Universidade Federal de Goiás
Regional Catalão
Unidade Acadêmica Especial de Biotecnologia
Curso de Bacharelado em Ciências da Computação

Proposta de um sistema automático de paralelização de código sequencial

Lucas Mesquita Borges

Catalão – GO
2019

Lucas Mesquita Borges

Proposta de um sistema automático de paralelização de código sequencial

Monografia apresentada ao Curso de Bacharelado em Ciências da Computação da Universidade Federal de Goiás – Regional Catalão, como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação. *EXEMPLAR DE DEFESA II*

Orientador: Prof. Dr. Dalton Matsuo Tavares

Catalão – GO
2019

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

Borges, Lucas Mesquita
Proposta de um sistema automático de paralelização de código sequencial [manuscrito] / Lucas Mesquita Borges. – 2019.
52 p.: il.

Orientador: Prof. Dr. Dalton Matsuo Tavares
Monografia (Graduação) – Universidade Federal de Goiás, Unidade Acadêmica Especial de Biotecnologia, Ciências da Computação, 2019.
Bibliografia.

1. Código Sequencial. 2. CUDA. 3. Laravel. 4. OpenCL. 5. Paralelização Automática. I. Tavares, Dalton Matsuo, orient. II. Título.

CDU 004

Lucas Mesquita Borges

Proposta de um sistema automático de paralelização de código sequencial

Monografia apresentada ao curso de Bacharelado em Ciências da Computação da Universidade Federal de Goiás – Regional Catalão.

Trabalho aprovado em 29 de Novembro de 2019.

Dalton Matsuo Tavares
Orientador

Marcos Aurélio Batista
UFG - Regional Catalão

Tércio Alberto dos Santos Filho
UFG - Regional Catalão

Catalão – GO
2019

AGRADECIMENTOS

Com esse espaço gostaria de agradecer à minha esposa Lorraynne, pelo apoio moral e emocional durante todos os quatro anos de graduação, nas horas que me ausentei para estudos e pesquisas. Gostaria de agradecer aos meus pais Silvio e Kelma, que desde minha infância me apoiaram e incentivaram em dar prioridade aos meus estudos. À minha irmã Kellen, que sempre me motivou, apoiou e inspirou a continuar evoluindo pessoalmente e profissionalmente.

Gostaria de agradecer ao meu orientador Dalton, que em todo esse período trilhou e guiou nas pesquisas e escritas, sempre de forma cordial, paciente e profissional. E ao corpo docente do DCC, que contribuíram e contribuem para a formação de cientistas da computação.

“Viva como se você fosse morrer amanhã.
Estude como se você fosse viver para sempre.”
(Mahatma Gandhi)

RESUMO

BORGES, L. M.. **Proposta de um sistema automático de paralelização de código sequencial**. 2019. 52 p. Monografia (Graduação) – Unidade Acadêmica Especial de Biotecnologia, Universidade Federal de Goiás – Regional Catalão, Catalão – GO.

Códigos sequenciais com grande quantidade de tarefas repetitivas podem demandar um longo tempo de execução. Por esse motivo, é necessário utilizar técnicas e métodos para otimizar o tempo de execução de algoritmos, sendo um exemplo a paralelização de laços. Com a intenção de disponibilizar um sistema automático de paralelização de códigos sequenciais que otimize algoritmos com alta complexidade de execução, realizou-se uma revisão de escopo e estudo de caso envolvendo paralelização de algoritmos. Com isso, foram buscados paralelizadores automáticos de códigos sequenciais, os quais podem possuir execução em processadores ou placas gráficas. Durante a revisão de escopo foram encontradas algumas ferramentas de paralelização de código sequencial. Com base nos dados obtidos no material de pesquisa da revisão de escopo, foi realizado uma comparação de performance entre os paralelizadores com o intuito de selecionar o que apresentasse melhores resultados de otimização do código paralelo gerado. Após comparação dos métodos, foi selecionado o paralelizador PPCG com diretivas OpenACC, o qual realiza a paralelização de códigos C em códigos CUDA e OpenCL. O *framework* PHP Laravel foi utilizado para disponibilizar a ferramenta de paralelização de forma *online* com acesso multiusuário.

Palavras-chave: Código Sequencial, CUDA, Laravel, OpenCL, Paralelização Automática.

ABSTRACT

BORGES, L. M.. **Proposta de um sistema automático de paralelização de código sequencial**. 2019. 52 p. Monografia (Graduação) – Unidade Acadêmica Especial de Biotecnologia, Universidade Federal de Goiás – Regional Catalão, Catalão – GO.

Sequential codes with a large number of repetitive tasks can take a long time to execute. Therefore, it is necessary to use techniques and methods to optimize the execution time of algorithms, using for example, the parallelization of loops. In order to provide an automatic sequential code parallelization system that optimizes algorithms with high execution complexity, a scoping review and a case study involving parallelization of algorithms was performed. As a result, automatic sequencing code parallelisers were sought, which may execute on processors or graphics cards. During the scoping review some sequential code parallelization tools were found. Based on the data gathered during the scoping review, a performance comparison was performed among the parallelisers in order to select the one that presented the best optimization results of the generated parallel code. After comparing the methods, the PPCG parallelizer with OpenACC directives was selected, which performs parallelization of C code into CUDA and OpenCL codes. The PHP Laravel framework was used to provide an online parallelization tool with multiuser access.

Keywords: Automatic Parallelization, CUDA, Laravel, OpenCL, Polyhedral, Sequential Code.

LISTA DE ILUSTRAÇÕES

Figura 1 – Diretivas OpenACC anotadas pelo DawnCC 36
Figura 2 – Primeiro teste execução código CPU x código GPU 41
Figura 3 – Segundo teste execução código CPU x GPU 42
Figura 4 – Paleta de blocos 43
Figura 5 – Algoritmo de ordenação bolha programado em blocos 44
Figura 6 – Código gerado do algoritmo de ordenação bolha 45
Figura 7 – Arquitetura de Paralelização 46
Figura 8 – Saída de execução na interface do sistema 47

LISTA DE TABELAS

Tabela 1 – Comparativo entre os paralelizadores identificados 34

SUMÁRIO

1 INTRODUÇÃO . . . 19
1.1 Objetivos . . . 20

2 METODOLOGIA . . . 21
2.1 Revisão de Escopo . . . 21
2.1.1 Paralelizadores Automáticos . . . 22
2.2 Considerações finais . . . 23

3 PARALELIZAÇÃO AUTOMÁTICA E OTIMIZAÇÃO DE CÓDIGO . . . 25
3.1 Apresentação dos Principais Métodos de Paralelização Automática 26
3.1.1 Graphite (VIEIRA, 2010) . . . 26
3.1.2 PPCG (VERDOOLAEGE et al., 2013) . . . 27
3.1.3 PGCC com diretivas OpenACC (MOREIRA, 2015) . . . 28
3.1.4 Aeminum (GONçALVES, 2013) . . . 29
3.1.5 PLUTO e CLooG (DI et al., 2012) . . . 29
3.1.6 Java JIT e JikesRVM (Leung, Alan Chun Wai, 2008) . . . 30
3.1.7 ChiLL (KHAN et al., 2013) . . . 31
3.1.8 Cetus (Dave et al., 2009) . . . 32
3.1.9 PENCIL (Baghdadi et al., 2015) . . . 32
3.1.10 PIPS e Par4All (MINI et al., 2011) . . . 33
3.2 Comparativo entre os paralelizadores identificados . . . 33
3.3 Otimização de código . . . 35
3.4 Considerações Finais . . . 36

4 PROPOSTA DO SISTEMA . . . 37
4.1 Características do Sistema . . . 37
4.1.1 Arquitetura MVC e padrões de desenvolvimento Web . . . 38
4.1.2 Framework Laravel . . . 39
4.2 Considerações Finais . . . 40

5 RESULTADOS OBTIDOS . . . 41
5.1 Implementação da interface do sistema . . . 42
5.2 Implementação da arquitetura do sistema . . . 42
5.3 Considerações Finais . . . 44

6 CONCLUSÕES 49

REFERÊNCIAS 51

CAPÍTULO

1

INTRODUÇÃO

Esta monografia se insere no contexto da abordagem integrada de Alshuwaikhat e Abubakar (2008) para alcance de universidades sustentáveis, o qual consiste em alcançar universidades sustentáveis por meio de vários conceitos diferentes relacionados às melhorias de gerência da própria universidade. Como exemplo pode-se citar a participação da comunidade em serviços comunitários e justiça social, seminários e cursos de sustentabilidade, além de pesquisa e desenvolvimento relacionados à energia renovável. É abordado a aplicação desses conceitos de várias formas, por exemplo, em redução de desperdícios de recursos e redução da poluição, promover consciência ambiental entre os empregados das universidades e a comunidade relacionada, minimizar impactos negativos das operações, diminuição do lixo gerado através da reciclagem e eficiência energética, dentre outras formas de alcançar maior sustentabilidade, redução de custos e preservação do meio ambiente.

Para aplicar o conceito proposto por Alshuwaikhat e Abubakar (2008) na computação, é necessário utilizar a definição de computação verde, a qual tenta reduzir os efeitos nocivos ao meio ambiente gerados pela computação. Observa-se que o uso de computadores gera o consumo de grandes quantidades de energia, aumentando a emissão de gás carbônico. Isso gera problemas ambientais resultantes da produção e armazenamento do hardware (LUNARDI; SIMõES; FRIO, 2014a).

A computação verde pode ser alcançada por diversos métodos, seja por economia de energia por meio de eficiência energética, gerenciamento da energia utilizada, otimização no design e localização de *data centers*, virtualização de servidores, reciclagem de materiais eletrônicos, geração de energia renovável, dentre outros. No tocante a eficiência energética, pode-se cogitar a redução no tempo de processamento de uma determinada tarefa, a qual consumiria um intervalo de tempo mais longo, de modo a reduzir o consumo energético provocado por esta. Para isso, é necessário buscar métodos e/ou ferramentas que permitam a diminuição do tempo de execução de um programa (LUNARDI; SIMõES; FRIO, 2014a).

Este trabalho estabelece sua importância exatamente no estudo de ferramentas que proporcionem a conversão de software ou algoritmos desenvolvidos de maneira usual (i.e. código sequencial) para códigos paralelos por meio do princípio de paralelização automática. O recurso disponível (i.e. o computador paralelo) será compartilhado com quaisquer atividades de pesquisa e desenvolvimento da própria universidade em que o tempo de espera computacional seja longo, tendo o propósito de acelerar o processamento de atividades repetitivas, antes realizadas sequencialmente, e com isso, reduzir o consumo de energia.

Segundo Lunardi, Simões e Frio (2014b), existem, em suma, dois tipos gerais de processamento paralelo: em uma Unidade Central de Processamento (CPU), e em Unidade de Processamento Gráfico (GPU). O processamento paralelo em GPUs tornou-se mais eficiente que o processamento paralelo em CPUs, graças à alta demanda por processamento gráfico exigido deste hardware e o tratamento de tarefas mediante a paralelização massiva destas, utilizando para tal uma grande quantidade de núcleos de GPU (milhares). Além disso, a evolução das CPUs está limitada ao incremento unitário do número de núcleos (em média 8), devido a fatores físicos e consumo de energia. É importante notar que o processamento paralelo em GPUs possui um custo significativamente menor e uma maior eficiência para paralelização massiva de tarefas, quando comparado a uma configuração de CPU equivalente (LUNARDI; SIMõES; FRIO, 2014b).

Assim sendo, serão destacados por meio de uma revisão de escopo, (discutido no Capítulo 2), os principais métodos de paralelização automática de código sequencial, os quais podem fazer uso de geradores de código e execução paralela em GPUs (LEVAC; COLQUHOUN; O'BRIEN, 2010).

No Capítulo 2 será descrito a metodologia de elaboração desta monografia. No Capítulo 3 serão apresentados os paralelizadores automáticos, será realizado uma comparação entre os mesmos e também serão apresentados otimizadores de código. No Capítulo 4 serão apresentadas informações referentes ao sistema de paralelização, sua arquitetura e interface. No Capítulo 6 serão apresentadas as considerações finais referentes a esta monografia.

1.1 Objetivos

- Selecionar um paralelizador automático
- Utilizar o paralelizador selecionado para implementar um sistema de paralelização automática de código sequencial
- Implementar o sistema automático em uma arquitetura em que o mesmo possa ser disponibilizado para os membros da Universidade
- Implementar o sistema de forma que o mesmo possua uma interface amigável a usuários iniciantes e experientes.

CAPÍTULO

2

METODOLOGIA

Neste capítulo será apresentado a metodologia de elaboração desta monografia, baseada no método de revisão de escopo de Levac, Colquhoun e O'Brien (2010). Por meio desta metodologia foram buscados e analisados os paralelizadores de código sequencial (subseção 2.1.1).

2.1 Revisão de Escopo

Para pesquisa e escrita dessa monografia foi utilizado o processo de revisão de escopo que consiste em um processo de mapeamento e sumarização de fontes relacionadas a um assunto de forma aprofundada, com a melhor eficácia possível, os quais estejam relacionados a hipótese e as questões de pesquisa, e deles extrair o conteúdo necessário para uma pesquisa científica. Isso proporciona a comparação de informações de uma fonte com as informações de outras fontes buscadas (LEVAC; COLQUHOUN; O'BRIEN, 2010).

A revisão de escopo é dividida em cinco passos, sendo:

(i) **Definir a hipótese de pesquisa**: a fim de refutá-la ou validá-la durante a execução da pesquisa;

(ii) **Definir as questões de pesquisa**: neste passo são identificadas questões de pesquisa a fim de facilitar a identificação de trabalhos de pesquisa correlatos a hipótese de pesquisa escolhida;

(iii) **Identificar estudos relevantes**: nesse estágio da revisão de escopo é realizada a identificação de estudos importantes com base nas questões de pesquisa formuladas no passo anterior. Assim, é importante desenvolver um plano de decisão, definindo quais fontes serão utilizadas (ex. quais motores de busca serão utilizados), quais os termos de busca são mais apropriados, o intervalo de tempo de publicação de artigos assumido como aceitável (ex. artigos publicados nos últimos 10 anos) e quais idiomas serão considerados nas buscas;

(iv) **Seleção de estudos**: nesta etapa os artigos buscados são selecionados ou excluídos de acordo com o idioma no qual estão escritos (primeiro critério de exclusão), com a adequação ao período no qual foram publicados (segundo critério de exclusão), com a adequação do título dos artigos ao tema/hipótese de pesquisa (terceiro critério de exclusão) e com a análise de resumo quanto a aderência a uma ou mais questões de pesquisa (quarto critério de exclusão);

(v) **Extração de dados**: nesta etapa é realizada a extração de dados dos artigos selecionados na etapa (iv). O material é extraído de acordo com a relevância relacionada com as questões de pesquisa e com algum método analítico que se deseje utilizar;

(vi) **Sumarização e relatório dos dados**: nesta etapa é realizada uma comparação entre cada um dos resultados extraídos com o intuito de obter a melhor informação a partir de cada uma das fontes relacionadas.

2.1.1 Paralelizadores Automáticos

Durante a revisão de escopo relacionada aos paralelizadores automáticos, foram formuladas as seguintes questões de pesquisa:

1. A aplicação da paralelização automática de código pode substituir a utilização de programas sequenciais?

2. Quais são os tipos de códigos sequenciais que podem ser paralelizados automaticamente?

3. Quais linguagens de programação (ou derivadas) são usadas?

Utilizando estas perguntas como base, foram selecionados artigos relacionados a paralelização automática. Os resultados buscados a partir desse método foram submetidos aos critérios de exclusão, conforme definidos no Capítulo 2 item (iv). Os artigos selecionados após essa triagem inicial foram utilizados na etapa de extração de dados.

Durante a extração de dados foram levantadas as características mais relevantes referentes ao algoritmo de paralelização de cada artigo. Os temas abordados pelos artigos selecionados permitiram averiguar, entre outros, o desempenho de cada método, o conceito de paralelização automática, princípios de tratamento de código e comparativos de *benchmarks* entre o método do artigo avaliado e métodos similares.

De um total de **54.039** artigos buscados em inglês e português, **41.775** artigos se enquadraram no filtro de data (2009-2019). Em seguida, foi realizada a filtragem por adequação de títulos, a qual resultou em **66** artigos selecionados. Finalmente, após a análise de resumo de cada um dos artigos, foram selecionados **16** artigos para a etapa de extração de dados.

2.2 Considerações finais

Neste capítulo foi apresentado a metodologia de revisão de escopo proposta por Levac, Colquhoun e O'Brien (2010), utilizada para elaboração desta monografia. No Capítulo 3 será apresentado a revisão de escopo referente a paralelização automática.

CAPÍTULO

3

PARALELIZAÇÃO AUTOMÁTICA E OTIMIZAÇÃO DE CÓDIGO

Desenvolver um programa de forma paralela é relativamente mais complexo do que o processo de desenvolvimento de um programa sequencial. As ferramentas de paralelização automática proporcionam a conversão do código sequencial para código paralelo, reduzindo o tempo de desenvolvimento e proporcionando uma menor quantidade de erros de desenvolvimento e de execução do sistema. Além disso, a paralelização automática de códigos sequenciais proporciona a otimização de código, que permite atingir ganhos significativos de eficiência energética, devido a redução em tempo de processamento em algoritmos que tenham um elevado tempo de execução resultante de um grande número de iterações em laços (Gonçalves *et al.*, 2014).

Neste trabalho será abordada a eficiência de paralelizadores baseados na paralelização de laços, pois os laços são iterados diversas vezes em programas, são considerados grandes responsáveis pelo aumento de tempo de execução e, ocasionalmente, apresentam alta granularidade de paralelismo (i.e. possibilidade de paralelização). Um código com alta granularidade de paralelismo é um código com baixa quantidade de dependências entre variáveis e que demanda grande tempo de processamento devido a execução em laços. Assim, antes da paralelização de fato, é realizado um processo de análise do código que elimina as falsas dependências entre variáveis e separa os blocos paralelizáveis entre as dependências verdadeiras. Dessa forma, cada bloco será executado em paralelo com o outro (Gonçalves *et al.*, 2014).

Um processo de paralelização que vem sendo aplicado fortemente é a aplicação através do conceito de polítopos, o qual é realizado através de poliedros. Estes são baseados no conceito de hiperplanos, onde um hiperplano é uma dimensão de tamanho n em um "universo" onde existem vários hiperplanos, e um poliedro é a intersecção destes hiperplanos. Também é utilizado neste e em outros compiladores o conceito de polítopos, o qual representa um poliedro de dimensões e tamanhos limitados (DI *et al.*, 2012).

O conceito de polítopos é aplicado durante o fluxo de iteração de laços, onde cada instância de uma variável durante a execução de um determinado laço é um ponto diferente no polítopo, e cada ponto deste polítopo é uma *thread* da GPU (DI *et al.*, 2012). É importante destacar que o modelo de poliedros tem sido a base para os maiores avanços na paralelização automática e paralelização de programas (VERDOOLAEGE *et al.*, 2013).

Devido ao custo de transferência de dados entre CPU-GPU o processo de paralelização automática em GPU's se beneficia exponencialmente quando comparado a quantidade de iterações do laço e dimensão, ou seja, quanto mais vezes um laço for executado, maior será o ganho com paralelismo em GPU. Sendo assim, em laços com poucas iterações a execução sequencial é mais vantajosa. É importante ressaltar que cada interação do laço deve ser independente da interação anterior para que este possa ser executado em paralelo (Leung, Alan Chun Wai, 2008).

3.1 Apresentação dos Principais Métodos de Paralelização Automática

Nesta seção serão abordados os métodos de paralelização automática encontrados nos artigos selecionados durante a revisão de escopo. Sendo eles o **Graphite** (tradução de linguagem C para código C paralelo usando *threads*), o **PPCG com diretivas OpenACC** (tradução de linguagem C para CUDA), o **Aeminum** (tradução para linguagem Java executado usando *threads*), o **PLUTO** e o **CLooG** (tradução de linguagem C para CUDA), o **Java JIT** (Java em GPU através de uma máquina virtual escrita para esse tipo de operação), o **PPCG** (tradução de linguagem C para CUDA), o **ChiLL** (tradução de linguagem C para CUDA), o **Cetus** (tradução de linguagem C para código C paralelo usando *threads*), o **PENCIL** (tradução de linguagem PENCIL para CUDA e OpenCL) e o **PIPS** juntamente com o **Par4All** (tradução de linguagem C para CUDA e OpenCL).

3.1.1 Graphite (VIEIRA, 2010)

O Graphite é uma ferramenta que realiza a paralelização de laços para aplicação em arquiteturas *multicore* (i.e. múltiplas CPUs). Os laços são representados através de polítopos. Posteriormente, aplica-se um conjunto de transformações gerando uma nova estrutura de laço com código mais compacto, com menor consumo de memória. Vale observar que a possibilidade de paralelismo somente é detectada em laços perfeitamente aninhados.

O Graphite é um *framework* de alto nível que é utilizado com o compilador *GNU Compiler Collection* (GCC), o qual apresenta possibilidade de paralelização através da eliminação de falsas dependências e co-dependências de variáveis. Existem dois tipos gerais de dependências de variáveis: as dependências verdadeiras e as falsas dependências. As dependências verdadeiras são inerentes a programação e não podem ser evitadas, o que prejudica o tempo de

execução de um código, pois dois trechos dependentes não podem ser executados em paralelo. As falsas dependências são ocasionadas por reuso de variáveis ou fluxo de iteração do código e, consequentemente, podem ser eliminadas, aumentando a granularidade de paralelismo.

É necessária a comunicação do Graphite com o GCC, a qual é realizada via a representação SSA GIMPLE, como linguagem de interface. SSA ou *Single Static Assignment*, é atribuição estática única e, segundo esta representação do programa, cada variável declarada possui valor atribuído exatamente uma única vez. Múltiplas atribuições para uma mesma variável no código fonte são reescritas como atribuições para variáveis derivadas a partir da original e independente. Por exemplo, para as atribuições a = 2; a = b * a; poderíamos ter a derivação a.0 = 2; a.1 = b * a.0;. Dessa forma, os módulos de tradução constroem uma representação em polítopos a partir do GIMPLE e, a partir deles, reconstroem o programa em GIMPLE.

O GIMPLE realiza uma correção de dependências de dados e, por meio de aprendizado de máquina, determina qual parte do código obterá maior ganho a partir de otimizações usando paralelização automática. Após essa análise, são necessárias duas etapas para organizar o aninhamento de laços. A primeira etapa, permite que o laço mais interno possa ser transferido para um nível mais externo sem alterar o fluxo de execução do programa. A segunda etapa permite organizar o laço mais interno para levar consigo todas as dependências do laço quando deslocado à posição mais externa.

Nem todos os tipos de códigos podem ser paralelizados utilizando-se esse método. Devido a restrição de aplicação, a paralelização somente é possível em laços perfeitamente aninhados. Observou-se que usando este método algumas aplicações ganharam de 0,02% a 11% de desempenho.

3.1.2 PPCG (VERDOOLAEGE et al., 2013)

O compilador PPCG (Polyhedral Parallel Code Generator) é um compilador de código C para CUDA e OpenCL, executado através de técnicas de implementação em poliedros. Este compilador realiza a otimização de laços adaptados a muitos níveis de paralelismo e da hierarquia de memória dos aceleradores da GPU. Também é apresentada geração de código relacionado a implementação com registradores e memória compartilhada, tirando vantagem do uso de *threads* internas, garantindo-se a consistência da alocação da memória local mesmo na presença de múltiplas referências de um laço de um determinado vetor.

Em um dispositivo CUDA, todas as *threads* de um bloco e os blocos em um *grid* são executados em paralelo. Porém, é necessário encontrar dois níveis de paralelismo, o que pode ser obtido aplicando-se uma técnica chamada *tiling*, capaz de otimizar laços aninhados. Essa técnica também é capaz de reduzir a quantidade de memória utilizada dentro de um bloco, sendo capaz de explorar melhor recursos de memória local a cada bloco.

O primeiro passo da paralelização de laços é extrair um modelo poliédrico. Esse modelo

consiste principalmente em iteração de domínio, relações de acesso e escalonamento, sendo cada um destes descritos por constantes. O domínio de iteração irá possuir instâncias das variáveis da entrada do programa, sendo cada instância identificada pelo valor de uma variável em cada iteração do laço. As relações de acesso são definidas dentro do domínio de iteração.

O escalonamento especifica as ordens em que cada instância do laço será executada. Uma tupla de inteiros é associada a cada iteração do laço e essas iterações são executadas de acordo com a ordem das tuplas.

O próximo passo é a verificação da granularidade de paralelização dos laços através da técnica de *tiling*, mapeando os laços para cada *thread* respectiva a um bloco de *threads*. Em geral, é necessário reordenar as execuções do laço para que seja melhor aproveitado o paralelismo. Por exemplo, em laços internos a outros laços, quando não existem dependências, o fluxo de execução pode ser alterado de forma que os laços internos sejam executados paralelamente.

Por fim, é determinado onde serão armazenados os dados resultantes da execução, tendo em vista que CUDA possui várias hierarquias de memória global, memória compartilhada e registradores. Dessa forma, os dados serão armazenados de acordo com a afinidade entre cada uma das variáveis do programa.

Os ganhos de tempo de execução de paralelização automática utilizando o compilador PPCG se dão em casos onde o número de iterações de um laço ou um grupo de laços é grande. Essa solução apresentou perda de desempenho em casos onde existem laços com pouca iteração, assim como ocorre em outros tipos de compiladores paralelos, os quais também apresentam esse problema. O compilador PPCG apresentou na maioria dos resultados um aumento de desempenho de até 10x em relação a execução sequencial.

3.1.3 PGCC com diretivas OpenACC (MOREIRA, 2015)

Existem compiladores que são caracterizados por serem compiladores de código anotado. Isso significa que em cada bloco de código, ocorre uma anotação em sequência que caracteriza a forma que o respectivo bloco será executado.

O compilador PGCC é um compilador de código anotado fechado que utiliza anotações OpenACC. Essas anotações possuem suporte às linguagens C e C++ e definem qual parte do código é executada na CPU ou na GPU de forma paralela, simplificando a análise do código sequencial.

Nesse processo a paralelização automática é dividida em algumas etapas. Primeiramente, todo o código é analisado, dividido em blocos e então são inseridas anotações OpenACC correspondentes sobre quais blocos podem ser executados em paralelo. Na segunda etapa, por meio de estimativas, analisa-se quais desses blocos irão ter o desempenho aumentado caso sejam paralelizados. Através do compilador PGCC traduz-se o código em C com diretivas OpenACC para a linguagem CUDA e OpenCL.

Os resultados desse método de paralelização foram no melhor caso, em multiplicação de matrizes tridimensionais, onde na CPU gastou-se 293,51 segundos em tempo de execução e 2,42 segundos na GPU. No pior caso, em um *benchmark* com problemas resolvidos sequencialmente, gastou-se 22,42 segundos na CPU e 251,83 segundos na GPU. O segundo programa possui um grande volume de reuso de dados, fato que demanda maior quantidade de cópias de dados entre CPU x GPU e não aproveita a capacidade paralela do chip gráfico, fato que provocou um aumento no tempo de execução quando este foi paralelizado. Dessa forma, demonstra-se que programas com dependência linear não são otimizados quando executados em paralelo. Contudo, no primeiro exemplo, em que não há alta dependência linear, e existe alta granularidade de paralelismo, o tempo de execução foi reduzido em 121 vezes.

3.1.4 Aeminum (GONçALVES, 2013)

O compilador Aeminum foi criado com a intenção de ser paralelo por omissão. Nesse caso, um programa Java sequencial é paralelizado automaticamente através do compilador Aeminum. O código a ser paralelizado deve ser dividido por tarefas, onde cada tarefa a ser executada pelo programa deverá ser especificada, pois esse método de paralelização automática não realiza análise de dependências.

Para melhorar o tempo de execução e auxiliar no escalonamento do compilador, diretivas relacionadas ao modo execução devem ser informadas por meio de código, por exemplo, diretivas para informar se existem tarefas dependentes de uma outra tarefa, se existem tarefas filhas, se existem laços, dentre outras informações. Essas dicas são necessárias para que a performance seja melhorada e o tempo de execução não seja pior que o algoritmo sequencial.

O código paralelizado automaticamente através do compilador Aeminum apresenta performance inferior a um código paralelo escrito por um programador, pelo fato de não realizar análise de dependências e não ser executado nenhum algoritmo de otimização. Em alguns casos, o código sequencial apresentou performance até 30 vezes maior que o código paralelizado, talvez pelo fato do compilador Aeminum ser relativamente novo, e de não realizar análise e tratamento de dependências.

3.1.5 PLUTO e CLooG (DI et al., 2012)

O PLUTO é um *framework* de paralelização automática de linguagem C para CUDA. Nele é realizado uma otimização do código e é utilizado o CLooG para geração de código durante a conversão C $\rightarrow$ CUDA.

Várias etapas são executadas nesse *framework* para a paralelização: primeiramente o código sequencial é analisado e a partir dele, é construída uma árvore abstrata de sintaxe (*Abstract Syntax Tree* ou AST). A partir desta árvore são extraídas as funções de acessos dos vetores e o espaço de execução do polítopo referente ao fluxo do programa. Após a análise de todo o

escopo do programa, é feita a análise de dependências de código e dependências internas do polítopo. Após o processo de extração de dependências reais de paralelização, é realizada uma nova ordenação das instâncias das variáveis, com foco no processamento paralelo. Em seguida, são gerados hiperplanos com base nas instâncias ordenadas de acordo com o fluxo de execução do programa paralelo. Por fim, os polítopos de instâncias das variáveis são repassadas para um gerador de código embasado em polítopos. Nesse caso, foi utilizado o CLooG gerando-se então código CUDA.

Em Di *et al.* (2012), este sistema de transformação de código foi comparado com outros métodos de paralelização, principalmente com o método manual. Com relação a escala de GFLOPS, que determina a capacidade de operações de ponto flutuante por segundo. Nesse caso, o método manual apresenta cerca de 75 GFLOPS enquanto que, por intermédio desse *framework* automático, foi possível alcançar valores em torno de 62 GFLOPS. Em todos os casos o código automatizado por CUDA foi em média 5x mais rápido que a execução sequencial.

3.1.6 Java JIT e JikesRVM (Leung, Alan Chun Wai, 2008)

Esta implementação foi desenvolvida sobre uma máquina virtual JikesRVM (*Jikes Research Virtual Machine*). O JikesRVM oferece um ambiente de testes flexível e aberto para tecnologias de máquinas virtuais e proporciona a experimentação com uma grande variedade de alternativas para projeto. Por exemplo, diferente de muitas tecnologias de VMs, o JikesRVM nunca interpreta bytecode do Java. Ao invés disso, ele depende somente de compilação *Just-in-Time* (JIT). O JikesRVM possui dois compiladores diferentes: um base (*baseline*) e um otimizado (*optimizing*). O compilador base oferece um compilador de *bytecode* para código de máquina compilado que é rápido, para a compilação inicial da maioria dos métodos Java. O compilador otimizado, por sua vez, oferece um compilador com otimização em múltiplos níveis, o qual requer maior tempo de compilação, mas que é mais adequado para recompilar métodos executados com frequência. O JikesRVM suporta as arquiteturas x86 e PowerPC, com suporte para arquitetura de 64 bits em fase de desenvolvimento.

Dessa forma, pode-se concluir que a paralelização em GPU foi realizada com o compilador JIT existente, via JikesRVM. Para minimizar a sobrecarga de paralelização, o compilador precisou se concentrar em métodos frequentemente executados do programa. Por meio do sistema de otimização adaptativa do JikesRVM, otimizações em níveis mais altos foram aplicadas somente a métodos que foram identificados como de uso frequente. A paralelização de GPU é feita no nível mais alto de otimização e apenas em código que possui expectativa de ser executado frequentemente.

O JikesRVM pode ser utilizado com a finalidade de realizar a detecção de granularidade de laços, ou seja, se o código contiver laços paralelizáveis, este será executado na GPU caso seja vantajoso em relação ao tempo de execução. Sendo permitido à VM a geração de código para GPU quando apropriado.

A implementação de um paralelizador automático em uma VM pode gerar algumas vantagens, como o conhecimento da infraestrutura do sistema e uma maior fonte de dados sobre o tempo de execução de cada método do código. O processo de paralelização automática é divido em 3 etapas:

- São buscadas informações sobre acessos perdidos a vetores multidimensionais, os quais estão perdidos nos *bytecodes*;
- Realiza análise de dependências nos acessos dos vetores e então constrói-se um gráfico de dependências e;
- Paralelização de fato e geração do código que será executado na GPU.

O compilador paralelizador não busca apenas quais laços são passíveis de paralelização, mas também analisa quais laços valem a pena serem paralelizados. Destes, se analisam duas formas de execução paralela, sendo a primeira executar um laço na CPU e implementar na GPU de forma implícita apenas uma parte do processamento do laço a cada iteração, ou, se implementar toda a execução do laço em paralelo de forma explícita.

Nesse método de paralelização, em 83% dos testes a velocidade de execução em CPU foi mais rápida que em GPU. Somente em um *benchmark* a execução em GPU foi melhor. Neste, a execução foi 13x mais rápida do que no código sequencial.

3.1.7 ChiLL (KHAN et al., 2013)

O compilador ChiLL realiza a conversão de códigos C em códigos CUDA e, após essa conversão, realiza automaticamente auto-otimização que é de grande importância quanto a performance de execução. Isso ocorre porque códigos executados em GPU têm eficácia diminuída devido à pouca otimização. A possibilidade de explorar diferentes alternativas de alocação de memória e mapeamento de blocos de *threads* junto com outras decisões de geração de código facilita encontrar uma solução de alto desempenho.

A meta de construir um compilador auto-otimizador que pode avaliar múltiplas variações de código e gerar esse código sistematicamente fez com que o compilador ChiLL fosse produzido de uma forma diferente dos outros compiladores. O *framework* do compilador é organizado em diferentes camadas de abstração. É importante ressaltar que um usuário do sistema pode operar em qualquer uma dessas camadas.

O processo de paralelização automática é dividido em várias tarefas. Primeiramente são gerados múltiplos códigos CUDA a partir de código C. Em seguida, esses códigos vão sendo analisados e selecionados de acordo com sua performance. Uma camada intermediária chamada CUDA-CHiLL recebe esses códigos gerados como *scripts* de código Lua e gera comandos para a próxima camada. Na camada mais inferior é executada a transformação de laços através da

implementação em poliedros e geração de código através do *framework* CHiLL, o qual realiza as transformações iniciais de código, e é acessado através de *scripts* de mais baixo nível do que na camada intermediária.

3.1.8 Cetus (Dave et al., 2009)

A ferramenta Cetus provê uma infraestrutura para pesquisa em compiladores *multicore*, os quais têm como foco a paralelização automática. A infraestrutura do compilador verifica programas em C e realiza a transformação de código C sequencial para código C paralelo executado em CPU, de forma orientada ao usuário e contém os passos mais importantes a serem seguidos numa paralelização.

A estrutura de dados do Cetus foi implementada na forma de uma hierarquia de classe em Java. Uma representação em alto nível provê uma visão sintática sobre o seu funcionamento, facilitando o entendimento.

O compilador Cetus realiza a paralelização automática através de análise de dependências, definindo acesso privado a vetores e escalares e substituição de variáveis. Este compilador apresenta performance semelhante a outros paralelizadores de execução em CPU como o Compiler Infrastructure (COINS) e o Intel C Compiler (ICC).

3.1.9 PENCIL (Baghdadi et al., 2015)

O *framework* PENCIL é uma plataforma que faz a conversão de linguagem PENCIL para OpenCL ou CUDA. A proposta desse *framework* é a implementação de código na linguagem PENCIL, a qual é baseado e bem semelhante a linguagem C, com conjuntos de regras de código relacionados à maneira que ponteiros podem ser manipulados.

As regras do PENCIL foram estruturadas de forma a se possibilitar uma melhor otimização e paralelização ao se realizar a tradução do código PENCIL para um código de baixo nível.

Algumas regras da linguagem PENCIL são:

- **Restrições de Ponteiro:** declarações de ponteiro e definições são permitidas, mas a manipulação é restrita, exceto quando se trata de uma referência de um vetor, a qual é permitida em argumentos de funções.

- **Recursão:** chamadas recursivas de funções não são permitidas, similar a linguagens como OpenCL.

- **Vetores estáticos:** vetores devem ser declarados de forma estática.

- **Laços *for* estruturados:** um laço *for* em PENCIL deve ter uma única variável de iteração. Valores de início, parada e incremento são fixos. Onde fixo, nesse sentido, significa que este valor não muda no corpo do laço.

PENCIL também suporta funções escalares de OpenCL como *abs, min, max, sin, cos*. O tipo de compilação utilizado é baseado em poliedros. O compilador é uma adaptação do compilador paralelo PPCG, vide subseção 3.1.2, para suportar o código PENCIL.

3.1.10 PIPS e Par4All (MINI et al., 2011)

O PIPS é um compilador paralelizador que vem sendo desenvolvido há 23 anos e inicialmente realizava a paralelização de Fortran usando poliedros, com abordagem baseada em álgebra linear. Mais recentemente o PIPS vem sendo utilizado com o compilador PAR4ALL. Ainda com a implementação baseada em poliedros, realiza análise de uso de memória e faz análise, teste e tratamento de dependências.

Atualmente a geração de código através do PIPS pode ser feita a partir de código sequencial C possibilitando a geração de código CUDA para GPUs NVIDIA e códigos OpenCL para outros tipos de placas gráficas e anotações de código do tipo OpenMP. A geração de código paralelo é feita através de várias etapas, como descompilação de código e engenharia reversa, regiões convexas de vetores, análise de complexidade do código, análise e testes de dependências.

3.2 Comparativo entre os paralelizadores identificados

Os paralelizadores apresentados neste capítulo foram aqueles identificados dentre os 54.039 artigos pesquisados durante a revisão de escopo para elaboração deste trabalho. Todos os métodos de paralelização encontrados durante a pesquisa foram descritos, analisados e comparados. O resultado desta comparação pode ser visto na Tabela 1. Em suma, os paralelizadores que apresentam melhor desempenho são aqueles com geração de código executado em GPUs, tais como o PPCG, o PLUTO e o PENCIL.

É possível destacar, com base em cada método abordado na Tabela 1, as seguintes características (os métodos foram ordenados do menos recomendado para o mais recomendado):

- **PIPS e Cetus:** Os compiladores PIPS e Cetus não apresentaram informações claras sobre sua performance de paralelização e/ou resultados de *benchmarks*;

- **Aeminum:** O compilador Aeminum é um paralelizador relativamente novo, portanto, apresentou perda de desempenho nos *benchmarks*, necessitando de “dicas” fornecidas por um agente externo, o descaracterizando como totalmente automático;

Tabela 1 – Comparativo entre os paralelizadores identificados

Método	Perf. Média	Melhor perf.	Pior perf.	GPU
Graphite	-	1.11x	1.02x	Não
PPCG com OpenACC	-	121x	0.885x	Sim
Aeminum	-	-	-30x	Não
PLUTO e CLooG	5x	-	-	Sim
JikesRVM	0.87x	13x	-	Sim
PPCG	-	10x	-	Sim
ChiLL	-	-	-	Sim
Cetus	-	-	-	Não
PENCIL	11x	1.4x	-	Sim
PIPS e Par4All	-	-	-	Sim

Fonte: Elaborada pelo autor.

- **Graphite:** O paralelizador Graphite apresentou ganho de performance mínimo quando comparado aos outros compiladores e tem restrição de aplicação, podendo paralelizar somente laços perfeitamente aninhados;

- **JikesRVM:** Embora a máquina virtual JikesRVM tenha apresentado um caso com 13 vezes mais desempenho, este apresentou perdas de desempenho na maioria das execuções;

- **ChiLL:** O compilador ChiLL não apresentou perda de desempenho, porém seus ganhos não foram altos, quando comparado a outros compiladores como PLUTO e PPCG;

- **PENCIL:** O compilador PENCIL apresentou ganhos consideráveis (em média 1,4 vezes mais rápido) na maioria das vezes em que foi testado e altos ganhos em testes específicos (11 vezes mais rápido em alguns casos). Todavia, necessita de uma linguagem específica de entrada, fato que dificulta seu uso em maior escala;

- **PLUTO e CLooG:** O compilador PLUTO apresentou performance média de execução 5 vezes mais rápida do que o programa equivalente em forma sequencial. Pode ser considerado um bom destaque no que se refere a um paralelizador automático de código sequencial C para código paralelo CUDA;

- **PPCG:** O compilador PPCG apresentou ganhos em média de 10 vezes mais rápidos que a execução sequencial, podendo ser considerado um bom paralelizador em relação aos demais, tendo em média o dobro da performance do compilador PLUTO e oferecendo geração de código CUDA e OpenCL;

- **PPCG com diretivas OpenACC:** O compilador PPCG com diretivas OpenACC apresentou os melhores resultados de *benchmarks*, através de um conceito simplificado de análise do código sequencial e geração do código paralelo. Sua melhor performance foi de cerca de 121 vezes em relação a um código sequencial equivalente e, embora na pior

performance tenha sofrido aumento do tempo de execução de 10x, este foi em um *benchmark* de execução totalmente linear. Este compilador também apresenta a possibilidade de paralelização de código sequencial C para código paralelo CUDA e OpenCL;

Os compiladores que utilizam o conceito de poliedros apresentaram boa performance, como no caso do PPCG e PLUTO. Por outro lado, sofrem perda de desempenho em programas com alta dependência de variáveis, sendo esta uma barreira natural a qualquer implementação paralela.

Em relação aos *frameworks* que apresentaram bons ganhos de performance, todos realizaram paralelização para código paralelo executado em GPU. Destes, foi escolhida a ferramenta de paralelização com o uso do compilador PPCG com diretivas OpenACC, por esta apresentar um melhor desempenho, uma menor curva de aprendizagem e geração de código paralelo em CUDA e OpenCL, possibilitando maior abrangência em termos de sistemas com placas gráficas diferentes. Outro fator de escolha para o PPCG com diretivas OpenACC se dá pelo fato das diretivas simplificarem a implementação do paralelizador, no que diz respeito a análise do código sequencial para geração do código paralelo. Essas diretivas são anotadas de forma automática com o uso do anotador de código DawnCC.

3.3 Otimização de código

O DawnCC é um anotador de código que realiza anotações automáticas de diretivas OpenACC em código C e C++. As diretivas OpenACC são utilizadas como indicadores de trechos de códigos sequenciais que podem ser paralelizados pelo compilador PPCG, sendo um exemplo os laços, que são responsáveis por aumentar a complexidade de execução de um código. Em contra-partida, os códigos podem ser executados totalmente em paralelo quando não há dependências de variáveis. Assim, uma iteração do laço não dependerá da iteração anterior, sendo possível executar cada iteração do laço em paralelo. Dessa forma, o anotador DawnCC realiza análise do código sequencial e escreve diretivas OpenACC.

As diretivas OpenACC indicam o tipo de execução suportada por um código, quantidade de memória de GPU a ser alocada, dentre outras diretivas que otimizam a execução do código paralelo. As diretivas possibilitam ganho de desempenho durante a paralelização automática, pois são paralelizados somente laços que possuem independência de variáveis. Sem essa análise, a paralelização de laços com dependência de variáveis torna-se possível, causando perda de eficiência do código paralelo em relação ao sequencial (MENDONcA *et al.*, 2017).

Na Figura 1 é apresentado um trecho de código com dois laços aninhados que foram anotados pelo DawnCC, sendo possível identificar as diretivas de paralelização anotadas. O primeiro `pragma` é uma diretiva de alocação de memória para a GPU referente ao espaço necessário para executar os laços em paralelo. O segundo `pragma` é necessário para indicar à

GPU que esta é uma diretiva de *kernel*. O terceiro e quarto `pragmas` indicam que os laços são independentes e portanto, podem ser executados totalmente em paralelo.

Figura 1 – Diretivas OpenACC anotadas pelo DawnCC

```
/*loop 1*/
#pragma acc data pcopy(n[0:1999])
#pragma acc kernels
#pragma acc loop independent
for (i = 0; i < 1000; i++) {
    /*loop 2*/
    #pragma acc loop independent
    for (j = 0; j < 1000; j++) {
        n[i+j] = i*j;
    }
}
```

Fonte: Elaborada pelo autor.

É possível utilizar o paralelizador PPCG sem as diretivas OpenACC, porém o código paralelo poderá possuir menor aproveitamento do processamento paralelo e com isso, apresentar menor desempenho. Esses dois casos podem ser observados em Verdoolaege *et al.* (2013) e MOREIRA (2015), sendo a principal diferença entre os dois, a melhor performance de execução do código paralelo quando as diretivas para o paralelizador são fornecidas.

3.4 Considerações Finais

Neste capítulo foi apresentada a revisão de escopo de ferramentas relacionadas à paralelização automática, conforme registrado na seção 3.1. Na seção 3.2 foi realizado a comparação entre as ferramentas de paralelização levantadas. A comparação das ferramentas foi realizada com base em critérios de desempenho, em menor restrição de paralelização e no código de entrada do paralelizador. Como resultado, foi possível selecionar o compilador PPCG com diretivas OpenACC para implementação do sistema de paralelização automática, aliado ao anotador de código DawnCC, descrito na seção 3.3. O anotador de código DawnCC será utilizado para realizar anotações OpenACC em um código sequencial C, que será fornecido como entrada no compilador PPCG.

CAPÍTULO

4

PROPOSTA DO SISTEMA

O sistema de paralelização automática de código foi desenvolvido para uso na Web, de forma a escalar o acesso para uma grande quantidade de usuários, podendo ser realizado de forma remota, sem quaisquer problemas de compatibilidade. Por esses motivos, foram pesquisadas tecnologias, *frameworks* e arquiteturas para a implementação do sistemas Web. Na seção 4.1 serão detalhadas as características e tecnologias utilizadas no sistema desenvolvido.

4.1 Características do Sistema

Para o desenvolvimento de sistemas Web escaláveis, considerando seu uso e manutenção contínua, é necessário utilizar arquiteturas de desenvolvimento que facilitem este processo. Por esse motivo será utilizada a arquitetura de desenvolvimento *Model*, *View* e *Controller* (MVC), conforme proposto em Pop e Altar (2014), a qual será detalhada e explicada na subseção 4.1.1. Também são detalhados alguns conceitos que devem ser considerados no desenvolvimento Web. Para implementar o sistema Web com a arquitetura MVC e os padrões de desenvolvimento Web que serão apresentados no decorrer deste Capítulo, será utilizado o *framework* Laravel (subseção 4.1.2).

Para ampliar a gama de usuários que terão capacidade de utilizar o sistema de paralelização automático, a interface do sistema foi implementada como uma ferramenta de programação em blocos, a qual gera o código sequencial C a ser fornecido como entrada para o paralelizador. Dessa forma, usuários com baixo conhecimento em programação poderão utilizar o sistema.

A principal motivação por trás da implementação de uma interface mais amigável ao usuário surge como uma tentativa para se minimizar a dificuldade em se aprender o vocabulário de uma linguagem de programação. Nesse contexto, blocos simplificam a entrada de dados, pois escolher um bloco em uma paleta de componentes é mais simples do que lembrar de uma palavra. Essa forma de interface utiliza a cognição ao invés de lidar unicamente com a capacidade de o

usuário se lembrar do comando correto (BAU *et al.*, 2017).

Através da interface de programação em blocos, espera-se despertar o interesse em programação em um possível usuário do sistema, tendo em vista a possibilidade de interação com o mesmo de maneira simplificada. Considerando um estudo envolvendo estudantes do 8º e 9º anos, onde em um primeiro grupo foi apresentada a programação em blocos e textual e em um segundo grupo, apenas a programação textual, foi possível detectar que no grupo que programou inicialmente com blocos houve o interesse de 88% dos alunos em outros cursos de programação, enquanto que no grupo que programou apenas em texto houve interesse de 47%. Outro efeito observado, principalmente na interação de usuários leigos é que existe um maior número de erros na programação em código textual. A programação em blocos minimiza a geração de erros básicos já que dois blocos incompatíveis não se conectam (BAU *et al.*, 2017).

4.1.1 Arquitetura MVC e padrões de desenvolvimento Web

Para desenvolvimento de aplicações web dinâmicas são combinados HTML, linguagens *server-side* e banco de dados, sendo necessário elaborar uma arquitetura que separe cada uma dessas abstrações. Por esse motivo, será apresentada a arquitetura MVC que realiza a separação entre *Model*, *View* e *Controller*. O *Controller* recebe e responde requisições do protocolo HTTP e entre a requisição e a resposta, realiza chamadas à *Model* e à *View* (POP; ALTAR, 2014).

A *Model* realiza comunicação com fontes de dados de acordo com as regras de negócio, tais como: bancos de dados, web *services*, arquivos, dentre outros. A chamada ao *Model* possibilita a geração de páginas com dados dinâmicos e a chamada às *Views* realiza a montagem da página com o *layout* em HTML e CSS. Portanto, o fluxo de uma requisição na arquitetura MVC é de: Requisição → *Controller* → *Model* → *Controller* → *View* → *Controller* → Resposta (POP; ALTAR, 2014).

Um *Object Relational Mapper* (ORM) estabelece uma relação entre programação orientada a objetos e bancos de dados, sendo modelados objetos relacionados a registros de bancos de dados. Uma arquitetura MVC deve disponibilizar uma forma de acessar e alterar dados de um banco de dados, sendo o conceito de ORMs, na maioria das vezes, a melhor forma de interação (POP; ALTAR, 2014).

Aplicações web utilizam diversas tecnologias diferentes e por isso, podem apresentar diversas vulnerabilidades diferentes. Os riscos de segurança podem ser divididos em alguns tópicos, sendo eles:

- Validação de entrada do usuário: estouro de *buffer*, *cross site scripting* (xss), SQL *injection*, *canocalization*, CSRF.
- Autenticação: *Sniffing*, ataque de força bruta, ataques de dicionário, falsificação de *cookies*, roubo de identidade.

- Autorização: Violação de privilégios de acessos, visualização de dados privados, alteração de dados.
- Configurações: Acesso não autorizado a painéis de configuração, base de dados em textos sem validações de sessão.
- Informações sensíveis: acesso a dados críticos do BD, alterações de dados e *sniffing*.
- Sessão: roubo de sessão, alteração de sessão, *man in the middle*.
- Criptografia: senhas fracas, criptografia fraca.
- Manipulações de parâmetros: *string* de consulta, campos do formulário, *cookies* e manipulação do cabeçalho HTTP.
- Gerenciamento de exceções: ataques de *Distributed Denial of Service* (DDoS).

Torna-se necessário proteger as aplicações contra todos esses tipos de ataques, em todas as camadas da arquitetura MVC. Nesse contexto, ferramentas de desenvolvimento devem prover ferramentas de roteamento HTTP, que agilizam o desenvolvimento de aplicações web, seja para controle de acesso, o que também auxilia na melhoria da segurança ou para maior confiabilidade no uso da aplicação quanto a navegação entre as páginas do aplicativo (POP; ALTAR, 2014).

4.1.2 *Framework* Laravel

Como as aplicações web são extensas, as empresas demandam a construção de suas aplicações Web de forma rápida e eficiente. A maioria dos *frameworks* são simples e por isso possuem diversas limitações. O *framework* Laravel padroniza diversas metodologias de desenvolvimento e encapsula diversas regras de desenvolvimento web que não são relacionadas com a regra de negócio. Isso torna o desenvolvimento de aplicações web mais confiável e rápido com robustez e escalabilidade (HE, 2015/01).

A arquitetura do *framework* Laravel é implementada em várias camadas. A primeira camada é o núcleo do Laravel, a segunda camada implementa uma camada persistente do banco de dados (BD), ou seja, cria *models* a partir do BD. A terceira camada possibilita operações de manipulação de dados, como *update*, *select*, *delete*, dentre outros (HE, 2015/01).

Implementações em Laravel possibilitam as seguintes vantagens de desenvolvimento:

- Possibilidade de computação distribuída.
- Funcionalidades podem ser incrementadas sem necessidade de recompilar todo o *framework*, ou nesse caso, sem necessidade de realizar um *deploy* completo.
- Módulos podem ser reaproveitados.

- Facilidade de atualização.
- Implementação separada da interface.
- Implementação nativa da arquitetura MVC.

Uma aplicação web desenvolvida em Laravel pode ser facilmente implementada de acordo com os requisitos MVC, visto que o Laravel possui um motor de rotas HTTP. Isso provê uma fácil implementação de *middlewares*, os quais representam interceptadores de uma requisição HTTP que podem atuar assim que a requisição é recebida ou antes da resposta ser enviada. Com isso, o *framework* é totalmente baseado na arquitetura MVC, além de implementar as *Models* como ORMs e disponibilizar componentes de autenticação (CHEN *et al.*, 2017).

Várias outras dependências podem ser incluídas, pois são disponibilizadas pela grande comunidade *open-source* que o Laravel possui.

4.2 Considerações Finais

Neste capítulo foi apresentado as arquiteturas, padrões de desenvolvimento, tecnologias utilizadas para implementação do sistema e tratativas de segurança que uma aplicação Web deve realizar para evitar ataques mal-intencionados.

No Capítulo 5 serão apresentados os resultados desta monografia, sendo: o sistema implementado, interface do sistema implementada e alguns testes de uso e eficiência de paralelização do sistema.

CAPÍTULO 5

RESULTADOS OBTIDOS

Com o intuito de testar o uso e eficiência do paralelizador PPCG com diretivas OpenACC, foi elaborado um estudo de caso utilizando um código C que realiza a multiplicação de duas matrizes 100x100. Para medir o tempo de execução do código e com isso, determinar a eficiência da multiplicação de matrizes entre CPU x GPU.

O código C foi anotado com diretivas OpenACC utilizando o DawnCC e em seguida paralelizado com o PPCG. Os programas foram executados duas vezes em CPU e duas vezes na GPU. No primeiro teste, o código executou em 0,018583 segundos na CPU e em 0,006266 segundos na GPU (Figura 2), sendo a execução em GPU cerca de 2,96 vezes mais rápida do que na CPU. No segundo teste, o código executou em 0,018660 segundos na CPU e em 0,016715 segundos na GPU (Figura 3), sendo que no segundo teste, a execução em GPU foi cerca de 1,11 vezes mais rápida que na CPU.

Os testes foram realizados em um computador com as seguintes configurações: Processador Intel(R) Core(TM) i5-7600K CPU @3.80GHz, Memória RAM 8GB DDR4 2400MHz, placa de vídeo NVIDIA GeForce GTX 1080 8GB, Sistema Operacional Ubuntu 18.04.3 LTS bionic com a versão *kernel* Linux 5.3.0-28-generic.

Figura 2 – Primeiro teste execução código CPU x código GPU

```
lucas@lucas-desktop:/v
gmas$ ./multmatrizCPU

Time CPU: 0.018583
lucas@lucas-desktop:/v
gmas$ ./multmatrizGPU

Time GPU: 0.006266
```

Fonte: Elaborada pelo autor.

Figura 3 – Segundo teste execução código CPU x GPU

```
lucas@lucas-desktop:/\
gmas$ ./multmatrizCPU

Time CPU: 0.018660
lucas@lucas-desktop:/\
gmas$ ./multmatrizGPU

Time GPU: 0.016715
```

Fonte: Elaborada pelo autor.

5.1 Implementação da interface do sistema

A interface do sistema foi estruturada de modo a disponibilizar o serviço de programação em blocos, por meio do *framework open-source BlocklyProp* desenvolvido pela empresa *Parallax Inc*. Para realizar a programação por meio de blocos o usuário irá dispor de uma paleta com vários blocos diferentes, como por exemplo, blocos do tipo laços, blocos condicionais, dentre outros. Na Figura 4 pode ser observado a paleta de blocos da interface do sistema.

Um exemplo de uso da interface em blocos é apresentado na Figura 5, no qual foi implementado o algoritmo de ordenação bolha. Na Figura 6 é mostrado o código equivalente em C, o qual foi gerado pela interface de programação em blocos.

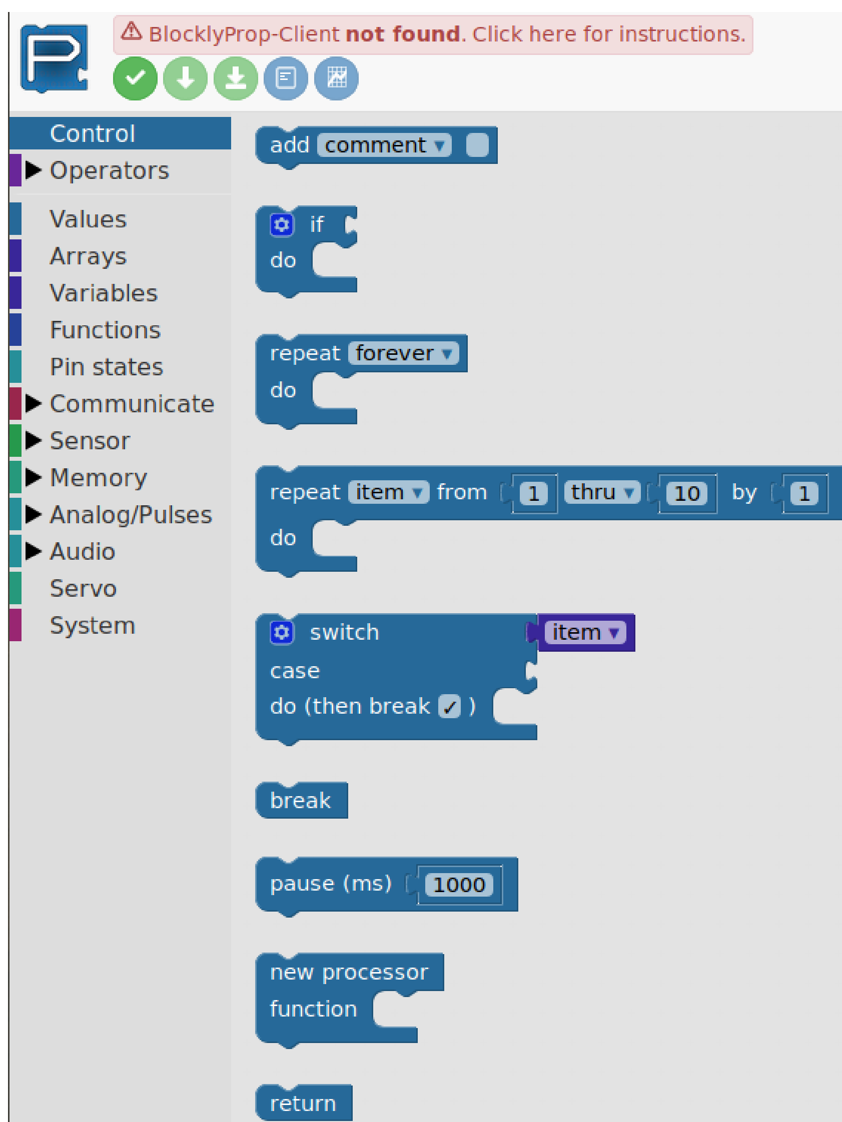
Usuários com conhecimentos intermediários em programação poderão utilizar o sistema sem a necessidade do uso da programação em blocos, podendo realizar a entrada de seu código-fonte sequencial em C diretamente, através de *upload* do arquivo que contenha o código-fonte em C. A interface gráfica é implementada acoplada junto ao paralelizador automático, sendo a *View* do sistema, demonstrando-se a forma de entrada do código sequencial e o resultado de execução do programa a partir do código paralelo gerado automaticamente.

5.2 Implementação da arquitetura do sistema

O desenvolvimento do sistema foi iniciado com foco em implementá-lo todo em uma máquina virtual, o que facilitaria a portabilidade de todo o projeto desenvolvido. Contudo, a implementação em uma máquina virtual foi descartada devido a alta complexidade de realizar o mapeamento e virtualização de uma GPU em uma máquina virtual, além de execução lenta do sistema.

Outra abordagem adotada foi a de se criar todo o sistema em um ambiente Docker, uma ferramenta de virtualização que utiliza recursos diretamente do sistema operacional hospedeiro, tornando a execução dos processos virtualizados mais leve e rápida, sem perder os ganhos de portabilidade e compatibilidade. Esta abordagem também foi descartada pelo mesmo motivo que ocorreu no uso direto com máquina virtual. A complexidade de mapeamento e virtualização da

Figura 4 – Paleta de blocos

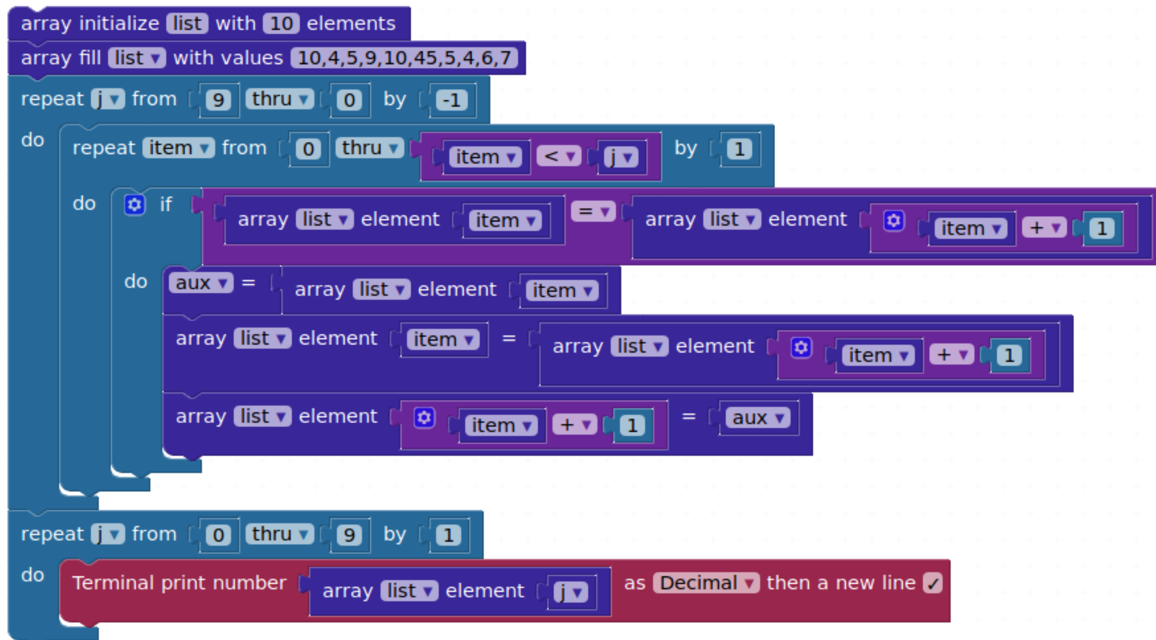


Fonte: Elaborada pelo autor.

GPU para um contêiner é alta, perdendo performance de execução do código paralelo devido ao processo de virtualização da placa gráfica. Para uso do sistema com o Docker seria necessário mapear a GPU para o contêiner do PPCG e para o contêiner do DawnCC, e tornar as chamadas de terminal desses dois contêineres acessíveis ao contêiner do servidor Web. Isso implicaria em dependências entre os contêineres, sendo estas interdependências contrárias à premissa do Docker, que consiste em utilizar contêineres independentes.

O sistema Web desenvolvido com o *framework* Laravel possui o fluxo de paralelização de um código sequencial conforme demonstrado na Figura 7. O início de execução da paralelização recebe como entrada o código sequencial, que pode ter como origem o *upload* de um arquivo ou uma interface de programação em blocos, apresentada na seção 5.1. Ambas as formas de entrada realizam uma requisição HTTP POST que envia o código sequencial para o *framework* Laravel,

Figura 5 – Algoritmo de ordenação bolha programado em blocos



Fonte: Elaborada pelo autor.

sendo essa a etapa de entrada de dados pela *view* (interface do sistema).

O *controller* Laravel recebe o código sequencial enviado pela *view* e realiza uma chamada para o anotador de código DawnCC que realiza a anotação do código com diretivas de execução paralela OpenACC descrito no Capítulo 3, seção 3.3. Após a anotação do código, o *controller* inicia a execução do compilador PPCG, descrito no Capítulo 3, seção 3.2, que realiza a paralelização do código sequencial para código paralelo CUDA ou OpenCL, visto que, o compilador PPCG suporta a saída em ambas as linguagens. Após paralelização do código sequencial, o *controller* realiza uma chamada de sistema para compilação do código paralelo e em sequência realiza a chamada para execução do mesmo em GPU. Após a execução do código paralelo, o *controller* Laravel recebe e retorna a saída do programa paralelo para a *view* que pode ser observado na Figura 8.

5.3 Considerações Finais

Neste capítulo foi apresentado um teste inicial de paralelização automática do PPCG com diretivas OpenACC, onde foi mostrado um exemplo de multiplicação de matrizes no qual o código paralelo executou em até 2,96 vezes mais rápido que o código sequencial, demonstrando a eficiência do método de paralelização automática utilizado.

Todo o sistema foi implementado de acordo com a arquitetura MVC e oferece uma interface de programação em blocos. O código gerado pela programação em blocos é paralelizado

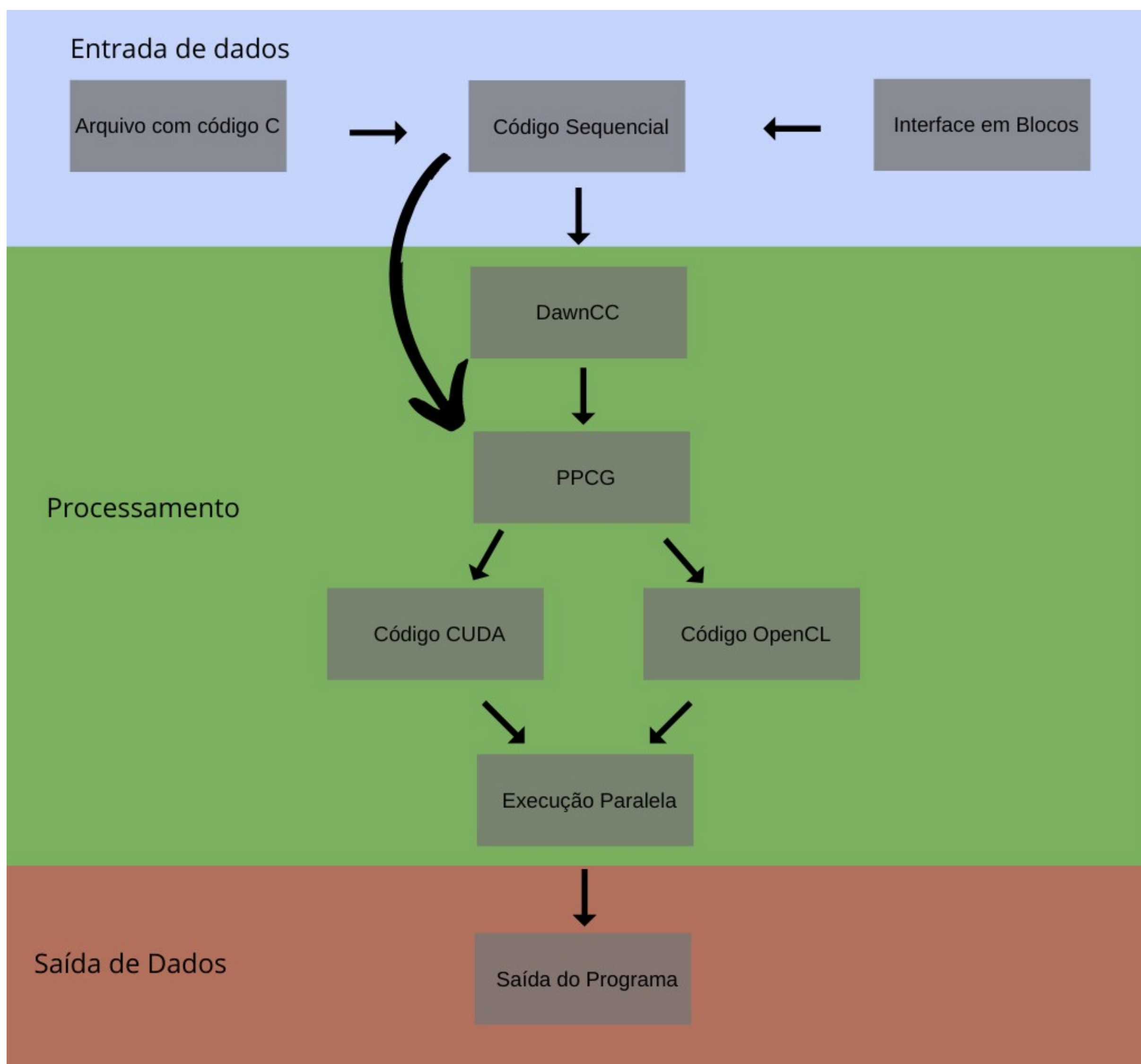
Figura 6 – Código gerado do algoritmo de ordenação bolha

```
/* SERIAL_TERMINAL USED */


// ------ Libraries and Definitions ------
#include "simpletools.h"

// ------ Global Variables and Objects ------
int list[10];
int j;
int item;
int aux;



// ------ Main Program ------
int main() {

  int __tmpArr3[] = {10, 4, 5, 9, 10, 45, 5, 4, 6, 7};
  memcpy(list, __tmpArr3, 10 * sizeof(int));
  for (j = 9; j >= 0; j--) {
    for (item = 0; item <= item < j; item++) {
      if (list[item] == list[item + 1]) {
        aux = list[item];
        list[constrainInt(item, 0, 9)] = list[item + 1];
        list[constrainInt(item + 1, 0, 9)] = aux;
      }


    }
  }
  for (j = 0; j <= 9; j++) {
    print("%d\r", list[j]);

  }
}
```

Fonte: Elaborada pelo autor.

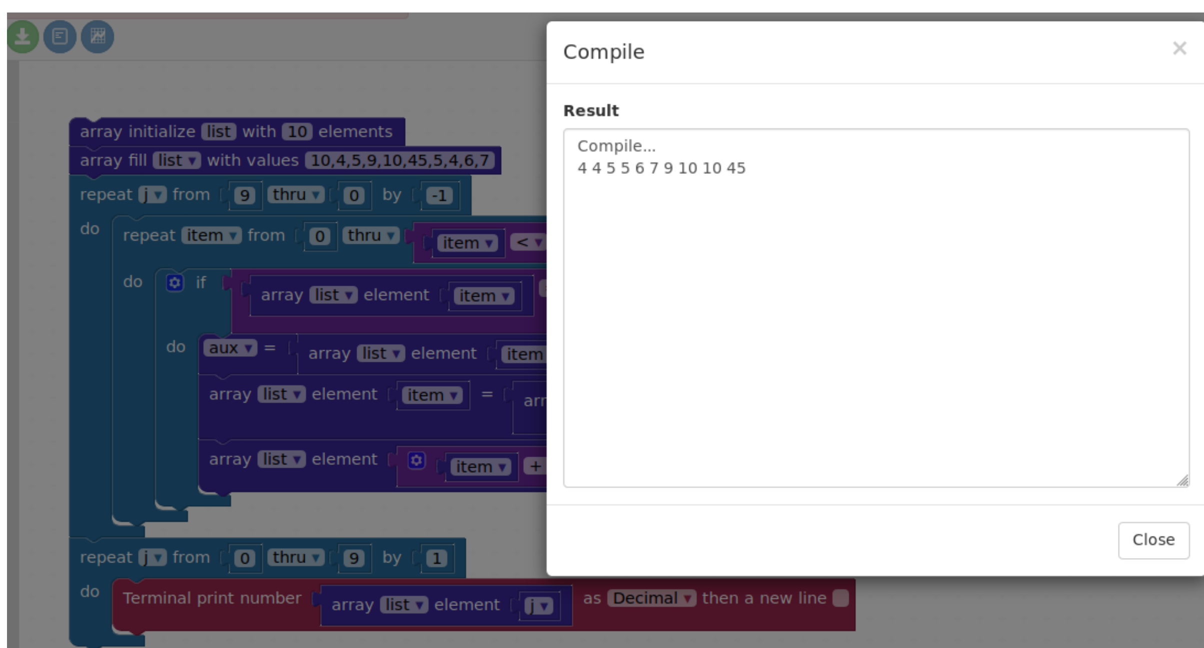
automaticamente, executado em GPU e o resultado da execução é disponibilizado ao usuário na interface do sistema (Figura 8).

Figura 7 – Arquitetura de Paralelização



Fonte: Elaborada pelo autor.

Figura 8 – Saída de execução na interface do sistema



Fonte: Elaborada pelo autor.

CAPÍTULO

6

CONCLUSÕES

Neste trabalho foi abordada a técnica de paralelização automática, enfatizando-se o seu ganho de performance que foi possível graças à redução do tempo de processamento de uma determinada tarefa. Nesse contexto, a programação paralela mostra sua importância em aplicações que demandam alto poder de processamento. Sendo diretamente ligado à computação de alto desempenho, esta é usualmente implementada em *clusters* e mais recentemente, em GPUs para tarefas que demandam alto poder computacional, sendo um exemplo, a técnica de *Machine Learning*.

Assim, a computação paralela é relativamente complicada e requer do programador conhecimentos relacionados ao fluxo de execução de um programa, ao fluxo computacional de *threads*, análise e tratamento de dependência de variáveis de forma que o código paralelo tenha mais desempenho que o código sequencial. Dessa forma, erros de lógica se tornam mais suscetíveis, devido a necessidade de conhecimento de programação paralela por parte do programador. Portanto é viável considerar o uso de paralelizadores automáticos para reduzir o tempo de execução de programas que realizam cálculos intensos e repetitivos.

Várias ferramentas de paralelização automática foram analisadas. Algumas não ofereceram ganhos positivos de performance, tais como: Graphite e Aeminum. Cetus e PIPS não ofereceram resultados claros da performance de paralelização. Ferramentas como PENCIL, ChiLL, PPCG, PLUTO, JikesRVM e PPCG com diretivas OpenACC apresentaram uma descrição e resultados mais completos.

Assim como descrito no Capítulo 3, seção 3.2, foi escolhido o compilador PPCG com diretivas OpenACC para implementação do sistema de paralelização automática proposto, por apresentar menor curva de aprendizado, possibilidade de geração de código paralelo CUDA e OpenCL e melhor performance, apresentando código paralelo com execução em até 2,96 vezes mais rápido que o código sequencial (Capítulo 4) . A geração de código paralelo CUDA e OpenCL possibilita a implementação em uma quantidade de maior de máquinas com diferentes

placas gráficas. Foi possível utilizar o paralelizador PPCG com diretivas OpenACC pelo fato das diretivas poderem ser automaticamente anotadas com o DawnCC, tornando o sistema paralelizador totalmente automático, sem necessidade de conhecimento prévio em programação paralela durante o processo de paralelização de código.

REFERÊNCIAS

ALSHUWAIKHAT, H. M.; ABUBAKAR, I. An integrated approach to achieving campus sustainability: assessment of the current campus environmental management practices. **Journal of Cleaner Production**, v. 16, n. 16, p. 1777 – 1785, 2008. ISSN 0959-6526. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0959652607002545>. Citado na página 19.

Baghdadi, R.; Beaugnon, U.; Cohen, A.; Grosser, T.; Kruse, M.; Reddy, C.; Verdoolaege, S.; Betts, A.; Donaldson, A. F.; Ketema, J.; Absar, J.; Haastregt, S. v.; Kravets, A.; Lokhmotov, A.; David, R.; Hajiyev, E. Pencil: A platform-neutral compute intermediate language for accelerator programming. In: **2015 International Conference on Parallel Architecture and Compilation (PACT)**. [S.l.: s.n.], 2015. p. 138–149. ISSN 1089-795X. Citado nas páginas 17 e 32.

BAU, D.; GRAY, J.; KELLEHER, C.; SHELDON, J.; TURBAK, F. Learnable programming: Blocks and beyond. **Commun. ACM**, ACM, New York, NY, USA, v. 60, n. 6, p. 72–80, maio 2017. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/3015455>. Citado na página 38.

CHEN, X.; JI, Z.; FAN, Y.; ZHAN, Y. Restful API architecture based on laravel framework. **Journal of Physics: Conference Series**, IOP Publishing, v. 910, p. 012016, oct 2017. Disponível em: <https://doi.org/10.1088%2F1742-6596%2F910%2F1%2F012016>. Citado na página 40.

Dave, C.; Bae, H.; Min, S.; Lee, S.; Eigenmann, R.; Midkiff, S. Cetus: A source-to-source compiler infrastructure for multicores. **Computer**, v. 42, n. 12, p. 36–42, Dec 2009. ISSN 0018-9162. Citado nas páginas 17 e 32.

DI, P.; YE, D.; SU, Y.; SUI, Y.; XUE, J. Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on gpus. In: **41st International Conference on Parallel Processing, ICPP 2012, Pittsburgh, PA, USA, September 10-13, 2012**. [s.n.], 2012. p. 350–359. Disponível em: <https://doi.org/10.1109/ICPP.2012.19>. Citado nas páginas 17, 25, 26, 29 e 30.

GONçALVES, C. F. M. Otimização de programas em runtime na plataforma aeminium. In: . [S.l.: s.n.], 2013. Citado nas páginas 17 e 29.

Gonçalves, C. O.; Spolon, R.; Lobato, R. S.; Manacero, A.; Lobato, D. C. Automatic loops parallelization. In: **2014 9th Iberian Conference on Information Systems and Technologies (CISTI)**. [S.l.: s.n.], 2014. p. 1–5. ISSN 2166-0727. Citado na página 25.

HE, R. Y. design and implementation of web based on laravel framework. In: **2014 International Conference on Computer Science and Electronic Technology (ICCSET 2014)**. Atlantis Press, 2015/01. ISBN 978-94-62520-47-9. ISSN 2352-538X. Disponível em: <https://doi.org/10.2991/iccset-14.2015.66>. Citado na página 39.

KHAN, M.; BASU, P.; RUDY, G.; HALL, M.; CHEN, C.; CHAME, J. A script-based autotuning compiler system to generate high-performance cuda code. **ACM Trans. Archit. Code Optim.**, ACM, New York, NY, USA, v. 9, n. 4, p. 31:1–31:25, jan. 2013. ISSN 1544-3566. Disponível em: <http://doi.acm.org/10.1145/2400682.2400690>. Citado nas páginas 17 e 31.

Leung, Alan Chun Wai. **Automatic Parallelization for Graphics Processing Units in JikesRVM**. UWSpace, 2008. Disponível em: <http://hdl.handle.net/10012/3752>. Citado nas páginas 17, 26 e 30.

LEVAC, D.; COLQUHOUN, H.; O'BRIEN, K. K. Scoping studies: advancing the methodology. **Implementation Science**, v. 5, n. 1, p. 69, Sep 2010. ISSN 1748-5908. Disponível em: <https://doi.org/10.1186/1748-5908-5-69>. Citado nas páginas 20, 21 e 23.

LUNARDI, G. L.; SIMõES, R.; FRIO, R. S. TI Verde: uma análise dos principais benefícios e práticas utilizadas pelas organização. **REAd. Revista Eletrônica de Administração (Porto Alegre)**, scielo, v. 20, p. 1 – 30, 04 2014. ISSN 1413-2311. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S1413-23112014000100001&nrm=iso>. Citado na página 19.

LUNARDI, G. L.; SIMõES, R.; FRIO, R. S. Um estudo do uso eficiente de programas em placas gráficas. **REAd. Revista Eletrônica de Administração (Porto Alegre)**, scielo, v. 20, p. 1 – 30, 04 2014. ISSN 1413-2311. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S1413-23112014000100001&nrm=iso>. Citado na página 20.

MENDONcA, G.; aES, B. G.; ALVES, P.; PEREIRA, M.; ARAúJO, G.; PEREIRA, F. M. Q. a. Dawncc: Automatic annotation for data parallelism and offloading. **ACM Trans. Archit. Code Optim.**, ACM, New York, NY, USA, v. 14, n. 2, p. 13:1–13:25, maio 2017. ISSN 1544-3566. Disponível em: <http://doi.acm.org/10.1145/3084540>. Citado na página 35.

MINI, M. L.; NCOURT, C. A.; OELHO, F. C.; REUSILLET, B. C.; UELTON, S. G.; RIGOIN, F. I.; OUVELOT, P. J.; ERYELL, R. K.; ILLALON, P. V. Pips is not (just) polyhedral software adding gpu code generation in pips. In: . [S.l.: s.n.], 2011. Citado nas páginas 17 e 33.

MOREIRA, K. C. A. **ANOTAÇÃO AUTOMÁTICA DE CÓDIGO COM DIRETIVAS OPENACC**. Dissertação (Mestrado) — Instituto de Ciências Exatas Programa de Pós Graduação em Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, junho 2015. Citado nas páginas 17, 28 e 36.

POP, D.-P.; ALTAR, A. Designing an mvc model for rapid web application development. **Procedia Engineering**, v. 69, p. 1172 – 1179, 2014. ISSN 1877-7058. 24th DAAAM International Symposium on Intelligent Manufacturing and Automation, 2013. Disponível em: <http://www.sciencedirect.com/science/article/pii/S187770581400352X>. Citado nas páginas 37, 38 e 39.

VERDOOLAEGE, S.; JUEGA, J. C.; COHEN, A.; GóMEZ, J. I.; TENLLADO, C.; CATTHOOR, F. 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>. Citado nas páginas 17, 26, 27 e 36.

VIEIRA, C. M. Paralelização automática de laços para arquiteturas multicore. In: . [S.l.: s.n.], 2010. Citado nas páginas 17 e 26.