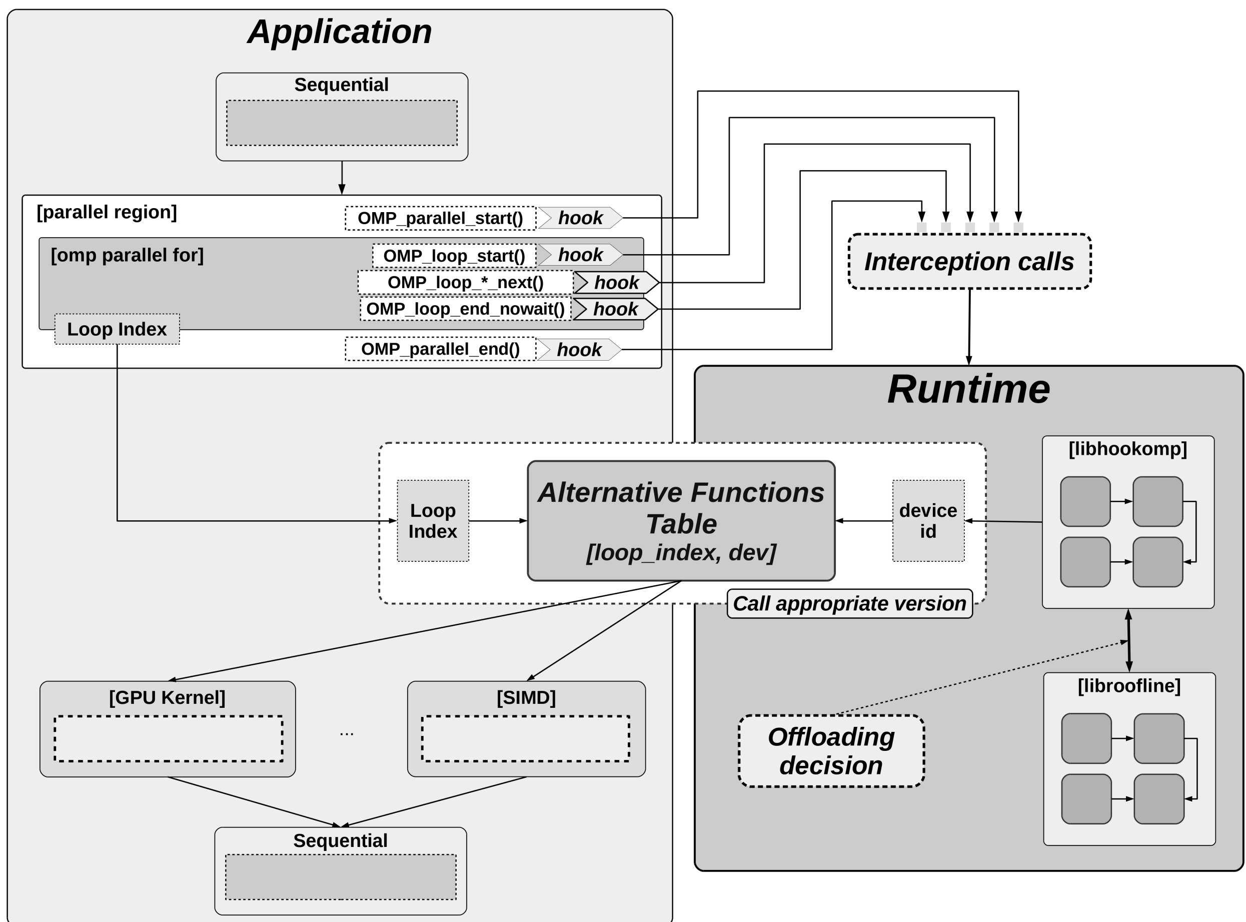

Figura 3.2. Arquitetura do ambiente de execução. Fonte: (GONçALVES, 2016)

reconstrução é feita utilizando como base a biblioteca `libffi` (SOURCEWARE, 2019), que provê ao programador uma série de convenções de chamada que permite, por exemplo, que o desenvolvedor chame uma função com base na sua assinatura e parâmetros (SOURCEWARE, 2019).

Essas informações são mantidas em uma estrutura chamada `Func`, como demonstrado no Código 3.1. Essa estrutura mantém todas as informações necessárias para que o construtor da biblioteca `libffi` seja capaz de encontrar a função e interpretá-la.

```c
/* Struct to store pointer and arguments to alternative functions calls. */
typedef struct Func {
    void *f;
    int nargs;
    ffi_type** arg_types;
    void** arg_values;
    ffi_type* ret_type;
    void* ret_value;
} Func;
```

Código-fonte 3.1. Estrutura que guarda os dados de uma função em C

3.2. Coleta de Métricas

Como citado anteriormente, para que o ambiente faça a decisão em si entre os diferentes dispositivos presentes no sistema heterogênea é necessário que um conjunto de métricas sejam analisadas para que os cálculos necessários sejam feitos.

Para fazer a coleta dos dados necessários utiliza-se o PAPI (PROJECT, 2019). O projeto `PAPI` tem o propósito de padronizar e implementar uma *API* eficiente e portável que seja capaz de acessar contadores de performances que existem na maioria dos processadores modernos atualmente e também em outros subsistemas (PROJECT, 2019). Esses contadores disponibilizam dados que podem ser utilizados por desenvolvedores para criar ferramentas de performance capazes de analisar ou otimizar código (PROJECT, 2019).

Os registros de memória que compõem os contadores são povoados pela contagem de **eventos** que ocorrem dentro de um determinado dispositivo durante a execução de um programa (PROJECT, 2019). Estes eventos por sua vez são ocorrências de sinais específicos relacionados ao funcionamento do dispositivo que está sendo monitorado, sendo que estes podem ser *hits* ou *misses* na memória *cache*, operações de ponto flutuante, leituras e escritas na memória e etc.(PAPI, 2019).

Estes tipos de eventos são monitorados devido a facilidade de criar uma correlação entre a estrutura do programa em execução e a eficiência do dispositivo de mapear dito código para a sua arquitetura em específico(PAPI, 2019).

Esses eventos podem ser divididos em dois grupos:

- **Eventos nativos**: Todos os contadores disponibilizados em uma determinada arquitetura, sendo que estes podem ser acessados diretamente pela interface do `PAPI`. Embora possível, é altamente improvável que um evento utilizado em uma arquitetura possua o mesmo nome em uma arquitetura diferente, tornando aplicações que utilizam esses eventos nativos exclusivos ao dispositivo pra quais foram compilados inviáveis em qualquer outra arquitetura (PAPI, 2019).
- **Eventos pré-determinados**: Conjunto de eventos comuns à boa parte dos dispositivos modernos, porém nomeados sob a nomenclatura própria do `PAPI`, criando um padrão que pode ser utilizado em diferentes arquiteturas (PAPI, 2019).

Para utilizar as métricas corretas relativas a arquitetura em que o ambiente de execução esta analisando e interceptando, é necessário defini-las em uma biblioteca separada para ser compilada juntamente com as bibliotecas *libhookomp* e *libroofline*. Essa biblioteca consiste na listagem de eventos utilizados pelo `PAPI` e informações sobre a velocidade de determinados eventos da arquitetura da `GPU` e `CPU`.

A arquitetura heterogênea utilizada neste trabalho para a realização de testes, e portanto, para a definição da biblioteca auxiliar de eventos foi a seguinte:

- `CPU`:

  - Processador: Intel(R) Core(TM) i3-8100 *Coffee Lake*
  - Frequência base: 3.60 GHz
  - Quantidade de Núcleos: 4
  - Quantidade de *threads*: 4
  - Arquitetura CPU: x86_64
  - Cache: 6 MB *SmartCache*
- GPU:
  - Modelo: GeForce GTX 780
  - Núcleos CUDA: 2304
  - *Clock* base: 863 MHz
  - Velocidade de memória: 6.0 Gbps
  - *Bandwidth*: 288.4
  - Arquitetura: Kepler

A Tabela 3.1 mostra a lista de eventos e dados considerados pelo ambiente de execução para a arquitetura utilizada neste trabalho. Podemos visualizar que são considerados contadores de escrita e leitura em cada nível de memória *cache* (L1, L2 e L3) e eventos de memória não *cache*.

Parâmetro	Evento de Leitura	Evento de Escrita
Memória	MEM_UOPS_RETIRED :ALL_LOADS	MEM_UOPS_RETIRED :ALL_STORES
Cache L3	PAPI_L3_DCR	PAPI_L3_DCW
Cache L2	PAPI_L2_DCR	PAPI_L2_DCW
Cache L1	PAPI_L1_DCR	PAPI_L1_DCW
Floating Point Op. (FPO)	PAPI_DP_OPS	-

Tabela 3.1. Lista de eventos capturados pelo PAPI

Já as tabelas 3.2 e 3.3 definem os detalhes de cada um dos dispositivos presentes na arquitetura heterogênea, de forma que os valores de *bandwidth* dos eventos de leitura e escrita e também o valor de latência destes eventos são especificados com base em *scripts* utilizados para determinar esses valores.

Effective Bandwidth

Evento	MEM_ALLOC_DEFAULT	MEM_ALLOC_PAGEABLE	MEM_ALLOC_PINNED
Leitura	21.0	21.0	21.0
Escrita	21.0	21.0	21.0

Latência

Evento	MEM_ALLOC_DEFAULT	MEM_ALLOC_PAGEABLE	MEM_ALLOC_PINNED
Leitura	1.0	1.0	1.0
Escrita	1.0	1.0	1.0

Tabela 3.2. Taxas de largura de banda e latência para os tipos de alocação de memória na CPU

Effective Bandwidth

Evento	MEM_ALLOC_DEFAULT	MEM_ALLOC_PAGEABLE	MEM_ALLOC_PINNED
Leitura	3.208838	3.208838	10.26707
Escrita	3.210552	3.210552	10.25683

Latência

Evento	MEM_ALLOC_DEFAULT	MEM_ALLOC_PAGEABLE	MEM_ALLOC_PINNED
Leitura	9.059737e-6	9.059737e-6	8.450636e-6
Escrita	6.212784e-6	6.212784e-6	7.787956e-6

Tabela 3.3. Taxas de largura de banda e latência para os tipos de alocação de memória na GPU

Capítulo 4

Desenvolvimento

Para auxiliar o processo de desenvolvimento da ferramenta de compilação, é necessário que um conjunto de passos sejam executados sobre um determinado programa inicial, escrito na linguagem C, para que a saída adequada para uso do ambiente de execução seja produzida. Consideramos estes passos como as etapas da arquitetura da ferramenta, definida previamente ao início do desenvolvimento com base nos *scripts* que eram utilizados pelo ambiente de execução e exemplificada na Figura 4.1.

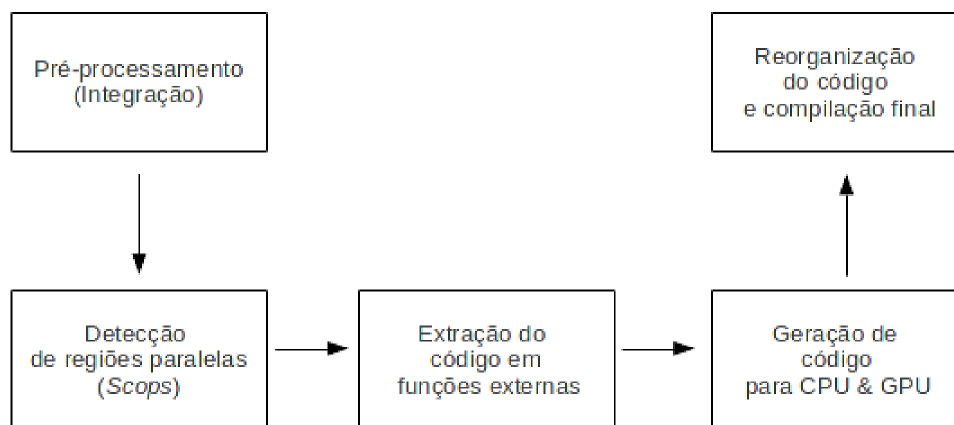

Figura 4.1. *Overview* da arquitetura da ferramenta de compilação.

Dentre as ferramentas capazes de auxiliar o desenvolvimento do compilador, descritas na Seção 2.4, a ferramenta `PoLLy` foi escolhida para aplicar as transformações e otimizações necessárias para a modificação do código. Como citado anteriormente na Seção 2.4.4, o `PoLLy` também serve de base para ferramentas como o `KernelGen`.

O `PoLLy` foi escolhido por ele apresentar um conjunto de funções (passos que podem ser utilizados dentro do contexto do `LLVM`) e variáveis de ambiente que podem ser utilizados individualmente, tornando mais flexível o desenvolvimento para um ambiente de execução tão específico. Esses passos nos auxiliam em quase todas as etapas de desenvolvimento. Além

disso, foi levado em consideração também a facilidade de leitura do código e disponibilidade de documentação.

Nas próximas seções, tratamos em maiores detalhes cada etapa da arquitetura definida, o uso da ferramenta `PoLLy`, os problemas que precisavam ser resolvidos pela ferramenta e as saídas que foram geradas em cada uma das etapas, além das restrições do programa desenvolvido.

4.1. Pré-processamento

Para que a ferramenta funcione completamente, é necessário que alguns parâmetros sejam definidos e algumas instruções sejam executadas dentro do código original para que seja possível a interceptação de código durante a execução. O compilador desenvolvido, no entanto, trabalha apenas com a representação `LLVM-IR`, tanto nas etapas intermediárias quanto na saída final do processo, impossibilitando a adição de código C diretamente nos arquivos intermediários e mesmo no código original.

Além disso, como o `PoLLy` aplica diversas otimizações e modifica o código original quase que completamente, sendo extremamente difícil inserir as instruções requeridas pelo ambiente no código final gerado. Portanto, antes de qualquer modificação feita pelo `PoLLy` ou pela `LLVM`, ou seja, diretamente no código C original, é necessário definir algumas restrições e trechos de código para que possamos dar continuidade na compilação. Nesta etapa, portanto, precisamos informar ao ambiente a quantidade de dispositivos utilizados, preencher a tabela de funções alternativas com os ponteiros para as funções destinadas à `GPU` e chamar a função destinada à `CPU`. Portanto, para cumprir com as necessidades da entrada, foi necessário estabelecer algumas restrições relacionadas à organização do código original. Essas restrições são:

- O código que deve ser analisado e paralelizado pela ferramenta deve estar dentro de uma função separada, já que o ambiente necessita que ambas estejam organizadas em funções distintas (uma destinada à `GPU` e outra à `CPU`), devido à necessidade de reconstrução das chamadas;
- A chamada dessa função deve ser simples (não sendo um parâmetro para uma função, sempre ser do tipo *void* e não deve possuir nenhum tipo de *casting*).

Além das restrição da organização do código, foi necessário também estabelecer um conjunto de palavras(*tags*) que devem ser adicionadas pelo programador para que elas sirvam como diretivas para que a ferramenta insira o código necessário nos locais corretos. Essas palavras são:

- `BEGIN_HYBRID` e `END_HYBRID`: A palavra `BEGIN_HYBRID` indica que uma função que vai ser analisada pelo compilador para verificar se a paralelização é possível está sendo

declarada e definida. Essa *tag* deve estar situada imediatamente antes do nome da função.

A palavra `END_HYBRID` indica o fim do corpo da função, sendo que deve estar sempre na última linha da função híbrida, ou seja, a função toda deve estar coberta para que o compilador funcione corretamente. Um exemplo do uso dessas palavras pode ser visto no Código 4.1.

```
void BEGIN_HYBRID foo() {
    /* Codigo oculto */
    END_HYBRID;
}
```

Código-fonte 4.1. Declaração e definição de uma função que pode ser otimizada (híbrida)

Portanto, se outras regiões do código que não estão no formato esperado forem consideradas paralelizáveis, a ferramenta vai simplesmente ignorá-las no momento em que código paralelo tenha que ser gerado para a `GPU` e `CPU`, sendo que as otimizações básicas serão feitas, mas não serão passadas para análise do ambiente de execução.

- `RUN_HYBRID` e `STOP_HYBRID`: Além das *tags* para definição e declaração da função, é necessário demarcar também o momento em que a chamada da função acontece, de forma que ela possa ser interceptada pelo ambiente. Para iniciar esse processo, utiliza-se a palavra `RUN_HYBRID`, que por sua vez vem sempre seguida do nome original da função que vai ser executada.

  Após a chamada, é necessário informar à ferramenta que a uma função híbrida foi executada e o fluxo de execução voltou para a origem por meio da palavra `STOP_HYBRID`. Um exemplo do uso dessas palavras chave pode ser visto no Código 4.2.

```
RUN_HYBRID foo;
    foo();
STOP_HYBRID;
```

Código-fonte 4.2. Chamada de uma função declarada como híbrida

Cada uma das funções anotadas são divididas em duas versões, uma com a *string* `_WITH_CPU` adicionada no nome da função e uma versão com `_WITH_GPU`. Essas funções são idênticas na lógica, sendo que a única diferença entre elas são as palavras chaves adicionadas para marcar pontos de captura de métricas especificas da `CPU` e da `GPU`, como mostrado nós Códigos 4.3 e 4.4.

O Código 4.3 mostra a função *foo*, descrita no Código 4.1 redefinida para a `GPU`, sendo que a *tag* *HOOKOMP_TIMING_DT_H2D_START* foi adicionada e defini o começo da execução para o ambiente de um *kernel*.

```
void foo_WITH_GPU(){
```

```
    HOOKOMP_TIMING_DT_H2D_START;
    //Codigo oculto
    HOOKOMP_TIMING_DT_H2D_STOP;
}
```

Código-fonte 4.3. Redefinição da função para a GPU

Já o Código 4.4 mostra uma redefinição para a CPU, seguindo a mesma lógica da função redefinida para a GPU.

```
void foo_WITH_CPU(){
    HOOKOMP_TIMING_OMP_START;
   //Codigo oculto
    HOOKOMP_TIMING_OMP_STOP;
}
```

Código-fonte 4.4. Redefinição da função para a CPU

Na chamada da função original (presente entre as *tags* RUN_HYBRID e STOP_HYBRID), o código responsável pelo preenchimento da tabela de funções é adicionado seguindo as especificações da função definida como híbrida. Um exemplo dessa construção é feita no Código 4.5, que mostra primeiro a alocação da tabela e seus dados, o seu preenchimento e então a chamada da função responsável pela criação da tabela no ambiente (create_target_functions_table).

```
int n_params_0 = 0;

Func *ff_0 = (Func *)malloc(sizeof(Func));
ff_0->arg_types = (ffi_type **)malloc((n_params_0 + 1) * sizeof(ffi_type *));
ff_0->arg_values = (void **)malloc((n_params_0 + 1) * sizeof(void *));
ff_0->f = &foo_WITH_GPU ;
memset(&ff_0->ret_value, 0, sizeof(ff_0->ret_value));

ff_0->ret_type = &ffi_type_void;
ff_0->nargs = n_params_0;
ff_0->arg_types[0] = NULL;
ff_0->arg_values[0] = NULL;
int nloops_0 = 1, ndevices_0 = 2;

if (create_target_functions_table(&table, nloops_0 , ndevices_0)) {
    assert(table != NULL);
    table[0][1][0] = *ff_0;
    TablePointerFunctions = table;
    assert(TablePointerFunctions != NULL);
}
```

Código-fonte 4.5. Criação da Tabela de Funções

4.2. Detecção de regiões paralelas

Tendo concluído a etapa de pré-processamento e obtido o código expandido que faz a conexão entre o código original e o ambiente de execução, é possível dar continuidade para a segunda etapa do processo de compilação. Essa etapa consiste na detecção das regiões paralelizáveis do código (SCoP) e pode ser subdividida em dois blocos sequenciais: execução de uma pré-otimização requerida pelo `PoLLy` e a detecção da região paralelizável propriamente dita. Os detalhes do funcionamento dessa etapa serão estudados em maiores detalhes nas Seções 4.2.1 e 4.2.2.

4.2.1. Reorganização (canonicalização) do código

Mesmo que um código qualquer esteja no formato `LLVM-IR`, o `PoLLy` ainda não é capaz de detectar as regiões paralelizáveis dentro dele. Para que isso seja possível, é necessário fazer uma série de otimizações que transformam o código para uma visão que `PoLLy` consiga analisar, assim como descrito no Capítulo 2.

Essa sub-etapa pode ser feita utilizando o passo definido dentro do projeto `PoLLy`, chamado `createPollyCanonicalizePass`. Para exemplificar o funcionamento desse passe, utilizamos Código 4.6, que passou pelo pelo pré-processamento e já está no formato `LLVM-IR`.

```
#define N 5
int vec[N];
void BEGIN_HYBRID foo() {
    int i, j;
    for (i = 0; i <= N; i++)
        vec[i] = i;
    END_HYBRID;
}
```

Código-fonte 4.6. Função original onde a canonicalização será aplicada.

A Figura 4.2 mostra a representação `LLVM-IR` do código, já a Figura 4.3 apresenta o mesmo trecho reorganizado, que mostra que o código foi encapsulado em um bloco que possui apenas uma entrada (`entry.split`) e condensado primariamente na região do *body*, incluindo também outras otimizações genéricas.

4.2.2. Detecção da área paralelizável

Imediatamente após a reorganização do código, podemos iniciar a segunda sub-etapa deste processo que consiste em detectar uma região paralelizável e gerar a sua representação em `LLVM-IR`. Ainda utilizando o Código 4.6 como exemplo, para aplicar os métodos necessários para completar esta etapa, utilizamos o passo `createScopDetectionPass` (responsável pela detecção) e então os passos `createDOTOnlyPrinterPass` e `createDOTPrinterPass`

```
entry:
  %i = alloca i32, align 4
  %j = alloca i32, align 4
  call void @hookomp_timing_start(i64* @omp_start)
  store i32 0, i32* %i, align 4
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %0 = load i32, i32* %i, align 4
  %cmp = icmp sle i32 %0, 5
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %1 = load i32, i32* %i, align 4
  %2 = load i32, i32* %i, align 4
  %idxprom = sext i32 %2 to i64
  %arrayidx = getelementptr inbounds [5 x i32], [5 x i32]* @vec, i64 0, i64 %idxprom
  store i32 %1, i32* %arrayidx, align 4
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %3 = load i32, i32* %i, align 4
  %inc = add nsw i32 %3, 1
  store i32 %inc, i32* %i, align 4
  br label %for.cond

for.end:                                          ; preds = %for.cond
  call void @hookomp_timing_stop(i64* @omp_stop)
  ret void
}
```

Figura 4.2. Código não canônico.

```
entry:
  br label %entry.split

entry.split:                                      ; preds = %entry
  tail call void @hookomp_timing_start(i64* nonnull @omp_start)
  br label %for.body

for.body:                                         ; preds = %for.body, %entry.split
  %indvars.iv = phi i64 [ 0, %entry.split ], [ %indvars.iv.next, %for.body ]
  %arrayidx = getelementptr inbounds [5 x i32], [5 x i32]* @vec, i64 0, i64 %indvars.iv
  %0 = trunc i64 %indvars.iv to i32
  store i32 %0, i32* %arrayidx, align 4
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %exitcond = icmp ne i64 %indvars.iv.next, 6
  br i1 %exitcond, label %for.body, label %for.end

for.end:                                          ; preds = %for.body
  tail call void @hookomp_timing_stop(i64* nonnull @omp_stop)
  ret void
}
```

Figura 4.3. Código canônico.

(responsáveis pela criação da representação visual desses SCoPs). Um exemplo dessas representações pode ser vista na Figura 4.4.

4.3. Extração do código em funções externas

Uma vez que à região paralelizável tenha sido detectada e sua representação gerada, é possível passar para a próxima etapa da ferramenta, consistindo na cópia dessas regiões paralelas (junto com suas dependências) do arquivo original para um novo arquivo secundário. No contexto do `LLVM-IR`, podemos dizer que essas regiões irão ocupar módulos diferentes do fluxo original do programa.

Essa etapa é indispensável devido à necessidade de criar blocos de código que possam

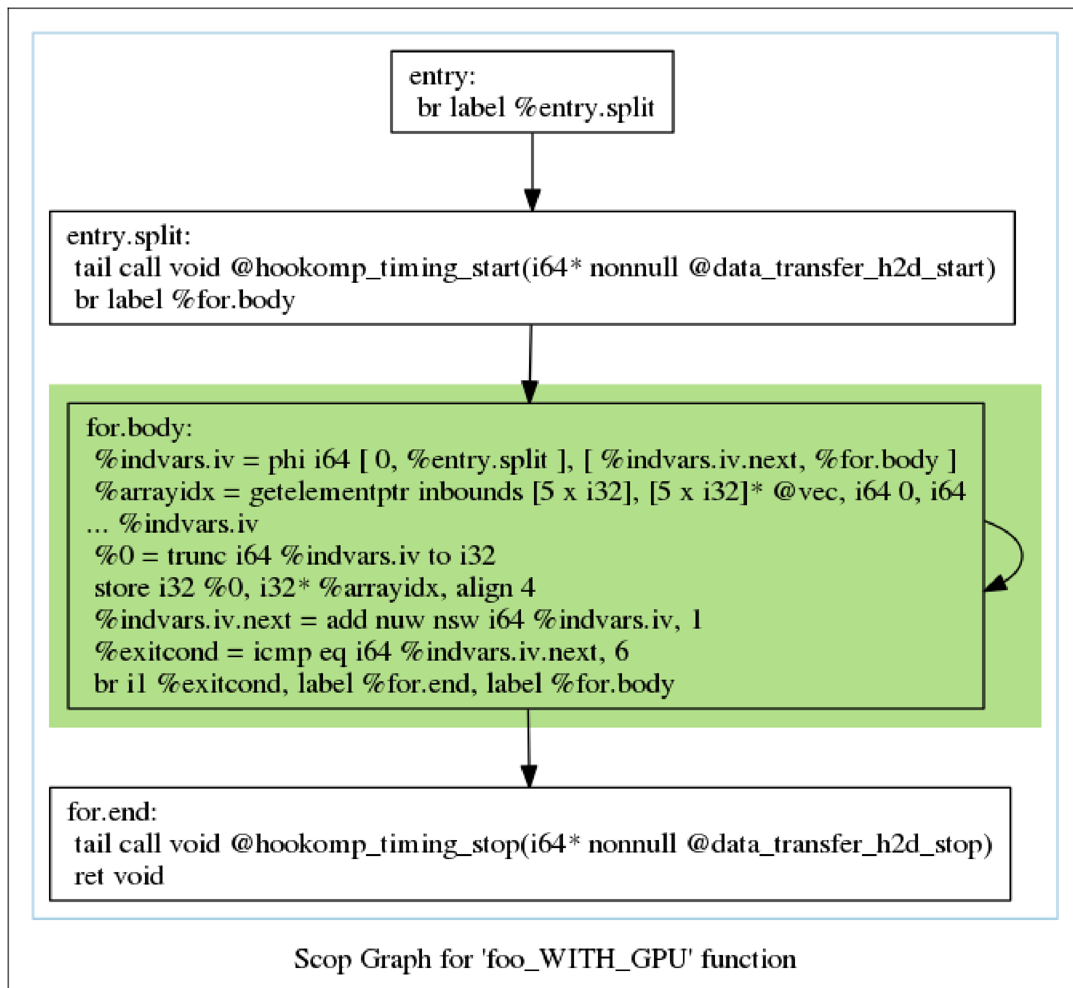

Figura 4.4. Representação gráfica do SCoP do Código 4.6.

ser compilados para a representação adequada e consequentemente possam ser descarregados para outros dispositivos (seja esse dispositivo `CPU` ou `GPU`). Se esta reorganização do código não ocorresse, só seria possível transformar o programa inteiro para ser executado em `CPU` ou em `GPU`, tornando o trabalho do ambiente de execução irrelevante. Essa funcionalidade é implementada em dois momentos, sendo eles: encapsulamento do código paralelo em uma função distinta e criação de um novo módulo para a função extraída.

4.3.1. Encapsulamento do código paralelo em uma função distinta

A primeira sub-etapa deste processo é a criação de uma nova função que contém o bloco de código paralelizável. Essa reorganização do código é feito pelo passo `createLoopExtractorPass`, que é um passo nativo do `LLVM`.

Para exemplificar o seu funcionamento, utilizamos o Código 4.7, que mostra um laço de repetição aninhado em outro laço, fazendo o preenchimento de uma matriz quadrada genérica. Após passar pelas etapas anteriores, chegamos ao código canônico e otimizado gerado pelo `PoLLy`, mostrado no Figura 4.5.

As modificações do passo `createLoopExtractorPass` feitas sobre esse código são mostradas na Figura 4.6, onde verificamos que a função `foo_WITH_GPU` chama uma outra rotina criada pelo `LLVM`, chamada `foo_WITH_GPU_for.cond1.preheader`, que contém o bloco paralelizável em sí.

```
#define N 5
float A[N][N];
void BEGIN_HYBRID foo() {
    for (int i = 0; i <= N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = 1;
        }
    }
    END_HYBRID;
}
```

Código-fonte 4.7. Função de preenchimento de uma matriz 5x5.

```
; Function Attrs: noinline nounwind uwtable
define void @foo_WITH_CPU() #0 {
entry:
  br label %entry.split

entry.split:                                      ; preds = %entry
  tail call void @hookomp_timing_start(i64* nonnull @omp_start)
  br label %for.cond1.preheader

for.cond1.preheader:                              ; preds = %for.inc6, %entry.split
  %indvars.iv3 = phi i64 [ 0, %entry.split ], [ %indvars.iv.next4, %for.inc6 ]
  br label %for.body3

for.body3:                                        ; preds = %for.body3, %
for.cond1.preheader
  %indvars.iv = phi i64 [ 0, %for.cond1.preheader ], [ %indvars.iv.next, %for.body3
]
  %arrayidx5 = getelementptr inbounds [5 x [5 x float]], [5 x [5 x float]]* @A, i64
0, i64 %indvars.iv3, i64 %indvars.iv
  store float 1.000000e+00, float* %arrayidx5, align 4
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %exitcond = icmp ne i64 %indvars.iv.next, 5
  br i1 %exitcond, label %for.body3, label %for.inc6

for.inc6:                                         ; preds = %for.body3
  %indvars.iv.next4 = add nuw nsw i64 %indvars.iv3, 1
  %exitcond5 = icmp ne i64 %indvars.iv.next4, 6
  br i1 %exitcond5, label %for.cond1.preheader, label %for.end8

for.end8:                                         ; preds = %for.inc6
  tail call void @hookomp_timing_stop(i64* nonnull @omp_stop)
  ret void
}
```

Figura 4.5. Código 4.7 otimizado pelo `PoLLy`.

```
; Function Attrs: noinline nounwind uwtable
define void @foo_WITH_GPU() #0 {
entry:
  br label %entry.split

entry.split:                                      ; preds = %entry
  tail call void @hookomp_timing_start(i64* nonnull @data_transfer_h2d_start)
  br label %codeRepl

codeRepl:                                         ; preds = %entry.split
  call void @foo_WITH_GPU_for.cond1.preheader()
  br label %for.end8

for.end8:                                         ; preds = %codeRepl
  tail call void @hookomp_timing_stop(i64* nonnull @data_transfer_h2d_stop)
  ret void
}

; Function Attrs: nounwind uwtable
define internal void @foo_WITH_CPU_for.cond1.preheader() #6 {
newFuncRoot:
  br label %for.cond1.preheader

for.end8.exitStub:                                ; preds = %for.inc6
  ret void

for.cond1.preheader:                              ; preds = %
for.inc6.for.cond1.preheader_crit_edge, %newFuncRoot
  %indvars.iv3 = phi i64 [ 0, %newFuncRoot ], [ %indvars.iv.next4, %
for.inc6.for.cond1.preheader_crit_edge ]
  br label %for.body3

for.body3:                                        ; preds = %
for.body3.for.body3_crit_edge, %for.cond1.preheader
  %indvars.iv = phi i64 [ 0, %for.cond1.preheader ], [ %indvars.iv.next, %
for.body3.for.body3_crit_edge ]
  %arrayidx5 = getelementptr inbounds [5 x [5 x float]], [5 x [5 x float]]* @A, i64
0, i64 %indvars.iv3, i64 %indvars.iv
  store float 1.000000e+00, float* %arrayidx5, align 4
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %exitcond = icmp eq i64 %indvars.iv.next, 5
  br i1 %exitcond, label %for.inc6, label %for.body3.for.body3_crit_edge

for.body3.for.body3_crit_edge:                    ; preds = %for.body3
  br label %for.body3

for.inc6:                                         ; preds = %for.body3
  %indvars.iv.next4 = add nuw nsw i64 %indvars.iv3, 1
  %exitcond5 = icmp eq i64 %indvars.iv.next4, 6
  br i1 %exitcond5, label %for.end8.exitStub, label %
for.inc6.for.cond1.preheader_crit_edge

for.inc6.for.cond1.preheader_crit_edge:           ; preds = %for.inc6
  br label %for.cond1.preheader
}
```

Figura 4.6. Função (`foo_WITH_GPU_for.cond1.preheader`) e sua função mãe.

4.3.2. Criação de um novo módulo para a função

Tendo as funções novas criadas pelo LLVM, é possível fazer uma cópia delas, juntamente com as suas dependências (variáveis globais que estão sendo utilizadas e chamadas de funções feitas dentro do código), de forma que ela possa ser montada em um módulo separado, que por sua vez é invocado no módulo original. Essa funcionalidade foi construída com base na

ferramenta `LLVM-extract`, também presente no projeto `LLVM`.

O algoritmo deste passo é estruturado da seguinte forma: O arquivo contendo o módulo com a região paralelizável é aberto e lido. Com base nos nomes das funções extraídas no passo anterior, a representação da função (junto com as suas dependências) é reconstruída em um novo módulo por meio das ferramentas disponibilizadas pelo `LLVM`. Essa nova representação é então escrita em um novo arquivo, cujo nome é o mesmo da função (neste caso, *foo_WITH_GPU_for.cond1.preheader*), que será utilizado nos próximos passos do compilador.

4.4. Geração de código para CPU e GPU

A geração do código final das regiões paralelas consiste na transformação das funções que foram extraídas no passo anterior para o código final que possa ser executado na `CPU` (funções que possuem a *tag* `WITH_CPU` no nome) e `GPU` (com o nome `WITH_GPU`), sendo que ambas possuem a mesma funcionalidade.

Neste momento, as especificações relacionadas à organização das *threads* e seus lançamentos dentro do código são definidas pelo `PoLLy` (sendo isso por meio de uma chamada de uma função interna ,diretamente por um comando ou por meio da definição de variáveis dentro do projeto `PoLLy`), embora sejam utilizados passos distintos para cada dispositivo da plataforma alvo. A quantidade de *threads* utilizadas pelo programa pode ser configurada no caso da `CPU`, por meio da definição de variável de controle no momento da compilação, e possui um valor padrão no caso da `GPU`, determinado pela quantidade de núcleos presente na arquitetura específica.

O tipo de escalonamento das *threads* da `CPU` é definido por meio de uma variável interna do projeto `PoLLy`, localizada dentro do arquivo *LoopGenerators.cpp*, que deve ser mantida como padrão para o escalonamento estático, ou modificada para o tipo de preferência (dinâmico, etc.). Após essa definição, é necessário recompilar o projeto `LLVM` por completo para que ela tome efeito.

A geração de código acontece em dois momentos distintos (sendo em que a ordem em que eles ocorrem é irrelevante): geração de código para a `CPU` e geração para a `GPU`.

4.4.1. Geração do código destinado à CPU

A geração do código destinado à `CPU` não é feito internamente por um passo existente do `PoLLy` ou da `LLVM`, mas sim por meio de uma chamada direta ao otimizador da `LLVM`. A chamada aplicada sobre a rotina é listada no Código 4.8. Como citado anteriormente, o otimizador `opt` é chamado com as *flags* necessárias gerando o binário do arquivo original. Após a geração do binário (`.bc`), o código é transformado novamente para o formato `LLVM-IR` e finalmente pode passar para a etapa final da ferramenta.

```
opt aplicacao.ll -O3 -polly-codegen -polly-vectorizer=polly -polly-parallel -polly-process-
    unprofitable -polly-num-threads=<qtd> > <nome.bc>
opt -S <nome.bc> > <nome_gomp.ll>
```

Código-fonte 4.8. Chamada do otimizador da LLVM para gerar código *multithread*

Para mostrar os efeitos dessa otimização, utilizamos o código 4.9, produzido a partir da função mostrada na Figura 4.6. Além de fazer as chamadas ao sistema para a criação das *threads* e quando elas devem ser lançadas na CPU, o opt modifica levemente a estrutura do código para adequar as chamadas das funções internas que são invocadas, como a GOMP_loop_runtime_next. Esse código foi gerado utilizando as configurações padrões (escalonamento estático com a quantidade padrão de de *threads* definida pela CPU, sendo esta no caso um total de quatro *threads*) sendo ele final, não sendo necessário passar por mais nenhuma modificação ou otimização.

```
define internal void @foo_WITH_CPU_for_cond1_preheader_polly_subfn(i8* nocapture readnone %
    polly.par.userContext) #2 {
polly.par.setup:
  %polly.par.LBPtr = alloca i64, align 8
  %polly.par.UBPtr = alloca i64, align 8
  %0 = call i8 @GOMP_loop_dynamic_next(i64* nonnull %polly.par.LBPtr, i64* nonnull %
      polly.par.UBPtr)
  %1 = icmp eq i8 %0, 0
  br i1 %1, label %polly.par.exit, label %polly.par.loadIVBounds.preheader

polly.par.loadIVBounds.preheader:                 ; preds = %polly.par.setup
  br label %polly.par.loadIVBounds

polly.par.exit.loopexit:                          ; preds = %polly.par.checkNext.loopexit
  br label %polly.par.exit

polly.par.exit:                                   ; preds = %polly.par.exit.loopexit, %
    polly.par.setup
  call void @GOMP_loop_end_nowait()
  ret void

polly.par.checkNext.loopexit:                     ; preds = %polly.loop_header
  %2 = call i8 @GOMP_loop_dynamic_next(i64* nonnull %polly.par.LBPtr, i64* nonnull %
      polly.par.UBPtr)
  %3 = icmp eq i8 %2, 0
  br i1 %3, label %polly.par.exit.loopexit, label %polly.par.loadIVBounds

polly.par.loadIVBounds:                           ; preds = %
    polly.par.loadIVBounds.preheader, %polly.par.checkNext.loopexit
  %polly.par.LB = load i64, i64* %polly.par.LBPtr, align 8
  %polly.par.UB = load i64, i64* %polly.par.UBPtr, align 8
  %polly.adjust_ub = add i64 %polly.par.UB, -2
  br label %polly.loop_header

polly.loop_header:                                ; preds = %polly.loop_header, %
    polly.par.loadIVBounds
      /*
      Codigo suprimido
```

```
    */
  br i1 %polly.loop_cond , label %polly.par.checkNext.loopexit , label %polly.loop_header
}
```

Código-fonte 4.9. Código final gerado para a `CPU`

4.4.2. Geração do código para a GPU

Para gerar código para a `GPU`, aplicamos um passo existente dentro do projeto `LLVM`, conhecido como `createPPCGCodeGenerationPass`, sobre o módulo que contém a função extraída. Nessa transformação, o `PoLLy` faz a alocação de blocos e *threads* dentro da `GPU`, a alocação de memória necessária para que o programa seja transferido, a cópia dos dados e também a liberação da memória após a execução.

```
define hidden void @foo_WITH_GPU_for.cond1.preheader() #0 {
newFuncRoot:
  %polly_launch_0_params = alloca [1 x i8*]
  %polly_launch_0_param_0 = alloca i8*
  %polly_launch_0_params_i8ptr = bitcast [1 x i8*]* %polly_launch_0_params to i8*
  br label %polly.split_new_and_old

polly.merge_new_and_old:                          ; preds = %polly.exiting , %for.inc6
  br label %for.end8.exitStub

for.end8.exitStub:                                ; preds = %polly.merge_new_and_old
  ret void

polly.split_new_and_old:                          ; preds = %newFuncRoot
  /*
    Codigo suprimido
  */
  br i1 %polly.rtc.result , label %polly.start , label %for.cond1.preheader.pre_entry_bb

for.cond1.preheader.pre_entry_bb:                 ; preds = %polly.split_new_and_old
  br label %for.cond1.preheader

for.cond1.preheader:                              ; preds = %
    for.inc6.for.cond1.preheader_crit_edge , %for.cond1.preheader.pre_entry_bb
  %indvars.iv3 = phi i64 [ %indvars.iv.next4 , %for.inc6.for.cond1.preheader_crit_edge ] , [
      0, %for.cond1.preheader.pre_entry_bb ]
  br label %for.body3

for.body3:                                        ; preds = %for.body3.for.body3_crit_edge ,
    %for.cond1.preheader
 /*
    Codigo suprimido
 */
  br i1 %exitcond , label %for.inc6 , label %for.body3.for.body3_crit_edge

for.body3.for.body3_crit_edge:                    ; preds = %for.body3
  br label %for.body3

for.inc6:                                         ; preds = %for.body3
```

```
  %indvars.iv.next4 = add nuw nsw i64 %indvars.iv3 , 1
  %exitcond5 = icmp eq i64 %indvars.iv.next4 , 6
  br i1 %exitcond5 , label %polly.merge_new_and_old , label %
      for.inc6.for.cond1.preheader_crit_edge

for.inc6.for.cond1.preheader_crit_edge:           ; preds = %for.inc6
  br label %for.cond1.preheader

polly.start:                                      ; preds = %polly.split_new_and_old
  br label %polly.acc.initialize

polly.acc.initialize:                             ; preds = %polly.start
  %6 = call i8* @polly_initContext()
  %p_dev_array_MemRef_A = call i8* @polly_allocateMemoryForDevice(i64 120)
  %7 = call i8* @polly_getDevicePtr(i8* %p_dev_array_MemRef_A)
  %8 = getelementptr [1 x i8*], [1 x i8*]* %polly_launch_0_params, i64 0, i64 0
  store i8* %7, i8** %polly_launch_0_param_0
  %9 = bitcast i8** %polly_launch_0_param_0 to i8*
  store i8* %9, i8** %8
  %10 = call i8* @polly_getKernel(i8* getelementptr inbounds ([545 x i8], [545 x i8]*
      @kernel_0, i32 0, i32 0), i8* getelementptr inbounds ([9 x i8], [9 x i8]*
      @kernel_0_name, i32 0, i32 0))
  call void @polly_launchKernel(i8* %10, i32 1, i32 1, i32 6, i32 5, i32 1, i8* %
      polly_launch_0_params_i8ptr)
  call void @polly_freeKernel(i8* %10)
  call void @polly_copyFromDeviceToHost(i8* %p_dev_array_MemRef_A, i8* bitcast ([5 x [5 x
      float]]* @A to i8*), i64 120)
  call void @polly_freeDeviceMemory(i8* %p_dev_array_MemRef_A)
  call void @polly_freeContext(i8* %6)
  br label %polly.exiting

polly.exiting:                                    ; preds = %polly.acc.initialize
  br label %polly.merge_new_and_old
}
```

Código-fonte 4.10. Função `foo` destinada para a `GPU`

É importante notar também que o `PoLLy` faz uma separação do código gerado em dois blocos: um bloco *new*, contendo o código destinado à `GPU` e um bloco *old*, que é o código prévio à geração sem nenhuma modificação. Utilizando o Código 4.10 como exemplo da transformação feita sobre a função mostrada na Figura 4.6, podemos perceber que de acordo com a linha especificada no Código 4.11, o `PoLLy` faz um *branch* entre a chamada do bloco *old* e *new* de acordo com um parâmetro interno.

```
br i1 %polly.rtc.result , label %polly.start , label %for.cond1.preheader.pre_entry_bb
```

Código-fonte 4.11. *Split* entre novo bloco de código e antigo

Porém, já que não existe interesse em executar o código antigo no contexto deste projeto, o seguinte *branch* é modificado para sempre ser direcionado para o bloco novo que executa o código na `GPU`, como mostrado no Código 4.12.

```
br label %polly.start
```

Código-fonte 4.12. Novo *branch* direcionado sempre para a GPU

4.5. Reorganização do código e compilação final

A etapa final da ferramenta também é formada por duas sub-etapas distintas. A primeira delas sendo a reorganização do código principal (onde as chamadas para as funções paralelas são feitas), de forma que seja possível formar o *link* entre as todas as funções executadas pelo programa. Já a segunda etapa é a compilação final do código para a criação dos arquivos binários e por fim o executável final que vai ser utilizado dentro do ambiente de execução.

4.5.1. Reorganização do código

A reorganização do código consiste em remover do arquivo original as definições das funções que foram extraídas para novos arquivos. Por exemplo, considere o Código 4.13, que mostra o cabeçalho de uma função `foo` parcialmente otimizado pelo `PoLLy`, onde a função com a região paralela é ao mesmo tempo declarada e definida.

```
; Function Attrs: nounwind uwtable
define internal void @init_WITH_GPU_for.cond1.preheader() #6 {
/*  Corpo da funcao */
}
```

Código-fonte 4.13. Definição e declaração de uma função extraída

Nesta sub-etapa, todas as funções extraídas são modificadas, sendo que o corpo da função é removido e apenas a declaração da função é mantida, funcionando como o cabeçalho. Desta forma, quando o código passar pela compilação final e os binários forem conectados, as funções paralelas serão chamadas como uma biblioteca externa e executadas corretamente. O Código 4.14 exemplifica o formato final da função no arquivo principal.

```
; Function Attrs: nounwind uwtable
declare void @foo_WITH_CPU_for.cond1.preheader()
```

Código-fonte 4.14. Definição de um cabeçalho no formato LLVM-IR

4.5.2. Geração dos executáveis

A compilação final é a simples geração do executável que será interceptado e analisado pelo ambiente de execução. Para atingir esse objetivo, primeiramente são gerados os binários para as funções `CPU`, por meio do comando definido em 4.15.

```
clang nome_do_arquivo -o nome_binario
```

Código-fonte 4.15. Geração de binários para a CPU

Seguindo essa chamada ao *clang*, a compilação do código para a GPU é feita, utilizando o `llc`, que é o compilador do projeto LLVM. O comando utilizado é mostrado em 4.16.

```
llc nome_do_arquivo -o nome_binario
```

Código-fonte 4.16. Geração de binários para a GPU

Por fim, o *link* entre esses binários é feito por meio do comando 4.17.

```
clang <nomes_dos_binarios> -o Temporary/GOMP+PPCG/hybrid.exe -fopenmp -lgomp -lffi -ldl -
    lcuda -lcudart -lroofline -lpapi -lGPURuntime
```

Código-fonte 4.17. Geração do executável

4.6. Conclusões

Neste capítulo, foram apresentados os detalhes sobre o desenvolvimento e funcionamento da ferramenta de compilação, que produz código híbrido CPU/GPU de forma que esse executável final possa ser utilizado em conjunto com o ambiente de execução para fazer o *offloading* de código para arquiteturas heterogêneas, passando desde um arquivo original em C e chegando até a geração do código executável.

Nós próximos Capítulos, serão definidos os testes e como eles foram produzidos, assim como os resultados atingidos por este projeto.

Capítulo 5

Experimentos e Resultados

Neste capítulo serão apresentados os experimentos realizados utilizando a ferramenta de compilação produzida neste projeto. Tais resultados são compostos pelas principais saídas, parciais e finais, geradas durante a execução dos passos determinados no Capítulo 3. Os dados aqui apresentados foram obtidos por meio do uso do compilador sobre um conjunto de casos de teste preparados para esta ferramenta em específico.

Nas próximas seções deste capítulo serão discutidos os detalhes dos casos de testes definidos, suas especificações e funcionalidades, as saídas geradas e por fim será feita discussão sobre o trabalho, os problemas enfrentados durante o desenvolvimento, as implicações dos resultados produzidos e as possíveis melhorias e correções que devem ser feitas no futuro.

5.1. Casos de Teste e Experimentos

Para demonstrar o trabalho feito pela ferramenta de compilação e como ela funcionou tanto de forma independente quanto em conjunto com o ambiente de execução, foram definidas duas provas de conceito que exemplificam o que aconteceu em cada uma das etapas da transformação do código. Para que esses programas pudessem ser interceptados pelo ambiente, foi necessário definir o formato de escalonamento como dinâmico dentro do `PoLLy`, já que a medição dos eventos do `PAPI` são determinados a partir de *chunks* de código, sendo necessário fazer uma chamada de uma função específica da biblioteca de *threads* para que e o ambiente consiga intercepta-la.

O planejamento inicial para testar o trabalho aqui desenvolvido era criar um conjunto de *benchmarks* utilizando primariamente o *Polybench* (POUCHET et al., 2012), porém foi decidido que realizar provas de conceito seriam suficientes para o contexto deste projeto. Além disso, ocorreram problemas com a integração da ferramenta com o ambiente, não sendo possível concluir os testes do sistema completo. Esses problemas serão discutidos em maior

profundidade na Seção 5.2.

Portanto, neste trabalho utilizamos dois casos de teste: Adição de matriz simples e Multiplicação de matrizes. Esses exemplos variam em níveis de complexidades devido à quantidade de operações em ponto flutuante e em níveis de trabalho que devem produzir, visto que a quantidade de dados que serão manipulados pelos programas e por consequência devem ser copiados para o dispositivo GPU são bastante diferentes.

Na tentativa de mostrar todas as possibilidades de testes e o funcionamento do compilador, foram produzidas duas versões distintas da ferramenta, sendo que uma delas tenta fazer a utilização das bibliotecas do ambiente de execução e a outra é utilizada para que seja feita o *offloading* sem fazer a coleta de métricas, já que o dispositivo de execução é escolhido a partir da linha de comando no momento de execução do programa. Para a visualização da divisão de trabalho deste código em ambos os dispositivos, foi utilizado a ferramenta de perfilamento `nvprof`, que possibilita a visualização do lançamento das *threads* e outros detalhes sobre a transferência de dados, tanto na CPU quanto na GPU.

Para a análise da execução na CPU, utiliza-se o comando mostrado no Código 5.1, que por sua vez permite que cada *thread* seja visualizada separadamente. Já no caso da GPU, a visualização e *profiling* da execução do *kernel* é criada a partir de uma chamada simples ao *nvprof*.

```
nvprof --cpu-profiling on --cpu-profiling-thread-mode separated ./<nome_do_executavel>
```

Código-fonte 5.1. Chamada para do *tracking* da CPU

Os próximos exemplos mostrados foram todos produzidos a partir do mesmo conjunto de opções, sendo que o valor padrão de *threads* foi definido como oito (no máximo) trabalhando sobre o escalonamento dinâmico.

5.1.1. Adição de Matrizes

Para realizar o primeiro teste, foi utilizado um algoritmo simples de adição de matrizes, onde são declaradas três matrizes globais quadradas de tamanho $N = 100$. Neste caso, duas matrizes são inicializadas com valores padrão e depois somadas e inseridas em uma terceira matriz.

```
#define N 100
float A[N][N], B[N][N], C[N][N];
void init_matrix() {
        int i, j;
        for (i = 0; i <= N; i++) {
                for (j = 0; j < N; j++){
                        A[i][j] = 1.0;
                        B[i][j] = 1.0;
                }
        }
```

```
}
void BEGIN_HYBRID add_matrix() {
        int i, j;
        for (i = 0; i < N; i++)
                for (j = 0; j < N; j++)
                        C[i][j] = A[i][j] + B[i][j];
        END_HYBRID;
}
int main() {
        init_matrix();
        RUN_HYBRID add_matrix;
        add_matrix();
        STOP_HYBRID;
}
```

Código-fonte 5.2. Código original de Adição de Matrizes

Como mostrado no Capítulo 4, as regiões anotadas são expandidas, divididas em versão GPU e CPU e então é feita a análise da região paralelizável. A representação gráfica da região paralelizável pode ser vista na Figura 5.3.

Em um primeiro momento, este código foi gerado para a CPU e então executado utilizando o comando listado no no Código 5.1. A saída gerada por esta chamada, vista na Figura 5.1, mostra que a quantidade adequada de *threads* foram lançadas na CPU (sete delas foram utilizadas para executar o programa). Dado a baixa complexidade do código, a ferramenta *nvprof* gera um perfilamento com poucas informações, não sendo capaz de listar o tempo em que cada uma das *threads* foram lançadas.

```
======== Thread 140459596069376
 100.00%       0us  __clone
 100.00%       0us  | start_thread
 100.00%       0us  |   _ZL19__kmp_launch_workerPv
 100.00%       0us  |     __kmp_launch_thread
 100.00%       0us  |       __kmp_fork_barrier(int, int)
 100.00%       0us  |         sched_yield

======== Thread 140459676907456
 100.00%       0us  main
 100.00%       0us  | add_matrix_WITH_CPU
 100.00%       0us  |   add_matrix_WITH_CPU_.preheader
 100.00%       0us  |     __kmp_join_call
 100.00%       0us  |       __kmp_internal_join
 100.00%       0us  |         __kmp_join_barrier(int)
 100.00%       0us  |           sched_yield

======== Thread 140459523435136
 100.00%       0us  __clone
 100.00%       0us  | start_thread
 100.00%       0us  |   _ZL19__kmp_launch_workerPv
 100.00%       0us  |     __kmp_launch_thread
 100.00%       0us  |       __kmp_fork_barrier(int, int)
 100.00%       0us  |         sched_yield

======== Thread 140459608664192
 100.00%       0us  __clone
 100.00%       0us  | start_thread
 100.00%       0us  |   _ZL19__kmp_launch_workerPv
 100.00%       0us  |     __kmp_launch_thread
 100.00%       0us  |       __kmp_fork_barrier(int, int)
 100.00%       0us  |         sched_yield

======== Thread 140459600267648
 100.00%       0us  __clone
 100.00%       0us  | start_thread
 100.00%       0us  |   _ZL19__kmp_launch_workerPv
 100.00%       0us  |     __kmp_launch_thread
 100.00%       0us  |       __kmp_fork_barrier(int, int)
 100.00%       0us  |         sched_yield

======== Thread 140459604465920
 100.00%       0us  __clone
 100.00%       0us  | start_thread
 100.00%       0us  |   _ZL19__kmp_launch_workerPv
 100.00%       0us  |     __kmp_launch_thread
 100.00%       0us  |       __kmp_fork_barrier(int, int)
 100.00%       0us  |         sched_yield

======== Thread 140459617060736
 100.00%       0us  __clone
 100.00%       0us  | start_thread
 100.00%       0us  |   _ZL19__kmp_launch_workerPv
 100.00%       0us  |     __kmp_launch_thread
 100.00%       0us  |       __kmp_fork_barrier(int, int)
 100.00%       0us  |         sched_yield
```

Figura 5.1. Lista de *threads* na *cpu* geradas pelo programa.

No caso da execução da GPU, o perfilamento pode ser visto na Figura 5.2. Nesta lista, podemos ver todas as chamadas de funções que são feitas relacionadas à ao dispositivo

acelerador, como as atividades de alocação de memória do *host* para o *device* (*HtoD*) e de volta do *device* para o *host*.

```
==26185== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 45.82%  11.936us         2  5.9680us  5.9200us  6.0160us  [CUDA memcpy HtoD]
 34.03%  8.8640us         1  8.8640us  8.8640us  8.8640us  kernel_0
 20.15%  5.2480us         1  5.2480us  5.2480us  5.2480us  [CUDA memcpy DtoH]

==26185== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 98.55%  68.166ms         1  68.166ms  68.166ms  68.166ms  cuCtxCreate
  0.63%  436.64us         1  436.64us  436.64us  436.64us  cuLinkCreate
  0.17%  118.06us         3  39.354us  3.8010us  106.76us  cuMemAlloc
  0.14%  97.600us         3  32.533us  4.8530us  85.679us  cuMemFree
  0.13%  88.541us         1  88.541us  88.541us  88.541us  cuLinkComplete
  0.10%  66.201us         1  66.201us  66.201us  66.201us  cuLinkAddData
  0.09%  61.566us         1  61.566us  61.566us  61.566us  cuModuleLoadData
  0.06%  42.975us         1  42.975us  42.975us  42.975us  cuMemcpyDtoH
  0.05%  35.642us         2  17.821us  17.544us  18.098us  cuMemcpyHtoD
  0.04%  30.918us         1  30.918us  30.918us  30.918us  cuDeviceGetName
  0.02%  15.436us         1  15.436us  15.436us  15.436us  cuLaunchKernel
  0.00%  2.3110us         3     770ns     127ns  1.4850us  cuDeviceGetCount
  0.00%  1.6840us         1  1.6840us  1.6840us  1.6840us  cuLinkDestroy
  0.00%     733ns         4     183ns     135ns     234ns  cuDeviceGetAttribute
  0.00%     726ns         3     242ns     187ns     325ns  cuDeviceGet
  0.00%     455ns         1     455ns     455ns     455ns  cuModuleGetFunction
```

Figura 5.2. Lista de *threads* na GPU geradas pelo programa.

Podemos perceber também que, visto que a ferramenta é primariamente utilizada para medição e perfilamento para a GPU, é possível captar muito mais informações sobre o funcionamento do programa no dispositivo acelerador. Também podemos observar que a performance é carregada primariamente pelo tempo de cópia de memória para e da GPU, que neste caso ocupou cerca de 65% do tempo.

O resultado gerado pela execução dessa aplicação dentro de ambos os dispositivos pode ser visto no Código 5.3, que mostra a posição da matriz final e o valor calculado pela operação de soma.

```
R[99][90] = 2.000000
R[99][91] = 2.000000
R[99][92] = 2.000000
R[99][93] = 2.000000
R[99][94] = 2.000000
R[99][95] = 2.000000
R[99][96] = 2.000000
R[99][97] = 2.000000
R[99][98] = 2.000000
R[99][99] = 2.000000
```

Código-fonte 5.3. Resultado parcial da adição de matrizes.

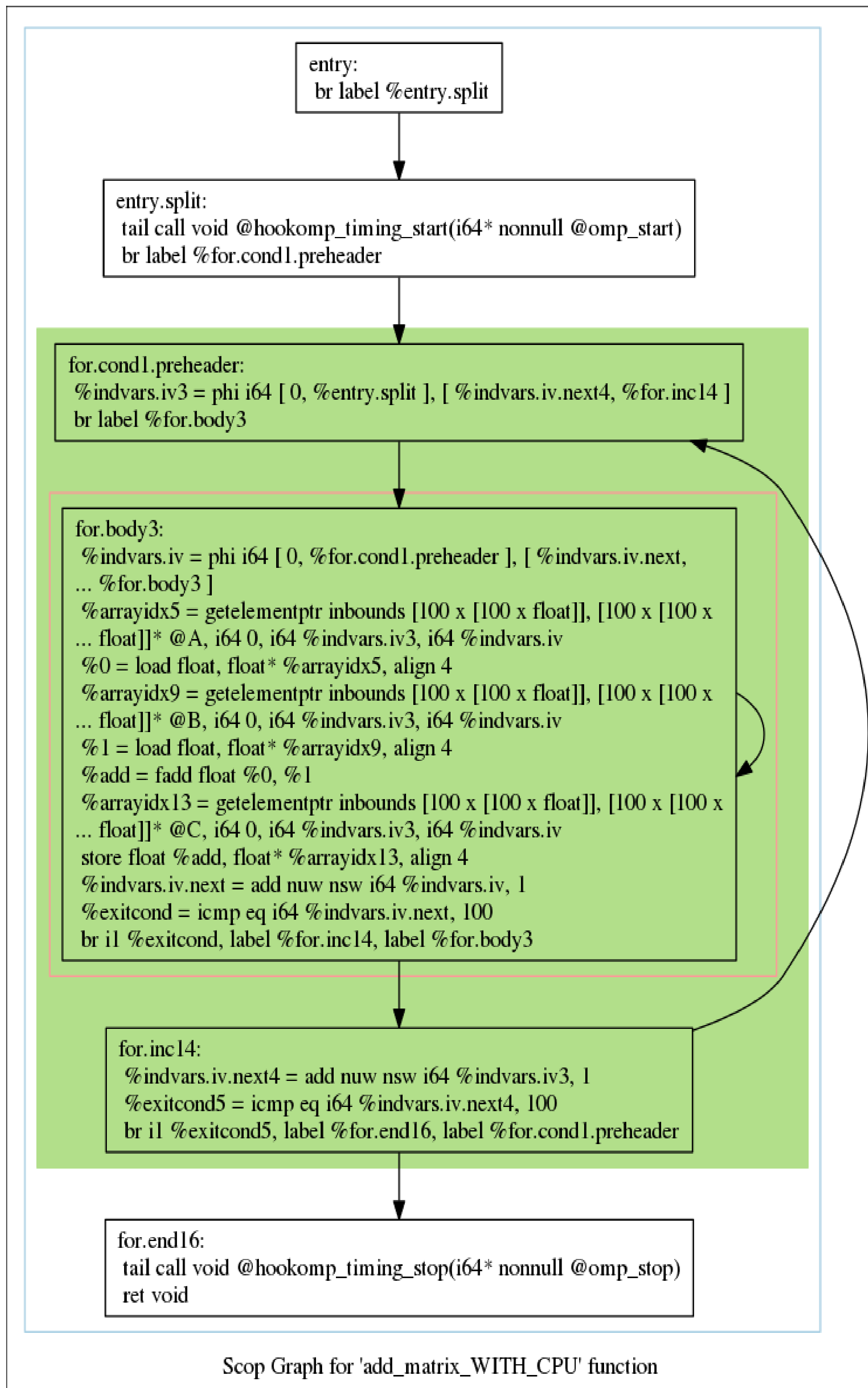

Figura 5.3. Região paralelizável detectada na função `add_matrix`.

5.1.2. Multiplicação de Matrizes

O exemplo de multiplicação de matrizes segue o mesmo padrão do exemplo anterior, ou seja, um conjunto de operações são realizadas sobre uma série de matrizes de tamanho padrão, sendo que a função responsável pela operação de multiplicação entre os valores inicializados dentro das variáveis é marcada para que a ferramenta analise e gere código híbrido para os dispositivos.

```
#include <stdio.h>
#define NX 10000
#define NJ 7500
#define DATA_TYPE float
float A[NX][NJ], B[NX][NJ], C[NX][NJ], D[NX][NJ];
float E[NX][NJ], F[NX][NJ],G[NX][NJ];
void init_array() {
    int i, j;

    for (i = 0; i < NX; i++){
        for (j = 0; j < NJ; j++){
            A[i][j] = ((DATA_TYPE) i * j) / NX;
            B[i][j] = ((DATA_TYPE) i * (j + 1)) / NX;
            C[i][j] = ((DATA_TYPE) i * (j + 3)) / NX;
            D[i][j] = ((DATA_TYPE) i * (j + 2)) / NX;
        }
    }
}

void BEGIN_HYBRID mm31() {
    int i, j, k;
    for (i = 0; i < NX; i++){
        for (j = 0; j < NJ; j++){
            E[i][j] += A[i][j] * B[i][j];
            F[i][j] += C[i][j] * D[i][j];
            G[i][j] += E[i][j] * F[i][j];
        }
    }
    END_HYBRID;
}

int main() {
    init_array();
    RUN_HYBRID mm31;
    mm31();
    STOP_HYBRID;
}
```

Código-fonte 5.4. Multiplicação de Matrizes

Seguindo o fluxo de execução determinado no exemplo anterior, inicialmente foi gerado uma versão para a CPU, sendo que os resultados dos passos de otimização e detecção pode ser vista na Figura 5.4, que mostra a área paralelizada e o código na representação intermediária do LLVM.

Figura 5.4. Região paralelizável detectada na função `mm31`.

O Código 5.5 mostra o perfilamento de uma saída gerada pelo Código 5.4 executado na `CPU`. Podemos perceber neste caso que o programa lançou todas as *threads* possíveis, visto que a quantidade de trabalho é extremamente grande comparado com o primeiro exemplo

aqui demonstrado. Podemos verificar também que a quantidade de informação retornada pelo perfilamento feito pelo *nvprof* é dependente da quantidade de processamento necessário.

```
======== Thread 139798444722112
 81.95%  1.68397s  init_array_.preheader
 81.95%  1.68397s  | init_array
 81.95%  1.68397s  |   main
 18.05%  370.87ms  mm31_WITH_CPU__preheader_polly_subfn
 18.05%  370.87ms  | mm31_WITH_CPU_.preheader
 18.05%  370.87ms  |   mm31_WITH_CPU
 18.05%  370.87ms  |     main

======== Thread 139798383523712
  9.76%  200.47ms  sched_yield
  9.76%  200.47ms  | __kmp_fork_barrier(int, int)
  9.76%  200.47ms  |   __kmp_launch_thread
  9.76%  200.47ms  |     _ZL19__kmp_launch_workerPv
  9.76%  200.47ms  |       start_thread
  9.76%  200.47ms  |         __clone
  8.29%   170.4ms  pthread_cond_wait@@GLIBC_2.3.2
  8.29%   170.4ms  | __kmp_suspend_64
  8.29%   170.4ms  |   __kmp_fork_barrier(int, int)
  8.29%   170.4ms  |     __kmp_launch_thread
  8.29%   170.4ms  |       _ZL19__kmp_launch_workerPv
  8.29%   170.4ms  |         start_thread
  8.29%   170.4ms  |           __clone

======== Thread 139798286756096
  8.78%  180.43ms  sched_yield
  8.78%  180.43ms  | __kmp_fork_barrier(int, int)
  8.78%  180.43ms  |   __kmp_launch_thread
  8.78%  180.43ms  |     _ZL19__kmp_launch_workerPv
  8.78%  180.43ms  |       start_thread
  8.78%  180.43ms  |         __clone
  8.29%   170.4ms  pthread_cond_wait@@GLIBC_2.3.2
  8.29%   170.4ms  | __kmp_suspend_64
  8.29%   170.4ms  |   __kmp_fork_barrier(int, int)
  8.29%   170.4ms  |     __kmp_launch_thread
  8.29%   170.4ms  |       _ZL19__kmp_launch_workerPv
  8.29%   170.4ms  |         start_thread
  8.29%   170.4ms  |           __clone
  0.98%  20.047ms  __kmp_fork_barrier(int, int)
  0.98%  20.047ms  | __kmp_launch_thread
  0.98%  20.047ms  |   _ZL19__kmp_launch_workerPv
  0.98%  20.047ms  |     start_thread
  0.98%  20.047ms  |       __clone

======== Thread 139798375127168
  9.76%  200.47ms  sched_yield
  9.76%  200.47ms  | __kmp_fork_barrier(int, int)
  9.76%  200.47ms  |   __kmp_launch_thread
  9.76%  200.47ms  |     _ZL19__kmp_launch_workerPv
  9.76%  200.47ms  |       start_thread
  9.76%  200.47ms  |         __clone
  8.29%   170.4ms  pthread_cond_wait@@GLIBC_2.3.2
  8.29%   170.4ms  | __kmp_suspend_64
  8.29%   170.4ms  |   __kmp_fork_barrier(int, int)
  8.29%   170.4ms  |     __kmp_launch_thread
  8.29%   170.4ms  |       _ZL19__kmp_launch_workerPv
  8.29%   170.4ms  |         start_thread
  8.29%   170.4ms  |           __clone

======== Thread 139798379325440
  9.76%  200.47ms  sched_yield
  9.76%  200.47ms  | __kmp_fork_barrier(int, int)
  9.76%  200.47ms  |   __kmp_launch_thread
  9.76%  200.47ms  |     _ZL19__kmp_launch_workerPv
  9.76%  200.47ms  |       start_thread
  9.76%  200.47ms  |         __clone
  8.29%   170.4ms  pthread_cond_wait@@GLIBC_2.3.2
  8.29%   170.4ms  | __kmp_suspend_64
  8.29%   170.4ms  |   __kmp_fork_barrier(int, int)
  8.29%   170.4ms  |     __kmp_launch_thread
  8.29%   170.4ms  |       _ZL19__kmp_launch_workerPv
  8.29%   170.4ms  |         start_thread
  8.29%   170.4ms  |           __clone

======== Thread 139798278359552
  9.76%  200.47ms  sched_yield
  9.76%  200.47ms  | __kmp_fork_barrier(int, int)
  9.76%  200.47ms  |   __kmp_launch_thread
  9.76%  200.47ms  |     _ZL19__kmp_launch_workerPv
  9.76%  200.47ms  |       start_thread
  9.76%  200.47ms  |         __clone
  8.29%   170.4ms  pthread_cond_wait@@GLIBC_2.3.2
  8.29%   170.4ms  | __kmp_suspend_64
  8.29%   170.4ms  |   __kmp_fork_barrier(int, int)
  8.29%   170.4ms  |     __kmp_launch_thread
  8.29%   170.4ms  |       _ZL19__kmp_launch_workerPv
  8.29%   170.4ms  |         start_thread
  8.29%   170.4ms  |           __clone

======== Thread 139798282557824
  9.76%  200.47ms  sched_yield
  9.76%  200.47ms  | __kmp_fork_barrier(int, int)
  9.76%  200.47ms  |   __kmp_launch_thread
  9.76%  200.47ms  |     _ZL19__kmp_launch_workerPv
  9.76%  200.47ms  |       start_thread
  9.76%  200.47ms  |         __clone
  8.29%   170.4ms  pthread_cond_wait@@GLIBC_2.3.2
  8.29%   170.4ms  | __kmp_suspend_64
  8.29%   170.4ms  |   __kmp_fork_barrier(int, int)
  8.29%   170.4ms  |     __kmp_launch_thread
  8.29%   170.4ms  |       _ZL19__kmp_launch_workerPv
  8.29%   170.4ms  |         start_thread
  8.29%   170.4ms  |           __clone

======== Thread 139798274161280
  9.76%  200.47ms  sched_yield
  9.76%  200.47ms  | __kmp_fork_barrier(int, int)
  9.76%  200.47ms  |   __kmp_launch_thread
  9.76%  200.47ms  |     _ZL19__kmp_launch_workerPv
  9.76%  200.47ms  |       start_thread
  9.76%  200.47ms  |         __clone
  8.29%   170.4ms  pthread_cond_wait@@GLIBC_2.3.2
  8.29%   170.4ms  | __kmp_suspend_64
  8.29%   170.4ms  |   __kmp_fork_barrier(int, int)
  8.29%   170.4ms  |     __kmp_launch_thread
  8.29%   170.4ms  |       _ZL19__kmp_launch_workerPv
  8.29%   170.4ms  |         start_thread
  8.29%   170.4ms  |           __clone
```

Figura 5.5. Lista de *threads* na `CPU` geradas pela multiplicação de matrizes.

O perfilamento da execução voltada para a `GPU` pode ser visualizada no Código 5.6. Em comparação com o resultado gerado pelo perfilamento do primeiro exemplo, podemos verificar que o tempo de execução entre a `CPU` e a `GPU` é muito mais próxima. A partir deste exemplo, vemos que a quantidade de tempo utilizado para a resolução do problema em sí é bastante superior na `GPU`, representada pelo tempo de execução do `kernel_0`, porém a quantidade de tempo utilizado para a cópia de dados ainda limita a performance da `GPU`, levando em torno de 57% do tempo.

```
==3071== NVPROF is profiling process 3071, command: ./hybrid.exe
==3071== Profiling application: ./hybrid.exe
==3071== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 44.14%  400.20ms         3  133.40ms  105.49ms  164.80ms  [CUDA memcpy DtoH]
 42.29%  383.38ms         7  54.768ms  43.125ms  63.483ms  [CUDA memcpy HtoD]
 13.57%  123.00ms         1  123.00ms  123.00ms  123.00ms  kernel_0

==3071== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 47.69%  524.05ms         3  174.68ms  130.25ms  228.72ms  cuMemcpyDtoH
 34.99%  384.51ms         7  54.930ms  43.283ms  63.685ms  cuMemcpyHtoD
 10.82%  118.91ms         7  16.988ms  269.00us  19.777ms  cuMemFree
  6.22%  68.400ms         1  68.400ms  68.400ms  68.400ms  cuCtxCreate
  0.21%  2.2701ms         7  324.30us  317.80us  334.06us  cuMemAlloc
  0.04%  439.92us         1  439.92us  439.92us  439.92us  cuLinkCreate
  0.01%  93.452us         1  93.452us  93.452us  93.452us  cuLinkComplete
  0.01%  79.812us         1  79.812us  79.812us  79.812us  cuModuleLoadData
  0.01%  73.750us         1  73.750us  73.750us  73.750us  cuLinkAddData
  0.00%  30.116us         1  30.116us  30.116us  30.116us  cuDeviceGetName
  0.00%  17.488us         1  17.488us  17.488us  17.488us  cuLaunchKernel
  0.00%  2.4640us         3     821ns     262ns  1.8220us  cuDeviceGetCount
  0.00%  1.5720us         1  1.5720us  1.5720us  1.5720us  cuLinkDestroy
  0.00%  1.0290us         3     343ns     177ns     553ns  cuDeviceGet
  0.00%  1.0070us         4     251ns     206ns     321ns  cuDeviceGetAttribute
  0.00%     613ns         1     613ns     613ns     613ns  cuModuleGetFunction
  0.00%     305ns         1     305ns     305ns     305ns  cuDeviceComputeCapability
```

Figura 5.6. Lista de *threads* na GPU geradas pela multiplicação de matrizes.

De acordo com estes testes preliminares, em que o *offloading* para o dispositivo e execução da CPU foram forçados pelo compilador, podemos afirmar que a ferramenta desenvolvida consegue produzir código híbrido para o dispositivo acelerador e o processador da arquitetura heterogênea. Visto que o desempenho do código gerado não entra em questão neste momento, não foi produzido um estudo mais aprofundado em relação aos tempos de execução listados pela ferramenta *nvprof*.

O resultado parcial gerado pela execução do código mostrado acima pode ser visto no Código 5.5, sendo esse resultado idêntico tanto para a CPU quanto para a GPU.

```
F[43][5129] = 486.884064
G[43][5129] = 236871.359375
E[43][5130] = 486.694305
F[43][5130] = 487.073853
G[43][5130] = 237056.078125
E[43][5131] = 486.884064
F[43][5131] = 487.263672
```

Código-fonte 5.5. Resultado da multiplicação de matrizes.

5.2. Testes com o Ambiente de Execução

Devido à problemas relacionados à integração da ferramenta de compilação e o ambiente de execução não foi possível completar os testes de coleta de métricas e *offloading* automático.

O principal impedimento enfrentado no momento em que a realização de testes estava sendo produzida foi a incompatibilidade entre o modelo de chamada de funções geradas pela ferramenta e o tipo de função que o ambiente é capaz de interceptar.

Anteriormente ao desenvolvimento da ferramenta, os *scripts* utilizados para a geração de código paralelizável era feito utilizando o compilador `gcc`. Na implementação desta ferramenta porém, foi utilizado a ferramenta `clang`, por se tratar de uma ferramenta do projeto `LLVM`. Até o momento de realização de testes acreditávamos que o formato gerado seria funcional, porém não foi possível executar com a interceptação do *loops* e *chunks* executados na `CPU`.

```
[TRACE]: [roofline.c:0000000841] Thread [140685893798144] in RM_start_counters(): Kind of
    Event Set[1].
[TRACE]: [ hookomp.c:0000000581] Thread [140685897996416] in HOOKOMP_generic_next(): [
    HOOKOMP]: [Before Call]-> Target GOMP_loop_*_next -- istart: 30 iend: 32.
[TRACE]: [ hookomp.c:0000000580] Thread [140685801929600] in HOOKOMP_generic_next(): [
    HOOKOMP]: [OTHERS/WAKE UP] Calling next function out of measures section after wake up.
[RM_papi_handle_error] roofline.c -> RM_start_counters [line 843]: PAPI error -1: Invalid
    argument
[TRACE]: [roofline.c:0000000821] Thread [140685893798144] in RM_start_counters(): [INI]:
    Adding [MEM_UOPS_RETIRED:ALL_STORES].
[TRACE]: [roofline.c:0000000828] Thread [140685893798144] in RM_start_counters(): EventCode
    Reseted:[0].
[TRACE]: [roofline.c:0000000833] Thread [140685893798144] in RM_start_counters(): Event Code
    [40000004]:[MEM_UOPS_RETIRED:ALL_STORES].
```

Código-fonte 5.6. *Log* gerado na tentativa de executar o sistema completo

O Código 5.6 mostra que as *threads* não conseguem ser iteradas e interceptadas corretamente, visto que um erro (`Invalid argument`) de número de argumentos na chamada da função do `PAPI` é encontrado. Isso se trata de mudanças feitas na interface do `PAPI`. Porém o principal problema não é mostrado neste *log* já que o formato das funções geradas impedem que as atividades sejam completadas.

5.3. Problemas de Desenvolvimento

Durante o desenvolvimento do trabalho foram enfrentados diversos problemas que dificultaram consideravelmente o processo de desenvolvimento. Podemos dividir estes problemas em duas categorias: dificuldades logísticas e dificuldades técnicas.

Entre os problemas técnicos, é possível listar:

- **Documentação da ferramenta `PoLLy`**: Embora claramente superior à todas as ferramentas alternativas listadas no Capítulo 2, o Projeto `PoLLy` possui problemas quando se faz necessário utilizar partes separadas de sua implementação, já que a documentação dentro do programa em si é bastante escassa, tornando a leitura do código um processo bastante lento e trabalhoso.

- **Problemas no versionamento das ferramentas utilizadas**: Um problema que foi enfrentado constantemente durante todo o desenvolvimento, foi a necessidade de recompilar bibliotecas com a versão correta do compilador *gcc* ou *clang*, dependendo do caso, tanto para os *drivers* utilizados para a `GPU`, quanto para as ferramentas do projeto `LLVM`. Isso levou a uma série de erros e tempo de depuração até que a versão correta do compilador fosse encontrada e os erros gerados resolvidos.
- **Complexidade da proposta do trabalho desenvolvido**: Como visto pelos problemas de entendimento da ferramenta `PoLLy`, este trabalho se mostrou complexo demais para que o estudo aprofundado da ferramenta e implementação acontecesse dentro do tempo previsto originalmente definido. Dado este problema, o escopo foi modificado em relação aos testes e também foi necessário implementar um pré-processamento, o que não havia sido planejado inicialmente.

Quanto ao problema logístico, a principal dificuldade enfrentada foi a disponibilidade de arquiteturas que davam suporte para os eventos necessários para que as medições do `PAPI` fossem feitas. Diversas máquinas e arquiteturas diferentes foram testadas para suporte aos contadores porém apenas a utilizada neste projeto permitiu o uso de eventos padrões do `PAPI` e os eventos nativos necessários para o funcionamento da biblioteca de captura de dados.

Essas questões dificultaram consideravelmente o desenvolvimento da ferramenta e impediram a conclusão da ultima e mais importante funcionalidade que seria utilizar o ambiente de execução para medição de métricas e testes. Porém, o trabalho aqui demonstrado fica próximo da resolução total do problema e precisaria de algumas modificações estruturais para que ele seja completamente finalizado.

Capítulo 6

Conclusões

Neste trabalho, foi produzido um estudo sobre a área da Ciência da Computação que engloba arquiteturas compostas por dispositivos de arquiteturas heterogêneas (mais especificamente nós computacionais formados por `CPU` e `GPU`). Este trabalho teve como principal motivação a criação de uma ferramenta capaz de fazer a geração de código para um ambiente de execução previamente desenvolvido em (GONçALVES, 2016).

Embora não tenha sido possível atingir todas as metas planejadas, a arquitetura e ferramenta desenvolvida no decorrer deste projeto possui diversas funcionalidades que auxiliam no desenvolvimento para arquiteturas heterogêneas e é parcialmente capaz de coletar métricas de desempenho na execução de código fora do contexto do ambiente de execução.

Portanto, a partir dos objetivos aqui alcançados, podemos concluir que:

- **Possibilidade de explorar arquiteturas heterogêneas é viável a partir de código legado**: Mesmo no escopo desta ferramenta, focada para um ambiente de execução em específico, podemos perceber que é viável alterar aplicações maiores e mais complexas sem que grandes modificações do código fossem necessárias. Se considerarmos que a arquitetura aqui desenvolvida fosse mantida, é possível que apenas pequenas regiões do código fossem anotadas, como a definição de funções e suas chamadas;
- **Ferramenta `PoLLy` ainda é relativamente incompleta para ser utilizada em aplicações comerciais**: Apesar de possibilitar a utilização de partes independentes de sua implementação, as partes finais do `PoLLy` ainda estão incompletas, o que dificulta a geração de código, customização das *threads* e definição escalonamento, visto que foi necessário modificar o código fonte do projeto `PoLLy` e utilizar comandos diretos sobre blocos de código para que a geração de código para a `CPU` fosse que os requisitos do ambiente fossem alcançados;
- **Processamento na `GPU` é extremamente custoso**: Embora *benchmarkings* mais elaborados não tenham sido produzidos, a partir dos exemplos listados como casos

de teste é possível afirmar que o tempo de execução na GPU é extremamente limitado pelo tempo necessário para que a cópia da memória seja feita. Portanto, é interessante estudar diferentes formas de chamar essas funções no caso da ferramenta ser expandida, ou seja, no caso de múltiplas chamadas à funções que serão descarregadas na GPU é necessário otimizar a forma que a transferência é feita.

6.1. Trabalhos Futuros

A partir da ferramenta desenvolvida, é possível que diversas outras funcionalidades sejam criadas e expandidas dentro desta própria arquitetura, seja isso por meio de correção de erros que existem dentro do compilador ou possíveis melhorias sobre o conceito original.

Entre os trabalhos que podem ser feitos a partir do desenvolvimento aqui exemplificado temos:

- **Finalização da integração entre ambiente e ferramenta de compilação**: Antes de qualquer outra modificação da ferramenta seria necessário assegurar a funcionalidade do sistema completo, portanto como prioridade maior para que essa ferramenta cumpra a sua funcionalidade principal seria necessário modificar o *back-end* do ambiente de execução ou fazer as modificações necessárias no código destinado à CPU gerado pela ferramenta, de forma que seja possível utilizar o ambiente para a coleta de métricas e decisão de *offloading*;
- **Expansão da ferramenta para que a necessidade de pré-processamento seja removida**: Uma das melhorias que impactariam esta ferramenta de forma mais significativas seria a remoção da necessidade de anotar determinadas regiões para que a geração de código seja feita. Com essa modificação, seria completamente viável encerrar a necessidade de reescrita de código e tornaria o compilador mais robusto quanto a sintaxe que é capaz de tratar;
- ***Front-end* para diferentes linguagens**: Outra possível melhoria dessa ferramenta é expandir a quantidade de linguagens que podem ser analisadas, o que aumentaria o leque de aplicações de código legado que pudessem utilizar completamente o poder de processamento de arquiteturas heterogêneas.

Referências

BACON, David F.; RABBAH, Rodric; SHUKLA, Sunil. Fpga programming for the masses - the programmability of fpgas must improve if they are to be part of mainstream computing. *Magazine Queue - Mobile Web Development*, v. 11, February 2013.

BRODTKORB, Andre R.; DYKEN, Christopher; HAGEN, Trond R.; HJELMERVIK, Jon M.; STORAASLI, Olaf O. State-of-the-art in heterogeneous computing. *Scientific Programming 18*, 2010.

BRODTKORB, André R.; HAGEN, Trond R.; SæTRA, Martin L. Graphics processing unit (gpu) programming strategies and trends in gpu computing. *J. Parallel Distrib. Comput.*, 2012.

CHAPMAN, Barbara; JOST, Gabriele; PAS, Ruud van der. Using openmp:portable shared memory parallel programming (scientific and engineering computation). *The MIT Press*, 2007.

FLYNN, MICHAEL J. Some computer organizations and their effectiveness. *IEEE TRANSACTIONS ON COMPUTERS*, c-21, n. 9, 1972.

GONçALVES, Rogério Aparecido. *Paralelização Automática de Código em Plataformas Heterogêneas Modernas para HPC*. Tese (Doutorado) — Instituto de Matemática e Estatística (IME), Universidade de São Paulo (USP, São Paulo, 2014.

GONçALVES, Rogério Aparecido. *A Runtime for Code Offloading on Modern Heterogeneous Platforms*. Tese (Doutorado) — Instituto de Matemática e Estatística (IME), Universidade de São Paulo (USP), São Paulo, Dezembro 2016.

GROSSER, TOBIAS; GROESSLINGER, ARMIN; LENGAUER, CHRISTIAN. Polly — performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, v. 22, n. 04, p. 1250010, 2012. Disponível em: <http://www.worldscientific.com/doi/abs/10.1142/S0129626412500107>.

GROSSER, Torsten Hoefler Tobias. Polly-acc -transparent compilation to heterogeneous hardware. *ICS 2016*, 2016.

INTEL. Intel OpenMP Runtime Library Interface. *Technical Report*, 2016. Disponível em: <https://www.openmprtl.org/sites/default/files/resources/libomp_20160322_manual.pdf>.

INTEL. *O Coprocessador Intel® Xeon Phi(tm) 5110p*. 2017. Disponível em: <http://www.intel.com.br/content/www/br/pt/products/processors/xeon-phi/xeon-phi-processors.html>.

KAUER, Mozart Lemos de Siqueira Anderson Uilian. Escalonamento em arquiteturas heterogêneas - apus. *ERAD-RS*, 2013.

KHRONOS. *The open standard for parallel programming of heterogeneous systems*. 2017. Disponível em: <https://www.khronos.org/opencl/>.

LATTNER, Chris; ADVE, Vikram. Llvm: A compilation framework for lifelong program analysis & transformation. *Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)*, 2004.

LIBGOMP, GNU. Gnu offloading and multi processing runtime library: The gnu openmp and openacc implementation. *Technical Report, GNU libgomp*, 2016. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc-5.4.0/libgomp.pdf>.

LLVM, Projeto. *The LLVM Compiler Infrastructure Site*. 2017. Disponível em: <http://llvm.org/>.

MIKUSHIN, Dmitry; LIKHOGRUD, Nikolay; ZHANG, Eddy Z. Kernelgen – the design and implementation of a next generation compiler platform for accelerating numerical models on gpus. *Rutgers Technical Reports*, 2013.

NVIDIA. Nvidia's next generation cuda compute architecture: Kepler gk110. *Relatório técnico*, 2012. Disponível em: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>.

NVIDIA. Whitepaper: Nvidia geforce gtx 750 ti. featuring first-generation maxwell gpu technology, designed for extreme performance per watt. 02 2014. Disponível em: <http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf>.

NVIDIA. *NVIDIA Announces Financial Results for Third Quarter Fiscal 2017*. 2016. Disponível em: <http://nvidianews.nvidia.com/news/nvidia-announces-financial-results-for-third-quarter-fiscal-2017>.

NVIDIA. Whitepaper: Nnvidia geforce gtx 1080 gaming perfected. 2016. Disponível em: <http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf>.

NVIDIA. Whitepaper: Nvidia tesla p100 the most advanced datacenter accelerator ever built featuring pascal gp100, the world's fastest gpu. 2016. Disponível em: <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>.

NVIDIA. *CUDA C PROGRAMMING GUIDE*. 2017. Disponível em: <http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf>.

NVIDIA. Nvidia cuda c programming guide. v. 8, jun. 2017. Disponível em: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html\#axzz4jKf0tngP>.

OPENACC. *Press Release: OpenACC Adoption Continues to Gain Momentum in 2016*. 2016. Disponível em: <http://www.openacc.org/content/press-release-openacc-adoption-continues-gain-momentum-2016>.

OPENACC. *Openacc directives for accelerators*. 2017. Disponível em: <https://www.openacc.org/>.

PAPI. *PAPI Events*. 2019. Disponível em: <https://icl.cs.utk.edu/projects/papi/wiki/PAPIC:Events>.

POUCHET, Louis-Noël; BONDUGULA, Uday; YUKI, Tomofumi. *PolyBench/C the Polyhedral Benchmark suite*. 2012. Disponível em: <http://web.cse.ohio-state.edu/~pouchet/software/polybench/>.

PROJECT, PAPI. *PAPI*. 2019. Disponível em: <https://icl.cs.utk.edu/projects/papi/wiki/PAPIC:Overview>.

SOURCEWARE. *libffi*. 2019. Disponível em: <https://sourceware.org/libffi/>.

STRINGHINI, D.; GONÇALVES, R.; GOLDMAN, A. Capítulo 7: Introdução à computação heterogênea. In: *Luiz Carlos Albini, Alberto Ferreira de Souza, Renata Galante, Roberto Cesar Junior, Aurora Pozo. (Org.). XXXI Jornadas de Atualização em Informática*. Curitiba, Paraná, Brazil: Sociedade Brasileira de Computação (SBC), 2012. v. 21, p. 262–309. Disponível em: <http://www.lbd.dcc.ufmg.br/bdbcomp/servlet/Trabalho?id=12580>.

TEAM, LLVM. *llc(1) - Linux man page*. 2019. Disponível em: <https://linux.die.net/man/1/llc>.

TEAM, LLVM. *LLVM: opt tool*. 2019. Disponível em: <https://releases.llvm.org/1.0/docs/CommandGuide/opt.html>.

TOP500. *Top 500 List June 2019*. 2019. Disponível em: <https://www.top500.org/lists/2016/11/>.

VERDOOLAEGE, Sven; GROSSER, Tobias. Polyhedral extraction tool. In: *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France: [s.n.], 2012.

VERDOOLAEGE, Sven; JUEGA, Juan Carlos; COHEN, Albert; GóMEZ, José Ignacio; TENLLADO, Christian; CATTHOOR, Francky. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, ACM, New York, NY, USA, v. 9, n. 4, p. 54:1–54:23, jan. 2013. ISSN 1544-3566. Disponível em: <http://doi.acm.org/10.1145/2400682.2400713>.