UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DECISÃO AUTOMATIZADA DE MAPEAMENTO DE TAREFAS COM OPENACC EM ARQUITETURAS PARALELAS HÍBRIDAS

DISSERTAÇÃO DE MESTRADO

Renato Pizzinato Ferrari

Santa Maria, RS, Brasil

2016

DECISÃO AUTOMATIZADA DE MAPEAMENTO DE TAREFAS COM OPENACC EM ARQUITETURAS PARALELAS HÍBRIDAS

Renato Pizzinato Ferrari

Dissertação apresentada ao Curso de Mestrado Programa de Pós-Graduação em Informática (PPGI), Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

Orientadora: Prof^a^. Dr^a^. Andrea Schwertner Charão

Santa Maria, RS, Brasil

2016

Ferrari, Renato Pizzinato

Decisão Automatizada de Mapeamento de Tarefas com OpenACC em Arquiteturas Paralelas Híbridas / por Renato Pizzinato Ferrari. – 2016.
76 f.: il.; 30 cm.

Orientadora: Andrea Schwertner Charão
Dissertação (Mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Informática, RS, 2016.

1. OpenACC. 2. GPU. 3. Processamento paralelo. I. Charão, Andrea Schwertner. II. Título.

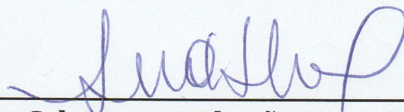
© 2016
Todos os direitos autorais reservados a Renato Pizzinato Ferrari. A reprodução de partes ou do todo deste trabalho só poderá ser feita mediante a citação da fonte.
E-mail: renato.ferrari@ufsm.br

Renato Pizzinato Ferrari

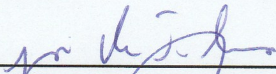
DECISÃO AUTOMATIZADA DE MAPEAMENTO DE TAREFAS COM OPENACC EM ARQUITETURAS PARALELAS HÍBRIDAS

Dissertação apresentada ao Curso de Mestrado do Programa de Pós Graduação em Informática (PPGI), Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

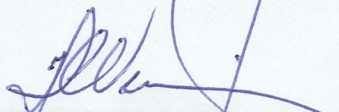
Aprovado em 28 de março de 2016:



Andrea Schwertner Charão, Dra. (UFSM)
(Presidente/Orientador)



João Vicente Ferreira Lima, Prof. Dr. (UFSM)



Haroldo Fraga de Campos Velho, Dr. (INPE)

Santa Maria, RS

2016

DEDICATÓRIA

-a Deus, pela fé que me mantém vivo e fiel à vida honesta de trabalho e de estudo;
-à minha família que soube entender a minha ausência nos muitos momentos desde que ingressei no mestrado, até a conclusão desta dissertação;
-à minha querida e amada esposa Mauára e meus filhos René e Lorenzo, pela ajuda e por aguentarem meus momentos de ansiedade e estresse nos meses em que me dediquei ao mestrado;
-a todos os professores que tive até hoje sem exceção ;

AGRADECIMENTOS

- À minha querida orientadora Prof^{a}. Dr^{a}. Andrea Schwertner Charão, pela dedicação, carinho, esperança, humildade e por ter recebido meu trabalho de forma profissional e fraterna;
- À minha querida e amada esposa Mauára, a meus filhos René e Lorenzo, que me apoiaram nas diversas madrugadas, feriados e fins de semana que fiquei trabalhando sobre esta dissertação;
- A todos os meus professores da UFSM, que participaram da minha formação científica;
- Aos meus amigos e colegas da UFSM.

"Todo o futuro da nossa espécie, todo o governo das sociedades, toda a prosperidade moral e material das nações dependem da ciência, como a vida do homem depende do ar. Ora, a ciência é toda observação, toda exatidão, toda verificação experimental. Perceber os fenômenos, discernir as relações, comparar as analogias e as dessemelhanças, classificar as realidades, e induzir as leis, eis a ciência; eis, portanto, o alvo que a educação deve ter em mira. Espertar na inteligência nascente as faculdades cujo concurso se requer nesses processos de descobrir e assimilar a verdade."

— RUI BARBOSA.

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

DECISÃO AUTOMATIZADA DE MAPEAMENTO DE TAREFAS COM OPENACC EM ARQUITETURAS PARALELAS HÍBRIDAS

AUTOR: RENATO PIZZINATO FERRARI
ORIENTADORA: ANDREA SCHWERTNER CHARÃO
Local da Defesa e Data: Santa Maria, 28 de Março de 2016.

O presente trabalho de mestrado concentrou-se no desenvolvimento de um método de decisão com etapas automatizadas, a fim de auxiliar o desenvolvedor a tomar a seguinte decisão em um dado sistema híbrido: *em qual unidade do sistema deve ser mapeada uma determinada tarefa, para que se obtenha o melhor desempenho no hardware disponível*? Os programas paralelos alvo deste trabalho devem ser desenvolvidos com o padrão OpenACC, que usa diretivas de compilação para expressar o paralelismo e foi criado para facilitar a programação em sistemas híbridos formados por CPU e GPU.A abordagem utilizada neste trabalho é empírica, baseada em observações do desempenho de programas em diferentes configurações e com diferentes parâmetros e dados de entrada. As propostas formuladas não têm por objetivo garantir a melhor decisão de mapeamento, mas sim abreviar, na medida do possível, o processo de decisão. Visando aprofundar esta questão de desempenho, no início deste trabalho de mestrado foram feitos experimentos com um *benchmark* para OpenACC. A abordagem adotada neste trabalho tem por hipótese que o desempenho em CPU e em GPU possa ser estimado para uma determinada tarefa, em um dado sistema híbrido real. Essa estimativa pode ser aproximada pois, no pior dos casos, será equivalente a uma estimativa errônea realizada manualmente, que será percebida e poderá ser corrigida para execuções subsequentes. Dessa forma propõe que a estimativa de desempenho em CPU e GPU seja feita baseando-se conjuntamente nos seguintes critérios:tamanho dos dados de entrada, complexidade no tempo e no espaço e desempenho do hardware alvo em *benchmarks*. Para formar uma base de apoio à decisão, propõe-se que seja construída e mantida uma tabela em que cada linha é um *benchmark* em OpenACC, possivelmente pertencente a uma *suite* de *benchmarks* como o EPCC. Sua criação, que requer várias execuções de alguns *benchmarks*, ocorre uma única vez para um dado sistema híbrido e seus dados são, potencialmente, aproveitados em diferentes aplicações e execuções.Visando atingir o objetivo de abreviar o processo e exigir um mínimo de interferência do desenvolvedor, desenvolveu-se uma ferramenta que automatiza partes desse processo. Foram escolhidos três programas, pertencentes ao *benchmark* Polybench. São eles: `gramschmidt` (decomposição pelo método de Gram-Schmidt), `lu` (decomposição LU) e `durbin` (solução de sistema com matriz de Toeplitz). Cada um deles possui complexidade computacional diferente. A eficácia da decisão automatizada pode ser verificada comparando-se os tempos de execução entre Host, Device e da Ferramenta. A decisão automatizada realizada pela ferramenta determinou que a execução da função de Gram-Schmidt fosse na GPU quando a ordem da matriz fosse maior ou igual 400. A diferença entre a ordem da matriz observada 300 para ordem calculada 400 é devida à diferença entre a quantidade de operações aritméticas estimadas da função de correlação e a função de Gram-Schmidt. A eficácia da ferramenta de decisão, que tem por base a análise de um *benchmark* é restringida aos algoritmos que possuem complexidade computacional no

tempo similar ao do *benchmark*. As diferenças dos valores da memória alocada pelo *benchmark* e o programa paralelizado devem-se a parâmetros que não são facilmente mensuráveis, como por exemplo a dependência entre as variáveis. Portanto recomenda-se que a escolha do valor da memória utilizada como critério de decisão seja feita através de um processo iterativo, tomando como parâmetro inicial o valor obtido na análise do *benchmark*.

Palavras-chave: OpenACC. GPU. processamento paralelo.

ABSTRACT

Master's Dissertation
Post-Graduate Program in Informatics
Federal University of Santa Maria

AUTOMATED DECISION ON TASK MAPPING WITH OPENACC OVER HYBRID PARALLEL ARCHITECTURES

AUTHOR: RENATO PIZZINATO FERRARI
ADVISOR: ANDREA SCHWERTNER CHARÃO
Defense Place and Date: Santa Maria, March 28st, 2016.

This master's work focused on the development of a decision-making method with automated steps in order to assist the developer to take the following decision in a given hybrid system: *in which system drive a particular task should be mapped, in order to obtain the best performance available hardware*? Parallel programs target this work should be developed with the standard OpenACC, using compiler directives to express parallelism and is designed to facilitate programming in hybrid systems consisting of CPU and GPU.A approach used in this study is empirical, based on observations performance programs in different configurations and with different parameters and input data. The formulated proposals do not aim to guarantee the best decision mapping, but short, as far as possible, the decision process. Aiming to further discuss this issue of performance at the beginning of this master's work were made experiments with a *benchmark* for OpenACC. The approach adopted in this study is hypothesis that performance CPU and GPU can be estimated for a given task at a given real hybrid system. This estimate can be approximated as, at worst, will be equivalent to an erroneous estimate made manually, which will be perceived and can be corrected for subsequent executions. Thus suggests that the performance estimation of CPU and GPU is made based jointly on the following criteria: size of the input data, complexity in time and space and performance target hardware *benchmarks*. To form a basis for decision support, it is proposed that a table is built and maintained on each line is a *benchmark* in OpenACC, possibly belonging to a *suite* of *benchmarks* as EPCC. His creation, which requires multiple runs of some *benchmarks*, occurs only once for a given hybrid system and its data are potentially utilized in different applications and executions. Aiming achieve the goal of shortening the process and require a minimum developer interference, has developed a tool that automates parts of this process. The assessment tool was carried out in order to test its functionality, limitations and quality of the forward estimates of scientific computing programs. Three programs, belonging to the *benchmark* Polybench were chosen. They are: `gramschmidt` (decomposition by Gram-Schmidt method), `lu` (LU decomposition) and `durbin` (system solution Toeplitz matrix). Each has different computational complexity. The effectiveness of automated decision can be verified by comparing the run times between Host, Device and Tools. The automated decision by the tool was determined that the Gram-Schmidt function execution on GPU when the order of the matrix was greater than or equal to 400. The difference between the observed order matrix 300 for Order 400 is calculated due to the difference between the estimated amount of arithmetic operations of the function correlation and Gram-Schmidt function. The effectiveness of the decision tool, which is based on the analysis of a *benchmark* is restricted to algorithms that have computational complexity in time similar to the *benchmark*. The differences in values of memory allocated by the *benchmark* and the parallelized program are due to parameters that are not easily measured, with for

example the dependence between variables. Therefore it is recommended that the choice of the memory value used as a decision criterion is made through an iterative process, taking as initial parameter value obtained in the analysis of *benchmark*.

Keywords: OpenMP. OpenACC.

LISTA DE FIGURAS

Figura 2.1 – Esquema Host Device de uma Unidade de Processamento Gráfico. Fonte: NVIDIA 24
Figura 2.2 – Esquema de um Grid de uma Unidade de Processamento Gráfico. Fonte: NVIDIA 25
Figura 3.1 – Exemplo de estrutura básica de paralelização com OpenACC, com aplicação da diretiva `kernels`. Fonte: (LARKIN, 2015) 32
Figura 3.2 – Resultado da compilação do trecho apresentado na figura 3.1 com o PGI Accelerator. Fonte: (LARKIN, 2015) 35
Figura 3.3 – Exemplo de aplicação da diretiva `data`. 36
Figura 3.4 – Exemplo de uso da cláusula `present`. Fonte: (WOLF, 2012) 36
Figura 3.5 – Estrutura básica de paralelização com OpenACC e chamada de uma função que encontra-se fora da região paralela. 37
Figura 4.1 – O Processo de Gram-Schmidt, Execução Sequencial, Paralelo com OpenMP com 8 threads, Paralelo na GPU com OpenACC com processador AMD 2.4 GHz e GPU Nvidia GTX 550. 44
Figura 5.1 – Esquema de código para tomada de decisão entre processamento sequencial e em paralelo com OpenACC na GPU, representado em linguagem C. 53
Figura 5.2 – Esquema de funcionamento da ferramenta de decisão. 54
Figura 6.1 – Fragmento de código paralelizado da Decomposição LU extraído do benchmark Polybench. 59
Figura 6.2 – Fragmento de código paralelizado da Solução da Matriz de Toeplitz. 61

LISTA DE TABELAS

Tabela 4.1 – Benchmark EPCC, com datasize default de 1048576 bytes, 10 repetições e tempo em microsegundos. 42
Tabela 4.2 – O Processo de Gram-Schmidt, Execução Sequencial, Paralelo com OpenMP com 8 threads, Paralelo na GPU com OpenACC com processador AMD 2.4 GHz e a GPU Nvidia GTX 550. 43
Tabela 5.1 – Exemplo de tabela de apoio à decisão usando funções do benchmark EPCC Nível 1. 50
Tabela 5.2 – Funções do EPCC Nível 1 que não chegaram a ter o speedup maior que 1. . . 51
Tabela 6.1 – Resultados do Experimento com o programa paralelizado com OpenACC que implementa o Processo de Gram-Schmidt 57
Tabela 6.2 – Resultados do Experimento com o programa paralelizado com OpenACC que implementa a Decomposição LU 59
Tabela 6.3 – Resultados do Experimento com o programa paralelizado com OpenACC que implementa a Solução da Matriz de Toeplitz, utilizando o compilador pgcc, uma CPU Xeon com GPU Tesla 2050 60

LISTA DE APÊNDICES

APÊNDICE A – Ferramenta para tomada de decisão 72
APÊNDICE B – Resultado da compilação de um programa para resolução de um sistema linear por decomposição de Cholesky com OpenMP e o compilador pgcc. 76

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
BLAS	Basic Linear Algebra Subprograms
CPU	*Unidade Central de Processamento*
CUDA	*Compute Unified Device Architecture*
EPCC	*Edinburgh Parallel Computing Centre*
FLOPS	*Operações de ponto flutuante por segundo*
FPGA	*Field Programmable Gate Array*
GPU	*Unidade de Processamento Gráfico*
GPGPU	*Unidade de Processamento Gráfico de Propósito Geral*
LDLt	*Lower, Diagonal e Lower triangular*
LLt	*Lower e Lower triangular*
LU	*Lower e Upper*
"O"	*Notação O*
OpenACC	*Open Accelerators*
OpenCL	*Open Computing Language*
OpenMP	*Open Multiprocessing*
PCIe	*Peripheral Component Interconnect Express*
PGI	*PORTLAND GROUP, INC.*
UFSM	*Universidade Federal de Santa Maria*
ULA	*Unidade Lógica e Aritmética*

SUMÁRIO

1 INTRODUÇÃO 17
1.1 Objetivo do Trabalho 18
1.2 Delimitação do Trabalho 19
1.3 Etapas da Pesquisa 19
1.4 Organização do Texto 20
2 ARQUITETURAS COMPUTACIONAIS DE ALTO DESEMPENHO 21
2.1 Processamento Multicore 22
2.2 Processamento em GPU 22
2.3 Arquiteturas Paralelas Híbridas 24
3 PROGRAMAÇÃO DE ALTO DESEMPENHO 27
3.1 HPF e Co-array Fortran 28
3.2 OpenMP 29
3.3 CUDA 30
3.4 OpenACC 31
3.4.1 Níveis de Paralelismo 32
3.4.2 Diretivas de Execução Paralela 33
3.4.2.1 Diretiva Kernels 33
3.4.2.2 Diretiva Parallel 34
3.4.2.3 Cláusula Reduction 34
3.4.3 Diretivas de Transferência de Dados 35
3.4.4 Chamada de Funções com OpenACC 36
3.4.5 Escolha do Dispositivo Acelerador 37
4 DESEMPENHO DE EXECUÇÕES COM GPU 38
4.1 Desempenho com Aceleradores 38
4.2 Desempenho de OpenACC 40
4.3 Automatização na Busca de Desempenho 44
5 DECISÃO AUTOMATIZADA 48
5.1 Critérios de Apoio à Decisão 48
5.2 Base de Apoio à Decisão 49
5.3 Estimativa de Desempenho e Decisão de Mapeamento 51
5.4 Ferramenta de Decisão 54
6 AVALIAÇÃO DA FERRAMENTA DE TOMADA DE DECISÃO 56
6.1 Experimentos com Decomposição de Gram-Schmidt 57
6.2 Experimentos com a Decomposição LU 58
6.3 Experimentos com a Matriz de Toeplitz 59
6.4 Discussão 60
7 CONCLUSÃO 62
7.1 Trabalhos Futuros 63
REFERÊNCIAS 65
APÊNDICES 71

1 INTRODUÇÃO

O processamento paralelo é uma solução para os problemas impostos pelas limitações físicas das arquiteturas de hardware que, originalmente, tinham como limite a capacidade computacional de um único processador. Em se tratando de processamento destinado à computação científica, em que há predominância de operações matemáticas, diversas soluções baseadas em hardware dedicado também foram propostas ao longo do tempo, com o objetivo de obter melhor desempenho. Dentre elas, pode-se citar os coprocessadores aritméticos, populares na década de 90, que foram um avanço para a época pois possibilitavam obter maior desempenho mantendo-se a mesma CPU. Este tipo de arquitetura obteve grande aceitação e a geração seguinte das CPUs passou a incorporar o coprocessador aritmético, tornando-se desnecessária a utilização de uma placa mãe com suporte a esse componente.

Dessa forma, observa-se que a necessidade de capacidade de processamento aumenta de forma superior à velocidade com que a tecnologia evolui, tornando-se necessária a utilização de arquiteturas que combinam vários dispositivos. Estes, após serem reconhecidos por sua eficiência trabalhando em conjunto, às vezes convergem para a formarem um único dispositivo. Porém, os desafios encontrados para tornar viável estas arquiteturas, que surgem fruto da composição de dispositivos de hardware com natureza diferente, ainda são objetos de pesquisa.

Atualmente, a GPU (*Graphics Processing Unit* – Unidade de Processamento Gráfico) é o dispositivo externo comumente utilizado para acelerar o desempenho de operações matemáticas. Este tipo de dispositivo, que não se restringe a processamento gráfico, tem sido utilizado para compor arquiteturas híbridas em que CPU e GPU cooperam no processamento. Ao contrário da CPU, o processamento na GPU é por natureza em paralelo. Sendo assim, nem todos os programas podem se beneficiar da GPU, somente aqueles que atendem às exigências do processamento em paralelo e as suas restrições.

Há também outros desafios em utilizar-se uma GPU em arquiteturas híbridas, incluindo desde a escolha do fabricante e modelo até a ferramenta de programação. Para além dessas escolhas, surgem novos desafios como, por exemplo, a predição do desempenho de códigos paralelizados neste tipo de arquitetura. De fato, é possível, por exemplo, que um código paralelo obtenha desempenho em uma determinada arquitetura híbrida CPU-GPU e, em outros casos, tenha desempenho inferior ao da execução somente na CPU. Desta forma, surge o problema de identificar estes casos para se poder facilmente escolher onde executar determinado código com

melhor desempenho, isto é, onde mapear as tarefas do código paralelizado.

Seguindo esta linha de pesquisa, no presente trabalho realizou-se experimentos usando *benchmarks* em GPU, em que ficou evidenciado que o processamento paralelo em GPU é uma alternativa viável para obter melhor desempenho, porém em algumas casos o desempenho é inferior ao da execução somente na CPU. Desta forma constata-se o problema de identificar estes casos e escolher onde executar com melhor desempenho.

Ao identificar estes casos, ganha-se tempo de execução do programa e no desenvolvimento da solução mais adequada. Estes fatos constituem exceções que, se não forem tratadas adequadamente, levarão um dado programa paralelo a não ter o desempenho esperado em todos os casos. Sendo assim, a solução deste problema tornaria possível a escolha dos dispositivos de hardware mais adequados, para solução de um dado problema, com maior rapidez e eficiência.

Para tanto, surge o desafio de identificar quando a execução na GPU é viável, determinando as principais variáveis responsáveis pelo desempenho, denominadas em (CHE; SKADRON, 2014) de métricas de desempenho. Neste trabalho, propõe-se uma forma de automatizar etapas deste processo de decisão do mapeamento de tarefas de um dado programa em CPU ou em GPU.

Após realizada essa decisão, a solução será considerada bem sucedida se a hipótese de que a GPU proporciona melhor desempenho para determinados programas for aprimorada para levar em conta as condições que estes programas forem executados, e as variáveis responsáveis pelo desempenho. Levando em conta estas variáveis, o processo de decisão automatizado deverá proporcionar ganho de desempenho se comparado com a execução somente na GPU, sem a tomada de decisão.

1.1 Objetivo do Trabalho

O presente trabalho de mestrado concentrou-se no desenvolvimento de um método de decisão com etapas automatizadas, a fim de auxiliar o desenvolvedor a tomar a seguinte decisão num dado sistema híbrido: *em qual unidade do sistema deve ser mapeada uma determinada tarefa, para que se obtenha o melhor desempenho no hardware disponível*?

1.2 Delimitação do Trabalho

Os programas paralelos alvo deste trabalho devem ser desenvolvidos com o padrão OpenACC, que usa diretivas de compilação para expressar o paralelismo e foi criado para facilitar a programação em sistemas híbridos formados por CPU e GPU. Escolheu-se OpenACC por ser um padrão recente, logo menos utilizado em trabalhos de automatização na busca por desempenho, que ao mesmo tempo oferece facilidades para expressar tarefas paralelas que possam ser executadas em diferentes dispositivos.

A abordagem utilizada neste trabalho é empírica, baseada em observações do desempenho de programas em diferentes configurações e com diferentes parâmetros e dados de entrada. As propostas formuladas não têm por objetivo garantir a melhor decisão de mapeamento, mas sim abreviar, na medida do possível, o processo de decisão. A automatização a que se refere este trabalho não significa operação sem interferência humana, mas sim um processo sistemático em que se associa o conhecimento do especialista (desenvolvedor) a operações repetitivas que podem ser reaproveitadas e realizadas automaticamente.

1.3 Etapas da Pesquisa

O trabalho norteou-se pelas seguintes etapas: estudo, planejamento, desenvolvimento, aplicação e avaliação dos resultados.

- Estudo: Teve seu início nas tentativas de paralelizar sistemas legados e na constatação da necessidade de realizar uma análise prévia da viabilidade de portar um sistema da forma sequencial para a paralela na CPU e GPU. A partir deste ponto procurou-se investigar as variáveis responsáveis pelo desempenho e para quais valores e limites destas variáveis o desempenho da versão paralela era satisfatório. Nesta etapa, analisou-se o *benchmark* EPCC (Edinburgh Parallel Computing Centre). Paralelamente foi realizado um revisão bibliográfica, buscando por trabalhos realizados que fossem relacionados a este tema e que pudessem contribuir para a realização deste trabalho.

- Planejamento: Após concluída a etapa de estudo e identificação das variáveis mais relevantes, inciou-se o projeto, em que foram desenvolvidos a lógica e funções presentes nos fragmentos de código de tomada de decisão que, inseridos no código fonte de um sistema paralelizado, tomam a decisão em tempo de execução entre executar na CPU ou na GPU.

- Desenvolvimento: Tendo sido concluído o planejamento do sistema, passou-se para a fase de implementação, onde foi escolhida a linguagem mais apropriada e escrito o código fonte das funções necessárias.

- Aplicação: Com a implementação concluída, partiu-se para a fase de aplicação, onde utilizou-se diversos programas paralelizados com OpenACC. O código fonte destes programas é de domínio público e são versões de problemas clássicos que conseguem melhorar seu desempenho utilizando na modelagem a abordagem de codificação para execução em paralelo.

- Avaliação dos resultados: Os resultados obtidos na fase de Aplicação foram organizados em tabelas de comparação dos resultados, em que visualiza-se o desempenho da proposta e suas limitações.

1.4 Organização do Texto

O texto está organizado da seguinte forma: no capítulo 2, que faz parte da fundamentação do trabalho, são apresentadas as principais arquiteturas de processamento computacional voltadas para alto desempenho e com influência neste trabalho.

O capítulo 3, que também fundamenta o trabalho, expõe o tema da programação de alto desempenho e as principais ferramentas que implementam esta funcionalidade, com destaque para a ferramenta utilizada neste trabalho: OpenACC.

No capítulo 4 discute-se o desempenho de programas em sistemas dotados de uma ou mais GPUs. Este capítulo é amparado em uma revisão bibliográfica e, adicionalmente, em experimentos realizados nas primeiras etapas desta pesquisa. Também neste capítulo, discute-se trabalhos relacionados que visam automatizar alguma etapa no processo de busca por desempenho.

No capítulo 5 apresenta-se a proposta de decisão automatizada, explicando seus critérios e funcionamento.

No capítulo 6 apresenta-se a avaliação da ferramenta de decisão.

Por fim, no capitulo 7 é sumarizada a contribuição do trabalho, as conclusões e as sugestões para trabalhos futuros.

2 ARQUITETURAS COMPUTACIONAIS DE ALTO DESEMPENHO

Na área de arquiteturas de computadores, a busca por melhoria de desempenho está sempre presente e, para isso, diferentes estratégias foram propostas e materializadas ao longo do tempo (HENNESSY; PATTERSON, 2011). Dentre algumas pode-se citar, por exemplo, o **pipelining**, capaz de acelerar uma sequência de operações com estágios especializados que se sobrepõem no tempo. Outra estratégia que merece destaque é o **processamento vetorial**, em que instruções operam sobre arranjos de dados ao invés de um único item. Uma constante entre diferentes estratégias é o uso de alguma forma de paralelismo, visando o processamento de um maior número de operações por unidade de tempo.

As arquiteturas computacionais paralelas já foram objeto de diferentes classificações e taxonomias (SKILLICORN, 1988). A taxonomia de Flynn (FLYNN, 1972), por exemplo, distingue as arquiteturas pela unicidade ou multiplicidade de fluxos de dados e de instruções. Outra classificação comum se baseia na organização de memória, que pode ser compartilhada ou distribuída entre múltiplas unidades de processamento (HWANG; XU, 1998). Do ponto de vista de mercado, observa-se que algumas arquiteturas paralelas se tornaram amplamente ofertadas por fabricantes de hardware. Como exemplos podemos citar os processadores com múltiplos núcleos (**multicore**), os FPGAs *(Field-programmable Gate Arrays)*, Cell BEA (Cell Broadband Engine Architecture) e as Unidades de Processamento Gráfico (*Graphics Processing Units* – **GPUs**) , também conhecidos como aceleradores (BRODTKORB et al., 2010). Ambos se encontram em uma gama variada de equipamentos, desde dispositivos de uso pessoal até grandes supercomputadores (STROHMAIER et al., 2016). Desta forma podemos citar os supercomputadores Cray (PEDRETTI et al., 2015), o IBM Power-8 com processador multicore que tem seu desempenho comparado as GPUs NVIDIA e Intel Xeon CPUs, utilizado *bechmark* de operações financeiras (REGULY; KEITA; GILES, 2015) e as arquiteturas Convey *Hybrid-Core Computer* , Maxeler *Dataflow Engines (MaxNodes)* e SGI RASC que utilizam dispositivos programaveis FPGA (STOJANOVIC SASA BOJIC DRAGAN, 2012).

Estas arquiteturas, destinam-se a domínios de problemas diferentes, a GPU fornece aceleração quando há grande quantidade de dados, processamento altamente paralelo, comunicação e sincronização podendo ser evitados. O Cell BEA é uma arquitetura flexível e proporciona ao programador a liberdade de núcleos independentes com alta largura de banda para sincronização e comunicação *inter-core*. As soluções baseadas em FPGA são adequadas para grande

volume de dados e cálculos complexos (STOJANOVIC SASA BOJIC DRAGAN, 2012).

No restante deste capítulo, serão apresentadas as principais características do processamento multicore e GPU, dois tipos de arquiteturas para processamento paralelo. Também serão discutidas arquiteturas híbridas, que combinam dois ou mais tipos.

2.1 Processamento Multicore

Arquiteturas *multicore* possuem duas ou mais unidades de execução (*cores* ou núcleos) no interior de um único chip. Os núcleos acessam a memória principal, geralmente via uma interconexão também compartilhada. O sistema operacional trata esses núcleos como se cada um fosse um processador diferente, com seus próprios recursos de execução. Em geral, cada unidade possui sua própria memória cache e pode processar várias instruções simultaneamente.

Um sistema multicore permite maior utilização de paralelismo no nível de threads, sobre tudo para aplicações que façam pouco uso do paralelismo a nível de instrução. Os projetos de software estão cada vez mais utilizando múltiplos processos e threads, em aplicações de multimídia. (OLUKOTUN et al., 1996)

Com múltiplos núcleos de processamento em um processador, as instruções das aplicações podem ser executadas em paralelo em vez de serialmente, como ocorre em um núcleo único. Pelas palavras de um fabricante de computadores: *Adicionar um novo núcleo assemelha-se a abrir uma nova pista em uma estrada para aliviar o trânsito: os carros não precisam dirigir mais rápido para chegarem mais cedo ao seu destino, eles apenas não são atrasados tanto pelo gargalo de poucas pistas e congestionamentos* (DELL, 2014). Esta descrição revela uma questão importante sobre o desempenho nestas arquiteturas: o ganho em relação a um núcleo único depende da fração paralela do software em execução. Ou seja: se o algoritmo utilizado tiver uma fração sequencial significativa, o ganho com múltiplos núcleos será pequeno ou inexistente.

2.2 Processamento em GPU

Uma Unidade de Processamento Gráfico (*Graphics Processing Unit* – GPU) é um microprocessador especializado em operações para criação e transformação de imagens. Com o objetivo de gerar gráficos realistas em tempo real, a GPU alcançou expressivo desempenho com operações de ponto flutuante. Hoje em dia, as GPUs ultrapassam grandemente as CPUs em velocidade de operações aritméticas, quando comunicação e sincronização são evitados,

tornando-se dispositivos ideais para processar e acelerar um variedade de aplicações em paralelo.

Os esforços para explorar a GPU para aplicações não-gráficas se concentraram, inicialmente, em utilizar APIs (*Application Programming Interfaces*) como DirectX e OpenGL para executar operações numéricas em paralelo. Estes esforços que utilizaram APIs gráficas para computação de propósito geral ficaram conhecidos como GPGPU (*General Purpose Programming on Graphics Processing Unit*) (KIRK; HWU, 2012). Esta abordagem permitia alcançar grandes velocidades de processamento, mas trazia alguns inconvenientes. Em primeiro lugar, requeria que o programador tivesse um conhecimento profundo das APIs gráficas e da arquitetura da GPU. Em segundo lugar, os problemas tinham que ser expressos em termos de coordenadas de vértices, texturas e sombreamento, gerando programas com grande complexidade. Em terceiro lugar, programas com caraterísticas básicas tais com leitura aleatória e escrita na memória não eram suportados, tornando este modelo de programação bastante restrito. Finalmente, a falta de suporte à precisão dupla significava que algumas aplicações científicas não poderiam executar na GPU.

A partir de 2007, com o lançamento da plataforma CUDA (*Compute Unified Device Architecture*) pela empresa NVIDIA, teve início uma nova fase nesta área da computação de alto desempenho. Com adições no hardware e uma API para acesso à GPU, a plataforma CUDA se apresentou como uma solução permitindo escrever programas em linguagem C com extensões de propósito geral e massivamente paralelas (NVIDIA, 2014a). Este modo novo de programação GPU, ou computação GPU, significa acesso a uma vasto suporte para as aplicações, garantido por uma linguagem de programação de alto nível, distanciando-se do modelo inicial de programação GPGPU.

No que concerne a arquitetura de modernas GPUs, há uma grande diversidade de esquemas e técnicas empregadas. A figura 2.1 apresenta um esquema simplificado e de alto nível, usando a terminologia da plataforma CUDA. Nesta figura, distingue-se o hospedeiro (*host*) e o dispositivo (*device*), que referem-se respectivamente à CPU e à GPU, ambas com suas memórias. O *host* gerencia sua memória e a da GPU, além de lançar funções para execução em GPU, conhecidas como *kernels*. O dispositivo (GPU) se organiza em *grids*, capazes de executar os *kernels*. Cada *grid* é composto por blocos e cada bloco é composto por *threads*. As *threads* de um mesmo bloco podem compartilhar dados entre si. *Threads* de blocos diferentes não podem compartilhar memória entre si de forma direta. Um *grid* não se comunica com outro *grid*. Os

kernels são executados por várias *threads* em paralelo.

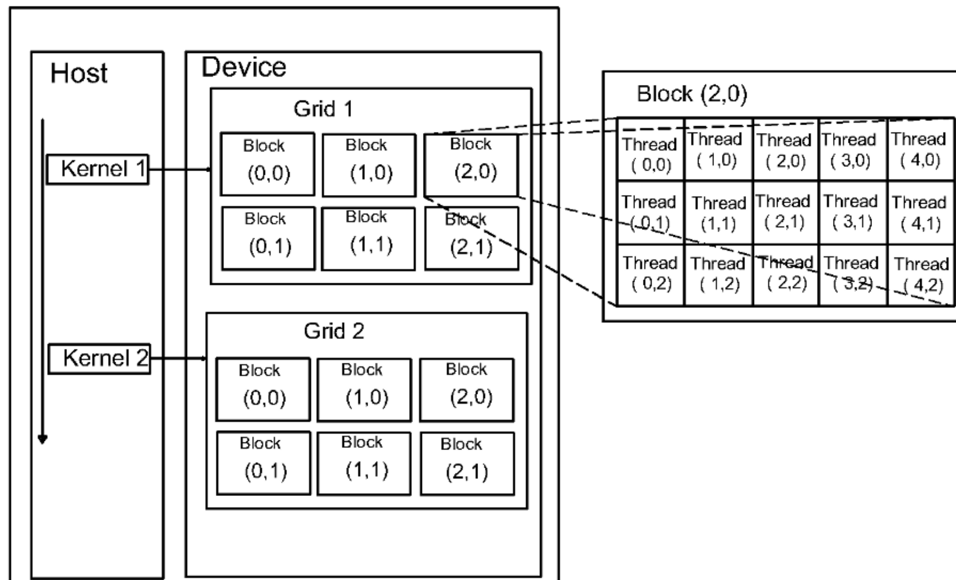


Figura 2.1: Esquema Host Device de uma Unidade de Processamento Gráfico. Fonte: NVIDIA

A figura 2.2 apresenta a disposição física de um *grid* de uma GPU. Cada *thread* no *grid* tem seu próprio registrador e uma área de memória local exclusiva para seu uso. A comunicação entre os blocos é feita através da memória global do *grid*. A memória global (*Global Memory*) da GPU é uma via de comunicação dupla, que serve para enviar e receber dados do *host*, para as *threads* dos blocos. Também pode ser utilizada para a comunicação entre *threads* de blocos diferentes. A memória constante (*Constant Memory*) e a memória de textura (*Texture Memory*) somente enviam dados para as *threads* e recebem e enviam dados para o *host*.

A figura 2.2 mostra que a comunicação entre *threads* de blocos diferentes é possível. No entanto, não é tão eficiente como a comunicação entre *threads* de um mesmo bloco, portanto o desempenho de um sistema é influenciado pelo dimensionamento dos blocos de *threads*. Embora uma GPU seja eficiente no processamento paralelo, a comunicação entre seus componentes é custosa.

2.3 Arquiteturas Paralelas Híbridas

Dada a diversidade de arquiteturas paralelas e a incessante busca por desempenho, é possível combinar duas ou mais arquiteturas, formando um sistema híbrido. Uma das primeiras combinações que se popularizaram na área de computação de alto desempenho foram os *clusters* formados por nós com múltiplos processadores (ALI; SYED, 2013). Essa arquitetura

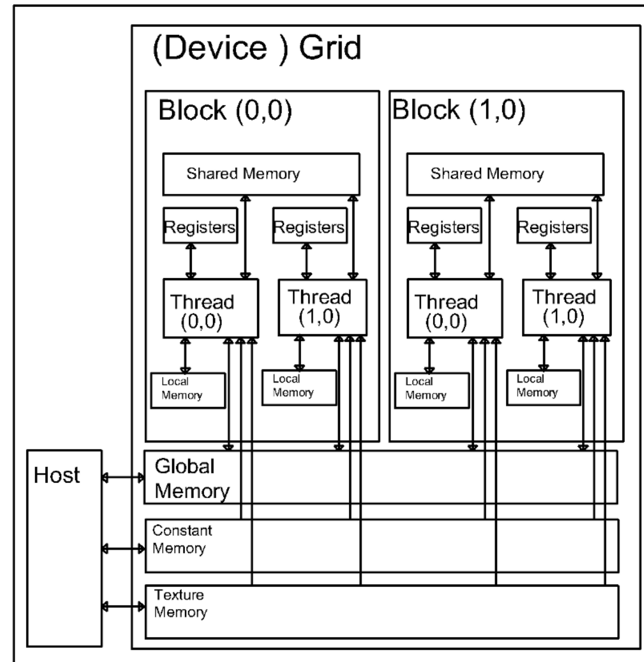


Figura 2.2: Esquema de um Grid de uma Unidade de Processamento Gráfico. Fonte: NVIDIA

paralela híbrida possui, ao mesmo tempo, memória distribuída entre os nós do *cluster* e compartilhada pelos múltiplos processadores em cada nó. Uma evolução natural desta combinação foi o surgimento de *clusters* com processadores *multicore*, mantendo a associação entre memória compartilhada e distribuída. A interconexão de vários nós confere escalabilidade a este tipo de arquitetura, ao mesmo tempo que a memória compartilhada em cada nó garante baixa latência para transferências de dados intra-nó. Do ponto de vista do desenvolvimento de software para este tipo de arquitetura, a interconexão entre os nós e os acessos à memória compartilhada são gargalos e fontes de perdas de desempenho.

Com a popularização de GPUs, surgiram outras oportunidades para o desenvolvimento de arquiteturas paralelas híbridas. Computadores que já utilizavam processadores *multicore* passaram a incorporar uma ou mais GPUs, combinando o poder desses dois tipos de arquiteturas paralelas. Essa combinação permite atingir paralelismo massivo a baixo custo. Do ponto de vista de organização de memória, tem-se memória compartilhada entre os núcleos, sendo que o mesmo ocorre entre *threads* de um mesmo bloco na GPU. Porém, a transferência de dados entre CPU e GPU se dá através de interconexões, o que assemelha-se a uma arquitetura com memória distribuída. Além disso, o processamento paralelo em uma CPU *multicore* difere do processamento que ocorre em GPU (por exemplo, em velocidade e instruções), portanto a divisão do trabalho entre os dois tipos de processadores deve ser criteriosa e não é uma tarefa

trivial. Mesmo assim, esse tipo de arquitetura híbrida tem sido cada vez mais utilizada em computação de alto desempenho (STROHMAIER et al., 2016), sob forma de *clusters* cujos nós são híbridos, por exemplo combinando CPUs e GPUs.

Por outro lado seguido a tecnologia do processamento *multicore* é apresentado o *Intel Many Integrated Core Architecture* ou Intel MIC pela Intel e denominado posteriormente de *Xeon Phi* formando um sistema para processamento em paralelo CPU-MIC no qual integra supercomputadores da lista dos Top 500. Um exemplo da aplicação desta arquitetura é exposto no trabalho de (WU et al., 2013), em que é desenvolvido um esquema de paralelização hierárquica para simulações de dinâmica molecular em sistemas CPU-MIC. O esquema explora paralelismo multi-nível combinando paralelismo no nível de tarefa com o paralelismo no nível de thread empregando decomposição multi-threading programada dinamicamente e paralelismo no nível de dados via tecnologia SIMD. Como resultado é alcançado speedup de até 2,25 no sistema CPU-MIC sobre o sistema de CPU puro, o que supera a aceleração alcançada quando utilizada uma plataforma CPU-GPU (NVIDIA Tesla M2050).

Da mesma forma o trabalho realizado por (INTA; BOWMAN; SCOTT, 2012) utilizando uma arquitetura composta por CPU/GPU/FPGA, aplicada a solução de problemas relacionados a Astronomia, concluí que a configuração de *hardware* utilizada é viável devendo ter o cuidado de evitar os gargalos de desempenho encontrados no transporte de dados pela *PCIe* e evidencia a dificuldade da programação com o dispositivo FPGA demandando maior tempo de programação.

A escolha de uma arquitetura adequada, visando alto desempenho deve ser pautada, por critérios tendo por base o domínio do problema a dificuldade de implementação e manutenção futura, bem como o ganho de desempenho alcançado.

3 PROGRAMAÇÃO DE ALTO DESEMPENHO

As diferenças nas arquiteturas paralelas de alto desempenho têm forte impacto nos modelos, linguagens e ferramentas de programação. De forma geral, o uso de recursos de programação paralela apropriados para cada arquitetura é essencial para se obter o desempenho esperado. Ferramentas que abstraem a arquitetura paralela para o programador (por exemplo, compiladores paralelizadores), costumam ter aplicabilidade e desempenho limitados, em comparação com aquelas que expõem aspectos arquiteturais (KASIM et al., 2008).

Embora exista uma variedade de ferramentas de programação paralela, algumas têm predominado sobre outras. Por exemplo, os padrões **OpenMP** (*Open Multi-Processing*) e **MPI** (*Message Passing Interface*) são, respectivamente, referências para programação paralela em arquiteturas com memória compartilhada ou distribuída (DIAZ; MUNOZ-CARO; NINO, 2012) superando aquelas que implementam a paralelização como uma extensão da linguagem tais como HPF *(High Performance Fortran)* e Co-array Fortran denominado F^{--}. No cenário da programação para GPU, a plataforma CUDA (*Compute Unified Device Architecture*) é considerada a tecnologia mais madura e com mais ferramentas de apoio à programação (BRODTKORB; HAGEN; SÆTRA, 2013), embora possua desvantagens nos quesitos portabilidade e padronização, por tratar-se de uma solução proprietária da empresa NVIDIA. Uma alternativa a CUDA é OpenCL (*Open Computing Language*), um *framework* para programação em plataformas heterogêneas, potencialmente compostas por múltiplas CPUs, GPUs e outros tipos de processadores aceleradores. OpenCL possui interface padronizada e definida por um consórcio (Khronos Group) de empresas e outras entidades interessadas, o que favorece a portabilidade e independência de fabricante. Por outro lado, OpenCL exige mais trabalho do programador na escrita e otimização do código para GPU (SU et al., 2012; PENNYCOOK et al., 2013).

No caso de arquiteturas paralelas híbridas, a programação combinando diferentes modelos e ferramentas pode levar a aumentos consideráveis de desempenho (DIAZ; MUNOZ-CARO; NINO, 2012). Para o programador, no entanto, isso representa um grau de dificuldade a mais, pois diferentes APIs precisam interoperar num mesmo programa. Uma abordagem capaz de mitigar este problema é a programação paralela auxiliada por diretivas de compilação, como ocorre no padrão OpenMP. Essa abordagem é usada em OpenACC (OPENACC, 2014), um padrão voltado para a programação em arquiteturas híbridas que combinam CPUs e GPUs, escolhido para este trabalho. Mesmo com auxílio do compilador, a programação paralela usando

este tipo de ferramenta exige decisões importantes do programador para que o paralelismo da arquitetura seja aproveitado da melhor forma possível para cada aplicação.

3.1 HPF e Co-array Fortran

O HPF *(High Performance Fortran)* é um sistema de programação paralela baseado em Fortran. O esforço para padronizar HPF começou em 1991, na Conferência de Supercomputação em Albuquerque, onde um grupo de líderes da indústria pretendia produzir uma linguagem comum de programação para a classe emergente de computadores paralelos com memória distribuída. A linguagem proposta concentraria em operações de dados em paralelo em um único segmento de controle, uma estratégia que foi iniciada por alguns anteriormente sistemas comerciais tais como CM Fortran, Fortran D, e Viena Fortran.

High Performance Fortran não teve o sucesso esperado por várias razões tais como compiladores com tecnologia imatura conduzindo ao mau desempenho, falta de distribuições flexíveis, implementações inconsistentes, falta ferramentas e de paciência por parte da comunidade de desenvolvedores. Mesmo assim, HPF incorporou uma série de ideias importantes que fez parte da próxima geração de linguagens computação de alto desempenho, incluindo um modelo de execução *single-thread* com um espaço de endereço global, a interoperabilidade com outros modelos de linguagem, e uma extensa biblioteca de operações primitivas para a computação paralela (KENNEDY; KOELBEL; ZIMA, 2007).

O emphCo-array Fortran, conhecido como F^{--}, é uma pequena extensão do Fortran 95 para processamento paralelo. Um programa Fortran emphco-array é replicado várias vezes e todas as cópias são executados de forma assíncrona. Cada cópia tem seu próprio conjunto de objetos de dados denominado imagem. Existem procedimentos intrínsecos para sincronizar imagens, que retornam o número de imagens, e devolvem o índice da imagem atual. (NUMRICH; REID, 1998)

O *Co-array Fortran* juntamente com HPF foram tentativas de tornar a linguagem fortran, através de extensões, um padrão de programação para sistemas paralelos. Embora o desempenho tenho sido aceitável, a utilização de um compilador paralelizador para linguagem Fortran, teve pouca aceitação pela comunidade de desenvolvedores, por motivos inerentes a tecnologia disponível na época de seu lançamento. Mesmo assim muitas ideias desenvolvidas na implementação do HPF foram incorporadas a OpenMP lançado em 1997 que tornou-se então um padrão em programação paralela, difundido e aceito pelo comunidade de desenvolvedores.

O *Co-array Fortran* é um projeto que teve continuidade pela comunidade desenvolvedores de *(*software free) e teve sua API implementada no *Gnu Fortran (Gfortran)*, em trabalho realizado por (FANFARILLO et al., 2014) é apresentado experimentos utilizando *benchmarks* que concluem quem em alguns casos o *Co-array Fortran* tem desempenho superior a implementação com MPI.

3.2 OpenMP

O padrão OpenMP (DAGUM; MENON, 1998; OpenMP.org, 2015) provê um modelo de programação paralela voltado para arquiteturas com memória compartilhada, para programas em linguagens Fortran e C/C++. A programação baseia-se essencialmente na inserção de diretivas de compilação com cláusulas e opções que guiam a geração do código paralelo pelo compilador. Essas diretivas especificam regiões paralelas, divisões de trabalho, sincronização e acesso aos dados. Trata-se de um modelo adequado para paralelização incremental de aplicações existentes, pois as diretivas não alteram significativamente o código original. Programas com OpenMP são portáveis entre arquiteturas de hardware, somente dependendo do suporte pelo compilador.

Programas em OpenMP se baseiam na criação de *threads* seguindo um modelo *fork-join*. Neste modelo, uma *thread* mestre é executada sequencialmente até encontrar uma região paralela, quando então é criado (*fork*) um grupo de *threads* que executam em paralelo. Quando todas as *threads* terminam a execução da região paralela, o fluxo de execução segue somente na *thread* mestre (*join*).

A paralelização com OpenMP emprega diretivas iniciando por `#pragma omp`. Caso o compilador não suporte a especificação OpenMP, as diretivas são ignoradas. A forma geral das diretivas obedece à seguinte sintaxe em C/C++:

```
#pragma omp <nome da diretiva> [cláusulas]
```

Exemplos de diretivas são: `parallel`, `for`, `section`, `single`, `master`, `critical` e `atomic`, que especificam como é a interação entre as *threads* e a sincronização no acesso aos dados. As cláusulas são responsáveis por alterar o comportamento das diretivas e são opcionais.

Uma das diretiva mais importantes é a `parallel`, que é responsável pela criação da região paralela. A região é um trecho de código delimitado ou, mais comumente, um bloco de repetição `for`. Dentre as cláusulas que podem ser adicionadas às diretivas, uma das mais importantes é a `reduction`, que permite efetuar operações aritméticas com os valores coletados

entre todas as *threads* e enviar um único resultado (redução) para a *thread* mestre.

Desde sua criação, o padrão OpenMP foi implementado por uma variedade de compiladores, tanto proprietários como de código aberto, o que contribuiu para sua popularização na área de programação de alto desempenho. O padrão passou por várias atualizações desde sua especificação inicial em 1997. A partir da versão 4.0, publicada em meados de 2013, o padrão passou a contemplar alguns recursos para utilização de GPUs. No entanto, até a presente data, há poucos compiladores que suportam essa versão de OpenMP. Além disso, cláusulas importantes, como `reduction` por exemplo, ainda não são implementadas com suporte a GPU.

No que diz respeito ao desempenho de aplicações paralelizadas com OpenMP, sabe-se que existe um *overhead* associado, por exemplo, à criação e sincronização de *threads* (FREDRICKSON; AFSAHI; QIAN, 2003; FÜRLINGER; GERNDT, 2007). O desenvolvedor deve, portanto, levar isso em conta ao tomar suas decisões na paralelização de uma aplicação, de modo que o ganho nas regiões paralelas suplante o *overhead* da gestão do paralelismo. Uma vez analisada a viabilidade da execução em paralelo e escolhidas as abordagens de divisão do trabalho, sincronização e acesso aos dados, a codificação em OpenMP demanda pouco esforço de programação se comparada, por exemplo, com Pthreads para arquiteturas *multicore* ou CUDA para programação em GPUs.

3.3 CUDA

A plataforma CUDA foi projetada pela empresa NVIDIA de forma a facilitar a exploração do paralelismo *many-core* das GPUs modernas (NVIDIA, 2014b). Seu suporte à programação paralela compreende essencialmente um compilador (NVCC) e bibliotecas que implementam uma API acessível originalmente em linguagens C/C++ e Fortran. Esses recursos de programação manipulam elementos da arquitetura CUDA apresentados na seção 2.2.

Do ponto de vista do desenvolvedor, as principais abstrações do modelo de programação CUDA são: uma hierarquia de grupos de *threads*, compartilhamento da memória e uma barreira para sincronização, que são acessíveis ao programador como extensões da linguagem (COOK, 2013). Em termos de organização da aplicação paralela, o código (*kernel*) a ser executado em GPU reside em um arquivo separado, com extensão `.cu`. O código que executa no *host*, escrito em linguagem C/C++ pura, é responsável por lançar a execução do *kernel* e controlar transferências de dados de/para a memória da GPU, de forma síncrona ou assíncrona.

A plataforma CUDA proporciona escalabilidade transparente, pois um *kernel* compilado

consegue executar em qualquer número de elementos de processamento da GPU (NVIDIA, 2014b). Ao desenvolvedor de aplicações paralelas, no entanto, cabem ainda vários tipos de decisão, desde a identificação das tarefas (funções ou (*kernels*, na terminologia CUDA) que devem ser mapeadas na GPU, até a organização dos dados na hierarquia de memória para acesso das *threads*, que são organizadas em blocos e *grids*. Comparativamente com um programa OpenMP, por exemplo, a programação em CUDA exige mais linhas de código e uma menor abstração de detalhes da arquitetura subjacente.

No apoio ao desenvolvedor, a NVIDIA oferece um conjunto de ferramentas adicionais, como depurador e *profiler*, que auxiliam em tarefas de programação. Essas vantagens, assim como a própria plataforma CUDA, são acessíveis a dispositivos de um único fabricante: NVIDIA. Como o mercado de GPUs e aceleradores também é disputado por outros fabricantes (por exemplo, Intel e AMD), essa limitação da plataforma CUDA pode se tornar proibitiva. Isso pode ser especialmente limitante em aplicações de grande porte, em que o esforço de paralelização consome muitos recursos humanos e, idealmente, não deveria ser reiniciado com a substituição do dispositivo acelerador (GPU) por outro mais recente de outro fabricante.

3.4 OpenACC

OpenACC é um padrão para programação paralela originalmente proposto em 2011, em uma parceria entre NVIDIA, Cray Inc., Portland Group (PGI), e CAPS Enterprise. Criado para facilitar a programação em sistemas híbridos formados por CPU/GPU, esse padrão se utiliza de diretivas de programação para expressar o paralelismo, assim como ocorre em OpenMP. Usando essa abordagem, OpenACC permite desenvolver programas paralelos portáveis em diferentes plataformas de hardware, incluindo sistemas *host*-GPU e CPUs *multicore*. Assim como em OpenMP, esse modelo de programação requer suporte do compilador, que reconhece as diretivas e gera código com otimizações para diferentes arquiteturas.

A API de programação do OpenACC descreve uma coleção de diretivas de compilação que especificam laços e regiões de código em C/C++ ou Fortran, que serão executados em um sistema composto por um *host* (CPU) e um dispositivo acelerador (GPU). Estas diretivas permitem criar programas de alto nível que implementam interações entre *host* e acelerador, sem que seja necessário inicializar explicitamente o acelerador, gerenciar dados e transferir programas entre o *host* e o acelerador. Todos estes detalhes são implícitos no modelo de programação e são gerenciados pelo compilador com suporte a OpenACC e seu sistema de execução. Para

oferecer também um controle mais fino ao programador, o modelo de programação permite que o programa envie informações ao compilador, incluindo especificações de dados locais para o acelerador, orientações sobre o mapeamento de laços e detalhes sobre o desempenho do dispositivo acelerador (OPENACC, 2014).

Na figura 3.1 tem-se um exemplo de código com aceleração em GPU utilizando OpenACC. Observa-se, na linha 3, uma das diretivas do OpenACC: a diretiva `kernels`, que será explicada mais adiante. A maneira como esta e outras diretivas são traduzidas para código paralelo é específica para cada compilador.

```
1      while ( error > tol && iter < iter_max ) {
2        error = 0.0;
3        #pragma acc kernels
4        {
5          for( int j = 1; j < n-1; j++) {
6            for( int i = 1; i < m-1; i++ ) {
7              Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
8                                  + A[j-1][i] + A[j+1][i]);
9              error = fmax( error, fabs(A[j][i] - Anew[j][i]));
10           }
11         }
12
13         for( int j = 1; j < n-1; j++) {
14           for( int i = 1; i < m-1; i++ ) {
15             A[j][i] = Anew[j][i];
16           }
17         }
18       }
19
20       if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
21       iter++;
22     }
```

Figura 3.1: Exemplo de estrutura básica de paralelização com OpenACC, com aplicação da diretiva `kernels`. Fonte: (LARKIN, 2015)

Uma das principais características do OpenACC é a facilidade de uso e a rapidez com que a implementação paralela na GPU pode ser realizada. No exemplo da figura 3.1, bastou acrescentar a diretiva `kernels` para a aceleração na GPU ser realizada.

3.4.1 Níveis de Paralelismo

A arquitetura das modernas GPUs pode ser vista como uma coleção de elementos de processamento, em que cada elemento é *multithread* e as *threads* executam instruções sobre um vetor em paralelo. Isso caracteriza 3 níveis de paralelismo com diferentes granularidades:

(a) um nível de granularidade mais grossa, em que tarefas podem ser mapeadas a diferentes elementos de processamento; (b) um nível de granularidade mais fina, caracterizado pelas múltiplas *threads* num único elemento e (c) operações vetoriais em paralelo, em um único elemento de processamento.

Em OpenACC, esses três níveis de processamento fazem parte do modelo de execução e devem ser conhecidos pelo desenvolvedor. Usando a terminologia OpenACC, os elementos de processamento são divididos em *gangs*, os *gangs* em *workers* e os *workers* em *vectors*. As diretivas que expressam regiões paralelas podem ser acrescidas de cláusulas que configuram *gangs*, *workers* e *vectors*, com parâmetros para dimensionar as divisões do processamento. Os valores ideais, ou seja, aqueles que proporcionam o melhor desempenho, variam de acordo com a arquitetura e configuração do dispositivo acelerador. Tendo em vista a importância disto, há compiladores com suporte a OpenACC, como os da linha PGI Accelerator (PGI, 2016), que apresentam valores padrões destes níveis de paralelismo com desempenho otimizado, liberando o programador de explicitar as cláusulas e seus parâmetros.

3.4.2 Diretivas de Execução Paralela

De forma semelhante ao padrão OpenMP, a forma geral das diretivas em OpenACC obedece à seguinte sintaxe em C/C++:

```
#pragma acc <nome da diretiva> [cláusulas]
```

Existem duas diretivas principais, que expressam execuções paralelas. Uma é a diretiva `kernels` e a outra é a diretiva `parallel`.

3.4.2.1 Diretiva Kernels

Esta diretiva define uma região do programa que deve ser compilada como uma sequência de *kernels* (como em CUDA), para execução no dispositivo acelerador (GPU). Tipicamente, cada laço aninhado dentro da região será convertido em um *kernel* distinto.

Do ponto de vista do compilador, há três passos para este processo de conversão (WOLF, 2012). O primeiro é identificar os laços que podem ser executados em paralelo, o segundo é mapear o paralelismo do laço no hardware paralelo concreto, e o terceiro é a geração de código otimizado para implementar o paralelismo.

A etapa de mapeamento é altamente dependente da arquitetura do acelerador. No caso de uma GPU NVIDIA da plataforma CUDA, um laço paralelo pode ser mapeado, por exemplo,

sobre os blocos de um *grid* (nível de paralelismo gang em OpenACC) ou sobre as *threads* de um bloco (paralelismo vector em OpenACC). O programador, no entanto, não precisa especificar que um dado laço deverá ser mapeado em nível de gang, worker ou vector, pois o compilador tomará essa decisão usando dados da arquitetura e de análise do código. Se desejar controlar manualmente o mapeamento do laço aos níveis de paralelismo, o programador pode acrescentar cláusulas à diretiva kernels, como por exemplo num_gangs, num_workers ou vector_length.

3.4.2.2 Diretiva Parallel

A diretiva parallel se assemelha à diretiva de mesmo nome que se encontra no padrão OpenMP (ver seção 3.2). Em OpenACC, essa diretiva cria um número de *gangs* paralelos que imediatamente começam a executar a região de código redundantemente. Quando o *gang* alcança um laço com alguma cláusula de divisão de trabalho, o *gang* executará um subconjunto das iterações do laço, dependendo da política de escalonamento. Uma diferença em relação a um bloco parallel em OpenMP é que, em OpenACC, não existe sincronização de barreira no final do laço com divisão de trabalho.

Os construtores kernels e parallel resolvem basicamente o mesmo problema, identificando o paralelismo nos laços e mapeando-os para o paralelismo de hardware. No entanto, o construtor kernels deixa as opções mais implícitas, dando ao compilador a liberdade para encontrar e mapear o paralelismo de acordo com os requisitos do acelerador alvo. O construtor parallel, por sua vez, é mais explícito e requer uma análise mais profunda do programador para determinar o que é melhor e mais apropriado (WOLF, 2012).

3.4.2.3 Cláusula Reduction

Assim como em OpenMP, o padrão OpenACC possui a cláusula reduction, que combina valores parciais calculados em paralelo para uma mesma variável, produzindo um único valor final. Em OpenACC, esta cláusula é permitida no interior de uma região demarcada com a diretiva parallel.

Em compiladores da linha PGI Accelerator, reduções são reconhecidas e tratadas automaticamente. Esta funcionalidade é exemplificada na figura 3.2, que ilustra a saída do compilador para um código que não usa a cláusula reduction. Nota-se, na linha 64, que a redução é gerada automaticamente para a variável error usando a operação max.

```
1
2   $ pgcc -acc -ta=tesla -Minfo=accel laplace2d-kernels.c
3   main:
4        56, Generating copyout(Anew[1:4094][1:4094])
5            Generating copyin(A[:][:])
6            Generating copyout(A[1:4094][1:4094])
7            Generating Tesla code
8        58, Loop is parallelizable
9        60, Loop is parallelizable
10           Accelerator kernel generated
11           58, #pragma acc loop gang /* blockIdx.y */
12           60, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x
                 */
13           64, Max reduction generated for error
14       68, Loop is parallelizable
15       70, Loop is parallelizable
16           Accelerator kernel generated
17           68, #pragma acc loop gang /* blockIdx.y */
18           70, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x
                 */
```

Figura 3.2: Resultado da compilação do trecho apresentado na figura 3.1 com o PGI Accelerator. Fonte: (LARKIN, 2015)

3.4.3 Diretivas de Transferência de Dados

Quando o objetivo é reduzir o tempo de execução, deve-se dedicar especial atenção às operações de transporte de dados entre dispositivos, que podem causar *overhead* significativo. Para gerenciar este tipo de operação, o OpenACC apresenta a diretiva `data` com suas cláusulas `copy`, `copyin`, `copyout` e `create`. A diretiva `data` representa um ajuste fino no código acelerado na GPU, com grande influência no desempenho do trecho paralelizado.

A cláusula `copy`, apresentada na figura 3.3, faz o transporte de dados entre CPU e GPU retornando os valores para CPU. Quando não há necessidade de retornar os dados para a CPU, deve-se usar `copyin`. Da mesma forma, quando a variável é criada na GPU e somente envia dados para CPU utiliza-se `copyout`. A diretiva mais eficiente em termos de tempo de execução no tratamento de dados é a `create`, utilizada para variáveis auxiliares quando toda a memória é alocada na GPU e não necessita retornar para a CPU, sendo também a mais limitada pois não permite utilizar seus dados na CPU. À primeira vista parece não ter importância esta limitação, mas tendo em vista que a GPU não faz operações de I/O, os dados que não retornam para a CPU não poderão ser impressos ou gravados, somente existirão na GPU enquanto esta estiver realizando o processamento.

Outro recurso para gerenciar o transporte de dados entre CPU e GPU é a cláusula

```
#pragma acc data copy(A, Anew)
while ( error > tol && iter < iter_max )
{
  // Codigo dentro do loop while nao muda
}
```

Figura 3.3: Exemplo de aplicação da diretiva `data`.

`present`, usada em conjunto com as diretivas `kernels` ou `parallel`. Esta cláusula informa ao compilador que os dados já estão presentes neste dispositivo, conforme ilustrado na figura 3.4. Isto é útil pelo fato de que o movimento de dados entre o *host* (CPU) e o acelerador (GPU) é uma operação custosa em termos de tempo de execução e, frequentemente, é considerado como um gargalo de desempenho. Esta cláusula deve ser usada com cuidado, pois caso os dados não estejam presentes no acelerador, ou estejam parcialmente presentes, um erro em tempo de execução será lançado.

```
#pragma acc data copy( a[0:n] )
{
  init( a, n );
  process( a, n );
}
...
void init( float* a, int n ) {
  #pragma acc kernels loop present(a[0:n])
  for ( int i = 0; i < n; ++i ) a[i] = sinf((float)i);
}
```

Figura 3.4: Exemplo de uso da cláusula `present`. Fonte: (WOLF, 2012)

3.4.4 Chamada de Funções com OpenACC

A eficiência de um programa paralelizado com OpenACC está relacionada à determinação correta dos trechos de código que podem beneficiar-se com a execução na GPU. À primeira vista este parece um conceito simples de ser implementado, mas esta definição impõe restrições para chamadas de códigos que estão fora desta região. Tal limitação somente foi contornada na versão 2.0 do OpenACC, que permite a inclusão de chamadas a funções que estão fora da região paralela, mas com a condição que a chamada desta função seja inserida em um trecho que realize o tratamento dos dados em OpenACC. Desta forma, o compilador identifica que este código deve ser executado na região paralela.

Na figura 3.5, encontra-se um exemplo com tratamento de dados usando a diretiva

`data`, na linha 1. Na linha 3, tem-se a chamada de uma função que encontra-se fora da região paralela. Tal funcionalidade torna possível a inclusão de bibliotecas com funções OpenACC externas ao código fonte.

```
1  #pragma acc data copy( a[0:n] )
2  {
3    funcao( a, n );
4  }
5  // ...
6  void funcao( float* a, int n )
7  {
8    #pragma acc kernels loop present(a[0:n])
9    for( int i = 0; i < n; i++ )
10     a[i] = sin((float)i)*sin((float)i)+cos((float)i)*cos((float)i);
11 }
```

Figura 3.5: Estrutura básica de paralelização com OpenACC e chamada de uma função que encontra-se fora da região paralela.

3.4.5 Escolha do Dispositivo Acelerador

O padrão OpenACC admite que possa existir mais de um dispositivo acelerador no sistema, como de fato se encontra no mercado atualmente. Assim, existem alguns recursos em OpenACC para especificar o dispositivo que se deseja utilizar, sobre o qual terão efeito as operações afetadas por diretivas. É importante notar, portanto, que OpenACC **não trata** da divisão do trabalho e do mapeamento de tarefas **entre** múltiplos dispositivos.

Os recursos para especificação de dispositivos compreendem variáveis de ambiente e funções da biblioteca do OpenACC. A variável de ambiente `ACC_DEVICE_TYPE` especifica o tipo *default* de dispositivo acelerador, supondo que o programa tenha sido compilado para mais de um tipo de dispositivo. Os valores possíveis para esta variável são definidos pelo compilador e não são especificados no padrão. Alguns exemplos de valores são: para GPU Nvidia `nvidia`, GPU AMD `radeon` ou CPU *manycore* `xeonphi`. A variável `ACC_DEVICE_NUM` permite escolher o dispositivo com o qual se quer trabalhar, usando um identificador numérico que inicia em zero e é limitado pelo número de dispositivos conectados.

A biblioteca do OpenACC oferece várias funções, distribuídas em 4 categorias: gerenciamento dos dispositivos, sincronização de execuções, teste do dispositivo e gerenciamento de dados e memória. Para escolha do dispositivo, há funções com propósito semelhante às variáveis de ambiente recém descritas: `acc_set_device_type` e `acc_set_device_num`, respectivamente equivalentes a `ACC_DEVICE_TYPE` e `ACC_DEVICE_NUM`.

4 DESEMPENHO DE EXECUÇÕES COM GPU

Desde a disponibilização dos primeiros modelos de modernas GPUs, muitos trabalhos têm se voltado a investigar o desempenho de diferentes aplicações que exploram o paralelismo desses dispositivos. Uma grande parte dos trabalhos se concentra na plataforma CUDA e, em muitos casos, pesquisadores e fabricantes de hardware têm divulgado *speedups* significativos (por exemplo, de 300 a 1000) para certas aplicações em GPU, comparativamente às mesmas aplicações executando somente em CPU (CHE et al., 2008; RYOO et al., 2008). Aplicações que empregam ferramentas mais recentes, como OpenACC, também têm sido alvo de estudos de desempenho, embora em menor proporção, à medida que alguns compiladores começaram a suportar esse padrão (WIENKE et al., 2012).

Mesmo com um grande potencial de aceleração, obter desempenho em GPU não é garantido e pode exigir um grande esforço de implementação e otimização. Assim, há linhas de investigação que buscam reduzir este esforço por meio de avaliações, decisões e/ou ações automatizadas. Nestas linhas, encontram-se pesquisas que se relacionam, em alguns aspectos, ao presente trabalho de mestrado.

Neste capítulo, discute-se inicialmente o desempenho com aceleradores (seção 4.1), mostrando que há classes de aplicações em que não se consegue obter *speedups* tão significativos como os citados acima. Depois, na seção 4.2, discute-se o desempenho de OpenACC, a ferramenta alvo desta dissertação. Essa discussão tem como base uma revisão bibliográfica e, também, experimentos realizados durante uma fase inicial do presente trabalho. Por fim, na seção 4.3, apresenta-se uma revisão de trabalhos que visam abreviar alguma etapa da busca por desempenho, por meio de algum tipo de automatização.

4.1 Desempenho com Aceleradores

Quando da concepção de uma aplicação de alto desempenho, é fundamental a utilização de um algoritmo eficiente, que resolva o problema proposto com o menor número de operações, tudo isso antes de pensar nos recursos computacionais necessários. Nesta seção, toma-se como referência o problema da solução de sistemas lineares, que pode ser resolvido por métodos clássicos, já validados pela comunidade científica. A partir disso, foram estudados alguns trabalhos que investigaram o desempenho desses métodos em arquiteturas híbridas com aceleradores.

Em (VOLKOV; DEMMEL, 2008) é apresentado é um estudo comparativo dos algorit-

mos para a solução de sistemas lineares pelo método de Cholesky, LU e QR frente a diversos dispositivos GPUs junto a CPUs Core2 Duo e Core2 Quad. O método mais eficiente foi de Cholesky, que obteve um *speedup* de 7,4 na GPU 8800GTX com a CPU Core2 duo e 5,5 na CPU Core2 Quad. Apesar da GPU 8800GTX ter frequência de 1,35 GHZ, inferior ao da GPU 8600GTS de 1,458 GHz, ela possui uma largura de banda (*bandwidth*) de 86 GB/s enquanto que a 8600GTS possui apenas 32 GB/s. O trabalho revela a dificuldade em avaliar o desempenho de uma GPU através do *speedup* que, classicamente, relaciona os tempos de execução sequencial e paralelo. De fato, fica evidente que a queda do *speedup* de 7.4 para 5.5 foi devido à maior capacidade da CPU Core2 Quad frente à Core2 Duo.

Em (YANG et al., 2010) é apresentado o método de Cholesky, com complexidade N^3 quando a decomposição da matriz é LL^t e N^3/3 quando a decomposição tem a forma LDL^t, sendo D uma matriz diagonal. Neste trabalho, é realizado um estudo comparativo entre um dispositivo programável FPGA e uma GPU. O artigo conclui que o FPGA executa em ciclos menores e é vantajoso para matrizes de pequeno porte, enquanto que a GPU e apresenta melhor desempenho com matrizes maiores. Ambos os dispositivos são conhecidos como aceleradores, mas o trabalho deixa evidente que, sob algumas condições, a aceleração pode não ser obtida.

Em (RENNICH; STOSIC; DAVIS, 2014) é apresentado um método de fatoração da matriz utilizada na decomposição de Cholesky, onde a ideia central é transmitir ramos da árvore de eliminação (sub-árvores que terminam em folhas) através da GPU e realizar a fatoração de cada ramo inteiramente na GPU. Isto evita a maioria da comunicação pelo barramento PCIe, que é um gargalo na busca por desempenho. O estudo conclui que é relativamente fácil atingir a aceleração na GPU para matrizes de ordem elevada, maior quantidade de zeros em relação as de menor ordem, mas é difícil atingir *speedups* significativos quando as matrizes possuem um índice de preenchimento com zeros menor. Uma contribuição deste trabalho está em discutir as principais métricas que têm influência no desempenho da GPU para o problema em questão, evidenciando também a preocupação com a largura de banda, que representa *overhead* quando o processamento é em GPU.

Em todos os trabalhos analisados, nota-se que obter desempenho com aceleradores pode não ser uma tarefa trivial. Ao identificar fatores que influenciam no desempenho de uma classe de aplicações, estes trabalhos contribuem para que, no futuro, tarefas de otimização e configuração possam ter etapas automatizadas, permitindo abreviar o esforço na busca por desempenho.

4.2 Desempenho de OpenACC

Com a publicação do padrão OpenACC e suas primeiras implementações em compiladores, alguns autores se dedicaram a avaliar seu desempenho e compará-lo com outras alternativas. Por exemplo, em (HOSHINO et al., 2013), os autores comparam o desempenho de CUDA e OpenACC em *kernels* e aplicações na área de dinâmica de fluidos computacional. Os resultados mostram que, no geral, o desempenho de OpenACC fica em torno de 50% mais baixo que CUDA para as aplicações consideradas, embora otimizações manuais permitam chegar até 98%.

Em outro trabalho (XU et al., 2014), os autores comparam implementações do *benchmark* paralelo NAS (BAILEY et al., 1991) em CUDA, OpenCL e OpenACC. Os resultados mostram uma diversidade de comportamentos de desempenho, com predominância de menores tempos de processamento com CUDA, embora em muitos casos a aceleração (*speedup*) com OpenACC fique bastante próxima, na ordem de 70% a 96% da aceleração obtida com CUDA. Os autores ressaltam vantagens da programação baseada em diretivas, incluindo a facilidade de paralelização e a possibilidade manter um único código que possa ser executado em múltiplas plataformas.

Nos trabalhos citados, fica evidente a aceleração em relação à execução sequencial dos códigos. No entanto, há trabalhos que revelam que o uso de aceleradores e OpenACC não resultou em *speedup* em alguns casos (HAGERNÄS; ANDRÉN, 2013; LEPPER, 2015). De fato, isso é possível devido a vários motivos relacionados a características da aplicação e do hardware considerado.

Visando aprofundar esta questão de desempenho, no início deste trabalho de mestrado foram feitos experimentos com um *benchmark* para OpenACC. Escolheu-se o EPCC OpenACC Benchmark Suite (JOHNSON, 2013), que à época era o único *benchmark* disponível para OpenACC e que, atualmente, continua sendo uma alternativa de código aberto para o SPEC ACCEL (JUCKELAND et al., 2014), um *benchmark* mais recente, com código fechado.

O conjunto de *benchmarks* EPCC é dividido em três partes: Nível 0, Nível 1 e Aplicações. No Nível 0 (Level 0), há 15 *micro-benchmarks* que têm por objetivo mensurar a velocidade de operação de diferentes diretivas OpenACC: ContigH2D, ContigD2H, SlicedD2H, SlicedH2D, Kernels_If, Parallel_If, Parallel_private, Parallel_1stprivate, Kernels_combined, Parallel_combined, Update_host, Kernels_Invocation, Parallel_Invocation, Parallel_Reduction,

Kernels_reduction. No Nível 1 (Level 1) tem-se 13 *benchmarks* contendo *kernels* de álgebra linear do BLAS (*Basic Linear Algebra Subprograms*), extraídos e portados para OpenACC a partir do código de outro conjunto de *benchmarks* (PolyBench e PolyBench/GPU). São eles: 2MM, 3MM, ATAX, BICG, MVT, SYRK, COV, COR, SYR2K, GESUMMV, GEMM, 2DCONV, 3DCONV. No nível Aplicações, tem-se 3 diferentes códigos de aplicações reais utilizadas em programas de computação científica: HIMENO, 27stencil e LE_core.

Os *benchmarks* do EPCC foram executados em um servidor com uma CPU Intel Xeon E5620 2.4GHz com 8 cores e uma GPU NVIDIA Tesla M2050, usando configurações *default* dos programas (*datasize* de 1048576 bytes e 10 repetições). O compilador utilizado foi o PGI Accelerator para linguagem C e sistema operacional Linux. Os resultados obtidos encontram-se na tabela 4.1, que apresenta os tempos de execução sequencial (somente em CPU), execução paralela com OpenACC (em GPU) e o *speedup* calculado pela divisão do tempo de execução sequencial pelo tempo de execução em paralelo. Conforme documentação incluída nos *benchmarks*, alguns deles poderiam apresentar tempos negativos por serem expressos como uma média de diferenças de tempos. Optou-se, nestes casos, por excluí-los da tabela para não confundir o leitor, já que os demais *benchmarks* foram suficientes para visualizar uma diversidade de *speedups*.

A tabela 4.1 está ordenada do maior ao menor *speedup* e, analisando-se todos os resultados, pode-se perceber que há vários casos em que o tempo de execução com OpenACC é menor que o de execução sequencial, resultando em *speedup* maior que 1. No entanto, observa-se também *speedups* menores que 1, revelando casos em que a execução sequencial obteve menor tempo de execução. Esses experimentos, embora não exaustivos, ilustram uma realidade que é pouco discutida: para algumas combinações de aplicações, modelo de programação, dados de entrada e hardware, uma execução em CPU podem ser mais eficiente do que em GPU, contrariando a expectativa amplamente alimentada pelos fabricantes de GPUs.

Considerando que as arquiteturas paralelas modernas podem ser híbridas, com paralelismo presente na CPU e na GPU, foi também realizado um outro conjunto de experimentos, desta vez com um código que implementa duas versões de uma mesma função: uma com OpenACC e outra com OpenMP. O código em questão implementa o método de Gram-Schmidt, um método clássico de fatorização que produz uma matriz ortogonal por meio de operações sucessivas com matrizes triangulares. Conforme o autor do programa (FARBER, 2012), a implementação paralela eficiente das operações envolvidas não é trivial em GPU, pois a carga de

Funções EPCC	Tempo Sequencial	Tempo OpenACC	Speedup
Kernels_Invocation	873,827934	29,230118	29,894
GEMM	43812,8233	2263,998985	19,351
3MM	31592,48829	2393,698692	13,198
Parallel_Invocation	875,353813	116,181374	7,534
SYRK	38024,44935	6143,426895	6,189
27stencil	25897,43137	4709,219933	5,499
SYR2K	25322,03198	6112,980843	4,142
COR	46105,88551	12453,19843	3,702
COV	44451,78509	12312,74605	3,610
2MM	34360,50415	10828,44734	3,173
HIMENO	425725,913	428155,9944	0,994
ATAX	1085,495949	1775,598526	0,611
BICG	1082,68261	1948,904991	0,555
MVT	1083,803177	2009,081841	0,539
2DCONV	845,193863	1899,147034	0,445
3DCONV	1805,615425	5294,370651	0,341
GESUMMV	273,58532	2514,266968	0,108
SlicedD2H	0,953674	302,243233	0,003
ContigH2D	0,953674	117991,1852	8,082E-006

Tabela 4.1: Benchmark EPCC, com datasize default de 1048576 bytes, 10 repetições e tempo em microsegundos.

trabalho pode variar internamente aos laços de repetição.

Os experimentos foram realizados em um computador com processador AMD 2.4 GHz e GPU Nvidia GTX 550. O compilador utilizado foi o PGI Accelerator para linguagem C, que possui suporte ao OpenMP e ao OpenACC. Foram considerados três casos de execução: sequencial na CPU, em paralelo com 8 *threads* na CPU, utilizando OpenMP, e em paralelo na GPU utilizando OpenACC. Os resultados encontram-se na tabela 4.2, que apresenta tempos e *speedups* obtidos em cada caso.

Processo de Gram-Schmidt - Tempo em seg.)					
Ord. Matriz	Seq.	OpenMP	speedup.	OpenACC	speedup
100	0,002	0,003	0,67	0,449	0,00
200	0,018	0,010	1,80	0,445	0,04
300	0,061	0,020	3,05	0,454	0,13
400	0,144	0,034	4,24	0,480	0,30
500	0,280	0,059	4,75	0,491	0,57
600	0,481	0,093	5,17	0,523	0,92
700	0,763	0,140	5,45	0,555	1,37
800	1,143	0,200	5,72	0,599	1,91
900	1,625	0,281	5,78	0,655	2,48
1000	2,230	0,378	5,90	0,729	3,06
1100	2,984	0,500	5,97	0,800	3,73
1200	3,888	0,648	6,00	0,886	4,39
1300	4,942	0,811	6,09	1,000	4,94
1400	6,171	1,000	6,17	1,108	5,57
1500	7,580	1,234	6,14	1,251	6,06
1600	9,230	1,529	6,04	1,388	6,65
1700	11,064	1,800	6,15	1,632	6,78
1800	13,140	2,120	6,20	1,834	7,16
1900	15,460	2,499	6,19	2,065	7,49
2000	18,079	2,920	6,19	2,316	7,81
3000	61,266	10,460	5,86	6,751	9,08
3500	97,197	16,760	5,80	10,388	9,36

Tabela 4.2: O Processo de Gram-Schmidt, Execução Sequencial, Paralelo com OpenMP com 8 threads, Paralelo na GPU com OpenACC com processador AMD 2.4 GHz e a GPU Nvidia GTX 550.

Analisando-se a tabela 4.2, pode-se observar que, conforme a ordem da matriz de entrada do método de Gram-Schmidt, tem-se melhor desempenho na CPU ou na GPU. Isso pode ser melhor visualizado na figura 4.1, que mostra o *speedup* em função da ordem da matriz. Quando a ordem é inferior a 200, a melhor escolha é processamento sequencial. Para ordens de

200 a 1500, o dispositivo com melhor desempenho é a CPU com OpenMP. Acima deste valor, a GPU passa ser a melhor escolha. Estes resultados vem comprovar o estudo realizado por (STOJANOVIC SASA BOJIC DRAGAN, 2012) sobre arquiteturas paralelas, em que demostrado que a GPU fornece aceleração eficaz, quando há grande quantidade de dados, processamento altamente paralelo, comunicação e sincronização podendo ser evitados.

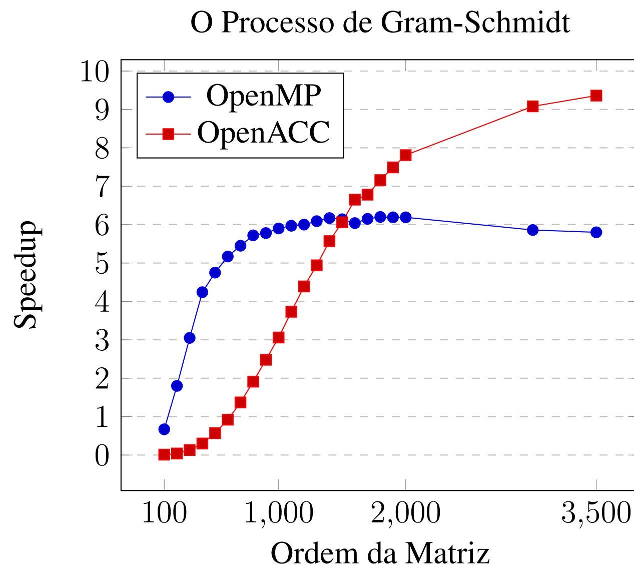


Figura 4.1: O Processo de Gram-Schmidt, Execução Sequencial, Paralelo com OpenMP com 8 threads, Paralelo na GPU com OpenACC com processador AMD 2.4 GHz e GPU Nvidia GTX 550.

Esses resultados reforçam a percepção de que nem sempre se obtém melhor desempenho em GPU. Além disso, para uma mesma aplicação e plataforma de execução, o desempenho pode ser diferente dependendo do tamanho dos dados de entrada. Uma solução para determinar o melhor mapeamento, em CPU ou em GPU, pode ser a execução de diferentes experimentos, como os que foram recém apresentados. Esse trabalho moroso, no entanto, pode ser abreviado por meio de soluções que visam automatizar etapas do processo de busca pelo melhor desempenho.

4.3 Automatização na Busca de Desempenho

A concepção de ferramentas com a finalidade auxiliar o desenvolvimento de programas com alto desempenho é um consenso entre pesquisadores, pois constitui uma tarefa sistemática e portanto passível de ter etapas automatizadas. Tendo em vista que a GPU ainda representa um desafio para pesquisadores, diversas abordagens têm sido empregadas, na tentativa de reduzir o

esforço para obtenção de melhores resultados.

Uma possibilidade de automatização na busca por desempenho é a geração automática de código otimizado. Em (GREWE; LOKHMOTOV, 2011), é proposto um sistema que permite a um compilador gerar um código otimizado para operações com matrizes esparsas, para execução em GPU. O sistema usa uma representação própria de matrizes esparsas e permite gerar código para GPU usando OpenCL ou CUDA. Este trabalho tira vantagem de uma particularidade das matrizes esparsas (grande quantidade de zeros) para reduzir a quantidade dados que trafegam entre CPU e GPU, proporcionando melhor desempenho com redução do tempo de execução. A matriz esparsa é compactada utilizando ferramentas de bibliotecas e então enviada da CPU para GPU e vice-versa, quando as operações aritméticas são concluídas.

O sistema proposto por (GREWE; LOKHMOTOV, 2011) foi aplicado a diferentes formatos e instâncias de matrizes esparsas, apresentando desempenho similar ou melhor do que quando as otimizações foram feitas manualmente. O trabalho aplica-se somente a matrizes esparsas e não tem a mesma eficácia quando aplicado às matrizes em geral. Uma contribuição importante deste trabalho está no esforço para reduzir a quantidade de dados transmitidos, que é um fator primordial para reduzir o tempo total de execução. De fato, segundo (GREGG; HAZELWOOD, 2011), qualquer estudo realizado visando otimizar o desempenho de uma arquitetura híbrida composta por CPU e GPU deve levar em conta a localização e a quantidade de dados envolvidas no processo. Dependendo desses fatores e da largura de banda entre *host* e GPU, pode tornar-se mais vantajosa a execução somente na CPU, mesmo que exista aceleração se a GPU for considerada isoladamente.

Em (SAMADI et al., 2012) é proposto o Adaptic, um sistema de compilação sensível a entrada de dados. Esta ferramenta faz a geração de código-fonte em CUDA para execução em paralelo na GPU e aborda o problema que ocorre quando acontecem mudanças no montante de dados envolvidos no processo, onde um código já otimizado pode apresentar desempenho pobre. Com base no desempenho de um *benchmark* embutido no Adaptic, é tomada a decisão de executar o código na CPU ou na GPU. Além disso, os autores mostram que o código gerado pelo Adaptic para CUDA, sensível aos dados, é portável para qualquer GPU ou seja, "otimize uma vez e execute em qualquer lugar".

No decorrer do artigo de (SAMADI et al., 2012), é feita uma extensa revisão bibliográfica apontando para trabalhos que fazem a geração de código fonte em CUDA e OpenCL. Como diferencial de Adaptic, está a geração de código sensível a entrada de dados. Apesar

de ser um estudo recente, não é feita referência ao OpenACC, que faz geração de código de forma transparente e apresenta parâmetros de configuração que o tornam sensível à variação da quantidade de dados. Mesmo assim, o trabalho vem confirmar a relação entre a quantidade de dados envolvida no processamento e a variação do *speedup*.

Numa linha de otimizações automáticas para OpenACC, tem-se o trabalho de (MAGNI; GREWE; JOHNSON, 2013), que aborda o problema da escolha do melhor mapeamento de laços sequenciais sobre *threads* paralelas, dado um programa e suas entradas. Os autores mostram que, encontrando-se a melhor configuração para os parâmetros `num_gangs` e `vector_length` da diretiva `kernels` do OpenACC, consegue-se um ganho de até 4,8x sobre o desempenho da configuração padrão do compilador, para os *benchmarks* utilizados no trabalho. Este estudo concentra-se em otimizar a busca dos parâmetros em termos percentuais, sem apresentar tempos reais de execução em GPUs existentes.

No trabalho apresentado em (CHE; SKADRON, 2014), os autores chamam a atenção para a rápida evolução das GPUs, que levam ao mercado novos dispositivos em curtos intervalos de tempo, o que acrescenta um grau de dificuldade a mais na busca por desempenho. Diante disso, o artigo defende a necessidade da realização de mapeamento específico e otimização para conseguir alto desempenho, bem como a necessidade de entender as métricas de primeira ordem que mais influenciam o desempenho e a escalabilidade na GPU. Além disso, destaca a importância de ferramentas e metodologias para prever o desempenho das aplicações GPU como auxílio ao usuário. Neste sentido, os autores propõem a realização de previsões de desempenho de aplicações em GPU através da correlação entre cargas de trabalho de diferentes *benchmarks*. A ideia dos autores é que, utilizando aplicações com cargas de trabalho padronizadas, torna-se possível realizar inferências sobre desempenho em aplicações GPU. O trabalho consistiu em identificar características de um conjunto de aplicações em GPU e então utilizá-las para prever o desempenho de aplicações arbitrárias, determinando a similaridade destas com os *benchmarks*. Com isso, os autores alcançam o objetivo de fazer previsões do *speedup* como forma de auxiliar o usuário na escolha do hardware mais adequado para suas aplicações, levando-se em conta as diversas opções de GPU.

O artigo de (CHE; SKADRON, 2014) foi publicado após o início do presente trabalho de mestrado, portanto não serviu de referência inicialmente. No entanto, ambos os trabalhos têm alguns fatores em comum, que corroboraram a abordagem explorada nesta dissertação. Esses fatores são as métricas de desempenho como forma de prever o comportamento da GPU

e amparar decisões, a utilização de *benchmarks* neste processo preditivo e a análise baseada em dados de execução, e não puramente baseadas em especificações de hardware. Por outro lado, (CHE; SKADRON, 2014) não consideram diretamente a quantidade de dados de entrada como fator causador de variações no *speedup*. Os experimentos são realizados utilizando toda a capacidade da GPU e, em nenhum momento, é cogitada a hipótese da CPU possuir melhor desempenho para determinada aplicação sob determinadas condições. O artigo parte do princípio de que a aplicação sempre tem melhor desempenho na GPU e, através da análise amparada por correlação com *benchmarks*, auxilia o usuário a escolher a melhor GPU para seu problema.

5 DECISÃO AUTOMATIZADA

Conforme discutido no capítulo anterior, a obtenção de alto desempenho numa arquitetura paralela híbrida pode se beneficiar de alguma ação automatizada. Neste trabalho de mestrado, focou-se na automatização da seguinte decisão num dado sistema híbrido: *em qual unidade do sistema deve ser mapeada uma determinada tarefa, para que se obtenha o melhor desempenho no hardware disponível*? Por unidade, neste trabalho, entende-se que possa ser uma CPU ou uma GPU. Entende-se por *tarefa* uma região de código que deseja-se executar em paralelo e esteja demarcada com diretivas `kernels` ou `parallel` do OpenACC.

O OpenACC, conforme apresentado no capítulo 3, disponibiliza cláusulas que interferem no mapeamento de computações de maior ou menor granularidade (*gang*, *worker* ou *vector*) sobre os elementos do dispositivo acelerador. Além disso, um compilador pode configurar automaticamente os parâmetros para essas cláusulas, buscando o melhor desempenho em uma dada GPU. O mapeamento a que se refere o presente trabalho está em outro nível, considerando que o sistema híbrido dispõe não somente de acelerador(es), mas também de pelo menos uma CPU (*host*), potencialmente *multicore*, capaz de executar a região paralela. Entende-se que este nível de mapeamento seja útil pois, conforme vários autores citados no capítulo anterior, há casos em que a GPU supera a CPU e outros em que o ocorre o contrário, e identificar esses casos exige um esforço considerável. Além disso, esse nível de mapeamento torna-se importante num sistema híbrido, em que o paralelismo pode estar presente em diferentes tipos de unidades.

Um passo em direção à automatização deste tipo de mapeamento é conhecer o desempenho de uma tarefa paralela em CPU e em GPU, ou seja, deve-se possuir dados sobre desempenho, mesmo que parciais, para apoiar a tomada da decisão de mapeamento. Outro passo é a tomada de decisão em si, incluindo ações de, possivelmente, alteração no código-fonte e/ou configuração do ambiente de execução. Para atingir o objetivo de abreviar o processo como um todo, almeja-se que esses passos sejam feitos com um mínimo de interferência do desenvolvedor.

5.1 Critérios de Apoio à Decisão

A abordagem adotada neste trabalho tem por hipótese que o desempenho em CPU e em GPU possa ser estimado para uma determinada tarefa, em um dado sistema híbrido real. Essa estimativa pode ser aproximada pois, no pior dos casos, será equivalente a uma estima-

tiva errônea realizada manualmente, que será percebida e poderá ser corrigida para execuções subsequentes. Isso não é válido para aplicações altamente irregulares, com alterações de comportamento em tempo de execução, mas é uma suposição razoável para muitas aplicações de computação científica.

Assim, este trabalho propõe que a estimativa de desempenho em CPU e GPU seja feita baseando-se conjuntamente nos seguintes critérios:

1. **tamanho dos dados de entrada**: assim como nas automatizações propostas por (SAMADI et al., 2012), considera-se que o tamanho dos dados de entrada seja um fator determinante para o desempenho de uma dada tarefa. De fato, trabalhos já citados mostram que uma mesma tarefa pode ou não ter desempenho em GPU, variando-se o tamanho dos dados de entrada, portanto considera-se importante levar em conta este aspecto;

2. **complexidade no tempo e no espaço**: sabe-se que é possível analisar a eficiência de um dado algoritmo por meio de sua complexidade no tempo e no espaço (LEVITIN, 2002). A complexidade no tempo representa o número de vezes que determinada operação de grande relevância é realizada. Comumente utiliza-se a notação "O" (*big oh*) para expressar essa característica do algoritmo. Já a complexidade no espaço representa a quantidade de memória necessária para armazenar os dados utilizados pelo algoritmo, além do espaço necessário para suas entradas e saídas. As medidas de complexidade são aproximadas, servindo principalmente como critério de comparação e classificação de algoritmos;

3. **desempenho do hardware alvo em *benchmarks***: sabe-se que um *benchmark* é um programa que permite avaliar o desempenho de um hardware real em termos quantitativos, contrastando com abordagens teóricas e analíticas que envolvem modelagem e simulação. Os resultados obtidos com *benchmarks* constituem observações reais do comportamento do hardware submetido a determinadas cargas e tipos de trabalhos. Essas observações devem ser reprodutíveis e podem servir como referência para predições de desempenho (HOSTE et al., 2006).

5.2 Base de Apoio à Decisão

Para formar uma base de apoio à decisão, propõe-se que seja construída e mantida uma tabela em que cada linha é um *benchmark* em OpenACC, possivelmente pertencente a uma *suite* de *benchmarks* como o EPCC. Para cada *benchmark*, mantém-se na tabela: sua complexidade

no tempo, em notação "big oh", sua complexidade no espaço (em bytes), o tamanho dos dados de entrada (que deve variar em uma faixa de valores), o tempo sequencial (CPU), o tempo em paralelo (em GPU com OpenACC) e o *speedup* calculado a partir desses tempos. Os tempos em questão devem ser obtidos para um ou mais sistemas híbridos em que se deseje executar aplicações com apoio à tomada de decisão de mapeamento.

O principal objetivo desta base é servir de referência para tomadas de decisão futuras. Sua criação, que requer várias execuções de alguns *benchmarks*, ocorre uma única vez para um dado sistema híbrido e seus dados são, potencialmente, aproveitados em diferentes aplicações e execuções.

A fim de ilustrar a composição da base de apoio à decisão, foram realizados experimentos utilizando funções do Nível 1 do *benchmark* EPCC, executadas em um sistema híbrido composto por uma CPU Intel Xeon E5620 2.4GHz e uma GPU NVIDIA Tesla M2050. A complexidade no tempo foi obtida pela análise do código-fonte e a complexidade no espaço foi extraída medindo-se a memória alocada pelos programas. Para as funções consideradas, variou-se a quantidade de dados de entrada (*datasize*) de 1,00E+3 a 1,00E+7. Calculado o *speedup* para cada caso, registrou-se o tamanho mínimo do *datasize* para obter *speedup* maior que 1. A base assim formada é apresentada na tabela 5.1. Nesta tabela, os tempos de execução são a média aritmética para uma amostra de cinco resultados da execução da função, com o seu respectivo desvio padrão.

Funções Nível- 1	**Complexidade no tempo**	**Complexidade no espaço (bytes)**	**DataSize (n)**	**Tempo Sequencial**	**Desvio Padrão**	**Tempo OpenACC**	**Desvio Padrão**	**Speedup**
COV	$2\,O(n^2) + O(n^3)$	1.182.816	593.000	0,009480	0,002500	0,005082	0,000140	1,86
MVT	$2O(n^2)$	2.121.840	2.100.000	0,00106	0,000006	0,001015	0,000003	1,04
ATAX	$2O(n^2)$	2.221.840	2.209.092	0,001443	0,000030	0,001085	0,000090	1,33
BICG	$2O(n^2)$	2.230.256	2.209092	0,001432	0,000026	0,001076	0,000004	1,33
COR	$3O(n^2)O(n^3)$	1.248.256	624.000	0,008703	0,000016	0,005857	0,000068	1,49
3MM	$3O(n^3)$	40.064	35.000	0,000120	0,000001	0,000114	0,000001	1,06
SYR2K	$O(n^2) + O(n^3)$	194.752	197.000	0,000938	0,000017	0,000897	0,000002	1,05
SYRK	$O(n^2) + O(n^3)$	240.048	240.000	0,001443	0,000065	0,001394	0,000055	1,03
GEMM	$O(n^3)$	32.832	26.000	0,000088	0,000000	0,000073	0,000001	1,20
2MM	$2O(n^3)$	602.208	510.000	0,003952	0,000002	0,003537	0,000007	1,12

Tabela 5.1: Exemplo de tabela de apoio à decisão usando funções do benchmark EPCC Nível 1.

Conforme apresentado no capítulo 4, seção 4.2, o Nível 1 do EPCC é composto por 13 *benchmarks*, porém somente 10 estão incluídos na tabela 5.1, pois os demais não apresentaram desempenho satisfatório em GPU até o limite da memória disponível. Na tabela 5.2, são apresentados os dados dessas 3 funções restantes que não obtiveram *speedup* maior que 1, ou

seja, cujo desempenho em GPU foi inferior ao obtido na CPU, em todos os casos considerados.

Funções Nível 1	**Complexidade do Algoritmo**	**Speedup**
GESUMMV	$O(n^2)$	0,75
2DCONV	$O(n^2)$	0,70
3DCONV	$O(n^3)$	0,49

Tabela 5.2: Funções do EPCC Nível 1 que não chegaram a ter o speedup maior que 1.

Uma vez formada a base de apoio à decisão, pode-se usá-la em estimativas de desempenho para aplicações que tenham comportamento semelhante (HOSTE et al., 2006). Quanto maior e mais representativa for essa base, melhores serão as estimativas.

5.3 Estimativa de Desempenho e Decisão de Mapeamento

Pela análise da tabela 5.1, pode-se inferir que o *speedup* tenha relação com a quantidade de operações primitivas que o algoritmo realiza. Uma operação primitiva causa um acesso à memória. A notação “O” faz uma estimativa do número de operações primitivas em função da entrada de dados (n), sendo que o valor de n pode ser estimado pela quantidade de memória alocada. Assim, propõe-se estimar o *speedup* pela notação ‘O”, usando como parâmetro a memória alocada.

Para decidir se uma tarefa deve ser mapeada em CPU ou GPU, o valor 1 para o *speedup* é importante, pois representa um limiar onde a execução em GPU começa a ser favorável. Analisando-se a tabela 5.1, observa-se que o *speedup* sofre influência da complexidade do algoritmo e também da quantidade de dados envolvida. No caso específico dos *benchmarks* EPCC analisados, nota-se que para obter *speedup* igual a 1, a complexidade no tempo deve ser no mínimo 2O(n^2). Se a complexidade no tempo for 2O(n^2), a complexidade no espaço para esta arquitetura de hardware deve ser no mínimo 2E+06 bytes. Se o programa alocar no máximo 3E+04, a complexidade no tempo deve ser no mínimo O(n^3).

De posse desse tipo de observação, pode-se usar os limiares identificados para tomar a decisão de mapeamento de uma dada tarefa, desde que a mesma possua complexidades no tempo e no espaço semelhantes, considerando-se também a entrada de dados. Para a tomada de decisão, quando o compilador utilizado for o PGI Accelerator com suporte a OpenACC, é possível definir o mapeamento fixando a variável de ambiente `ACC_DEVICE`. Quando tem-

se a definição `ACC_DEVICE=host`, a execução é realizada em CPU. Caso a definição para esta variável seja `ACC_DEVICE=nvidia`, a execução acontece na GPU. Isto é possível porque, quando o PGI Accelerator compila o programa com o parâmetro `-ta=nvidia,host`, são geradas duas cópias e a escolha de qual cópia será executada é definida pela variável de ambiente.

A aplicação que deseja se beneficiar da decisão automatizada deve conhecer sua quantidade de memória alocada para consultar a base de apoio à decisão. Depois de consultada a base e feita a estimativa, a variável `ACC_DEVICE` deve ser configurada de acordo com a decisão tomada. Esse processo deve ocorrer a cada execução e exige pequenas modificações na aplicação, mas é possível implementar isso com o mínimo de interferência do desenvolvedor, usando um código reaproveitável numa biblioteca. Na figura 5.1, é apresentado um esquema de tal código, em linguagem C, mostrando como calcular a memória alocada pelo sistema e setar a variável ACC_DEVICE para execução sequencial ou na GPU.

```
1  #include < stdio.h >
2  #include < stdlib.h >
3  #include < string.h >
4  #include < malloc.h > //biblioteca com funcao para leitura da memoria
       alocada
5  int conta_mem = 1; // permite que a memoria alocada seja calculada
6  unsigned long int mi1, mi2, mi3, mi4;
7  struct mallinfo mi; // criacao da estrutura que armazena a memoria alocada
8  int main(int argc, char * argv[]) {
9      mi = mallinfo(); // determinacao da memoria alocada antes da alocacao
10     // por malloc  e armazenamento  na variavel mi.
11     mi1 = mi.uordblks;
12     mi3 = mi.hblkhd;
13     //.................
14     //.................
15     funcao_paralelizada_com_OpenACC();
16
17 }
18 void funcao_paralelizada_com_OpenACC() {
19     // Alocacao de memoria por malloc  malloc
20     //  .................
21     //  .................
22     mi = mallinfo(); // determinacao da memoria alocada depois da alocacao
23     // por malloc  e armazenamento  na variavel mi.
24     if (conta_mem) { // Garantia que a memoria alcada antes da execucao do
25         // programa seja calculada uma unica vez.
26         mi2 = mi.uordblks;
27         mi4 = mi.hblkhd;
28     }
29     conta_mem = 0;
30     long int memoria_calculada = mi4 - mi3 + mi2 - mi1; //memoria alocada
           // pela funcao
31     printf("Memoria alocada: %d \n ", mi4 - mi3 + mi2 - mi1);
32     int tam;
33     long int memoria_calculada = mi4 - mi3 + mi2 - mi1; //memoria alocada
           //pela funcao
34     char str_g[100];
35     FILE * fp;
36     fp = fopen("memoria_benchmark.txt", "r");
37     rewind(fp);
38     fscanf(fp, "%s", str_g);
39     int memoria_benchmark = atoi(str_g);
40     printf("memoria benchmark: %d", memoria_benchmark);
41     if (memoria_calculada > memoria_benchmark) {
42         if (putenv("ACC_DEVICE=nvidia")) {
43             printf("Erro ao adicionar a variavel de ambiente  ACC_DEVICE=
                   nvidia\n");
44         }
45     }
46     else {
47         if (putenv("ACC_DEVICE=host")) {
48             printf("Erro ao adicionar a variavel de ambiente ACC_DEVICE=
                   host\n");
49         }
50     }
51
52 printf("ACC_DEVICE: %s \n ", getenv("ACC_DEVICE"));
53 }
```

Figura 5.1: Esquema de código para tomada de decisão entre processamento sequencial e em paralelo com OpenACC na GPU, representado em linguagem C.

5.4 Ferramenta de Decisão

Nas seções anteriores, apresentou-se uma proposta de critérios e método para responder à seguinte questão para uma determinada arquitetura paralela híbrida: *em qual unidade do sistema deve ser mapeada uma determinada tarefa, para que se obtenha o melhor desempenho no hardware disponível*? Visando atingir o objetivo de abreviar o processo e exigir um mínimo de interferência do desenvolvedor, desenvolveu-se uma ferramenta que automatiza partes desse processo. Dado um sistema híbrido real (CPU+GPU), a proposta da ferramenta é manter uma base de apoio à decisão (que deve ser inicializada e pode ser atualizada) e fornecer meios para que programas OpenACC sejam automaticamente ajustados para usar GPU ou CPU, dependendo do desempenho estimado.

Na figura 5.2, apresenta-se um esquema de funcionamento da ferramenta desenvolvida. Nesse esquema, há um banco de dados que é consultado para tomada de decisões. Esse banco de dados é alimentado com resultados de análise e execução de *benchmarks* com OpenACC em CPU e GPU, no sistema híbrido em questão. Os dados no banco têm a forma da tabela 5.1 e devem ser preenchidos pelo menos uma vez (inicialização) para o dado sistema híbrido. A inicialização (e posterior atualização) requer a execução dos *benchmarks* e análise de complexidades conforme descrito na seção 5.2. Para os *benchmarks* EPCC Nível 1, as complexidades no tempo e espaço já se encontram na tabela 5.1 e podem ser reaproveitadas em outros sistemas CPU+GPU, bastando-se levantar os novos tempos e calcular o *speedup*.

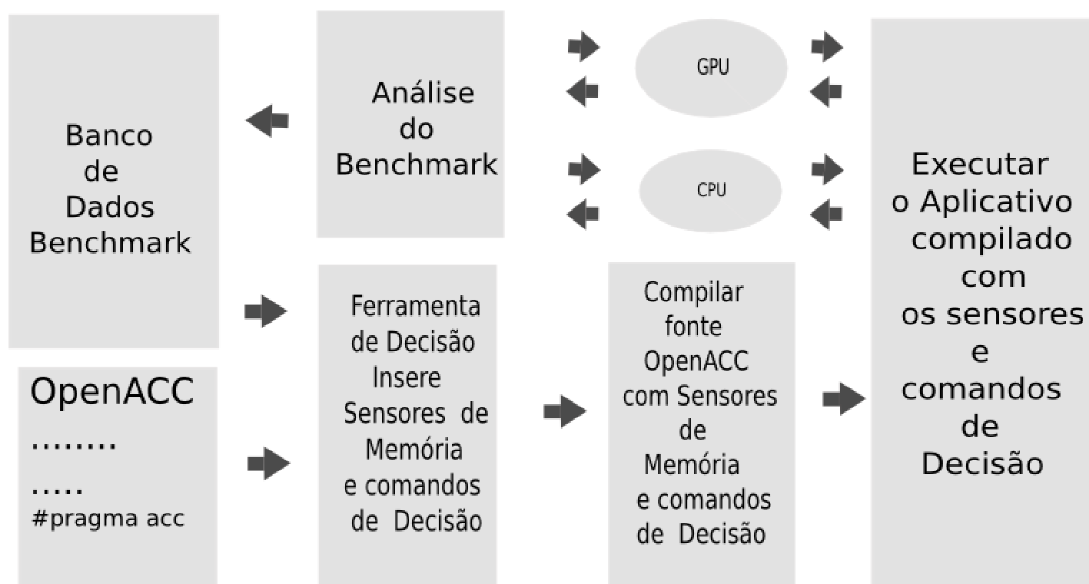


Figura 5.2: Esquema de funcionamento da ferramenta de decisão.

A eficácia da tomada de decisão é condicionada à existência de pelo menos um *benchmark* que represente a complexidade no tempo para a classe do programa que se deseja executar. Portanto, a base de dados deve ser atualizada conforme a natureza da complexidade computacional dos programas que serão utilizados. Desta forma, a margem de erro na escolha do dispositivo onde será feita a execução do problema será menor. Tanto a inicialização como a atualização da base devem ser feitas manualmente, não necessariamente por um desenvolvedor, podendo ser uma tarefa delegada a um administrador do sistema. Dadas as características da base, seu armazenamento pode ser feito em arquivo, não sendo necessário utilizar um sistema gerenciador de banco de dados.

Dado um programa paralelizado com OpenACC, que se deseje executar no sistema híbrido, o usuário desenvolvedor precisa analisar a região paralela (ou regiões) e determinar sua complexidade no tempo, para poder consultar a base de dados em busca de uma função com complexidade similar. Esta informação será usada pela ferramenta junto com outra informação sobre a memória alocada, que é obtida automaticamente. Para isso, a ferramenta faz a leitura do programa paralelizado com OpenACC e insere sensores que fazem a leitura da memória do programa em execução, antes da alocação da memória e após a alocação da memória. A diferença entres os valores antes e depois é a memória alocada pelo programa.

A ferramenta também insere comandos que fazem a leitura dos dados do *benchmark* na base de apoio à decisão, juntamente com comandos que, quando da execução do programa, decidirão sobre o mapeamento da tarefa em CPU ou GPU. A decisão, conforme já explicado, usará a quantidade de memória alocada como parâmetro: se a memória alocada for maior do que a do *benchmark* de referência, a tarefa será mapeada em GPU; em caso contrário, o mapeamento será na CPU. Para se beneficiar da decisão automatizada, o programa deve ser recompilado. A versão presente da ferramenta faz a decisão entre CPU e GPU usando variáveis de ambiente do OpenACC, que afetam o comportamento do programa durante sua execução. Versões futuras poderão incluir o OpenMP, tendo em vista que diretivas semelhantes às do OpenACC estarão presentes no OpenMP na versão 4.0, para uso de GPUs.

6 AVALIAÇÃO DA FERRAMENTA DE TOMADA DE DECISÃO

A avaliação da ferramenta foi realizada com o objetivo de testar sua funcionalidade, limitações e qualidade das estimativas frente a programas de computação científica. Para alcançar os objetivos, foram escolhidos como amostra programas de domínio público destinados a computação científica, que demandam recursos computacionais que justifiquem a utilização do processamento em paralelo e que tenham sido paralelizados com OpenACC. Uma vez realizada a seleção das amostras, o código fonte de cada programa selecionado é passado para ferramenta conforme descrito na seção anterior.

Foram escolhidos três programas, pertencentes ao *benchmark* Polybench. São eles: `gramschmidt` (decomposição pelo método de Gram-Schmidt), `lu` (decomposição LU) e `durbin` (solução de sistema com matriz de Toeplitz). Cada um deles possui complexidade computacional diferente. Os experimentos foram realizados variando-se a entrada de dados, representada pela ordem de uma matriz quadrada. Para cada experimento, obteve-se a memória alocada em bytes, o tempo de execução sequencial (Host) e o tempo de execução na GPU (Device), com seus respectivos desvios padrões, obtidos pela repetição do experimento cinco vezes com o programa original. Para analisar a possível sobrecarga no tempo causada pela execução do código fonte alterado, o procedimento acima foi realizado com o programa original e com o programa alterado (neste segundo caso, os tempos de execução são identificados como “Ferramenta Host” e “Ferramenta Device”).

Todos os experimentos foram executados em um sistema híbrido composto por uma CPU Intel Xeon E5620 2.4GHz e uma GPU NVIDIA Tesla M2050. O compilador utilizado foi o PGI Accelerator para linguagem C, com suporte a OpenACC. Em todos os experimentos, a base de apoio à decisão foi composta pelos *benchmarks* EPCC, com os dados já apresentados anteriormente (tabela 5.1).

A eficácia da decisão automatizada pode ser verificada comparando-se os tempos de execução entre Host, Device e da Ferramenta. Para que o tempo de execução da amostra executada pela ferramenta seja satisfatório, ele deve ser aproximadamente igual ao melhor tempo entre Host e Device. Ao deparar-se com diferenças de tempo pequenas, neste caso deve-se avaliar o impacto da sobrecarga de tempo causado pela utilização da ferramenta. No entanto, se o tempo da ferramenta for aproximadamente igual ao pior tempo entre Host e Device, isto significa que o parâmetro de decisão não proporciona a escolha pelo melhor tempo de execução. Neste caso,

o parâmetro com o valor da memória alocada pode ser alterado para mais ou para menos de forma que na próxima execução o dispositivo seja escolhido corretamente.

6.1 Experimentos com Decomposição de Gram-Schmidt

Nestes experimentos, utilizou-se a ferramenta para tomar a decisão de executar na CPU ou na GPU de forma automatizada. O critério para tomada decisão foi a quantidade de memória alocada para obter *speedup* maior ou igual 1 pela função COR (Correlation) do benchmark EPPC (1.248.256 bytes), tabela 5.1. A função Correlation foi escolhida pois apresenta complexidade computacional no tempo semelhante ao do processo de Gram-Schmidt. A utilização do valor da memória da função Correlation serviu como um parâmetro inicial. Após analisado o resultado deste valor, este poderia ter sido alterado visando melhorar o desempenho. A alteração poderia ser facilmente executada modificado o valor no banco de dados do *benchmark*, melhorando o desempenho na próxima utilização do programa.

Os resultados do experimento estão na tabela 6.1, onde observa-se que a utilização do OpenACC somente foi vantajosa para matrizes de ordem igual ou superior a 300, porém para matrizes de ordem inferior, foi mais eficiente o uso da CPU no processamento, pois foram alcançados tempos menores com uso deste dispositivo.

Ortogonalização de Gram-Schmidt (segundos) - Memória alocada.(bytes)											
Matriz	Memória	Host	Desvio Padrão	Device	Desvio Padrão	Ferramenta Host	Desvio. Padrão	Ferramenta. Device	Desvio Padrão	Ferramenta.	Decisão
100	80.032	0,0026	0,000474	0,02	0,03	0,002607	0,000609	-	-	0,002607	host
200	327.680	0,0174	0,002519	0,0245	0,037	0,0175	0,00081	-	-	0,0163	host
300	720.896	0,054	0,000272	0,032	0,040	**0,055317**	0,001974	-	-	0,0556	host
400	1.286.144	0,127	0,000596	0,038	0,038	-	-	**0,037**	0,036	0,039	device
500	2.007.040	0,2489	0,001863	0,047	0,039	-	-	0,0466	0,038	0,048	device
600	2.883.584	0,4235	0,0064	0,060	0,037	-	-	0,0602	0,0362	0,0602	device
700	3.923.968	0,6702	0,01385	0,074	0,038	-	-	0,0740	0,0379	0,0740	device
800	5.128.192	0,97821	0,04746	0,085	0,0372	-	-	0,0860	0,0387	0,0860	device
900	6.488.064	1,3634	0,06541	0,106	0,038	-	-	0,1072	0,0388	0,1072	device
1000	8.003.584	1,8475	0,0610	0,124	0,0377	-	-	0,12479	0,03816	0,12479	device
1500	18.006.016	6,0329	0,11734	0,290	0,039	-	-	0,290325	0,03895	0,290325	device
2000	32.006.144	14,2620	0,0535	0,591	0,039	-	-	0,592	0,0383	0,5903	device

Tabela 6.1: Resultados do Experimento com o programa paralelizado com OpenACC que implementa o Processo de Gram-Schmidt

A decisão automatizada realizada pela ferramenta determinou que a execução fosse na GPU quando a ordem da matriz fosse maior ou igual 400. A diferença entre a ordem da matriz observada 300 para ordem calculada 400 é devida à diferença entre a quantidade de operações aritméticas estimadas da função de correlação e a função de Gram-Schmidt, estimadas pela notação “O”. De acordo com (GOODRICH; TAMASSIA, 2009) a notação “O” faz um análise

simplificada e uma estimativa da quantidade de operações aritméticas e portanto estas diferenças são passiveis de acontecer.

No experimento, foi verificado o impacto do uso da ferramenta no desempenho da aplicação, executando o programa original e o programa modificado pela ferramenta, que encontra-se no Apêndice A. Os tempos de execução são apresentados na tabela 6.1 e observa-se que a diferença entre os tempos de execução não foi significativa e estão dentro da margem de erro representada pelo desvio padrão. Sendo assim, o uso da ferramenta não provoca queda de desempenho que possa ser considerada significativa.

6.2 Experimentos com a Decomposição LU

A decomposição LU (em que LU vem do inglês *lower* e *upper*) é uma forma de fatoração de uma matriz como o produto de uma matriz triangular inferior (*lower*) e uma matriz triangular superior (*lower*) (ANTON; BUSBY, 2006).

Os experimentos foram realizados apenas com o código fonte alterado pela ferramenta. Para medir o desempenho, foi calculado o *speedup* das amostras e, para que isso fosse possível, o parâmetro de decisão foi configurado como zero, forçando a escolha pelo dispositivo *host* e os resultados foram identificados como "Ferramenta Host". Da mesma forma, foi determinado que a execução fosse somente no dispositivo *device* configurando o valor da memória alocada para um valor superior ao maior valor de memória alocada no experimento.

A decomposição LU possui complexidade O(n^3). Dessa forma, a função do *benchmark* EPCC que serviu como base foi GEMM, que possui a memória alocada de 32.832 bytes, conforme a tabela 5.1 (base de apoio à decisão).

Os resultados dos experimentos são apresentados na tabela 6.2, onde observa-se que o ponto de tomada decisão seria quando a ordem da matriz LU fosse da ordem de 400, quando o *speedup* tem como valor 1 e memória alocada 1.292.048 bytes. No entanto, o parâmetro de decisão determina que a execução ocorra no *device* quando a memória for superior a 32.832 bytes, havendo uma diferença significativa entre eles.

Este fato, juntamente com a análise do código fonte representado na figura 6.1, o qual evidencia a influência das dependências entre as variáveis no processo de decisão automatizada, causa uma diferença entre o parâmetro utilizado e aquele considerado como sendo ideal.

Matriz LU (segundos) - Memória alocada.(bytes)							
Matriz Ordem	Memória	Ferramenta Host	Desvio Padrão	Ferramenta Device	Desvio Padrão	Speedup	Decisão
100	80.048	0,000551	0,000011	0,018343	0,040000	0,03	device
200	323.584	0,003969	0,000044	0,018802	0,039000	0,21	device
300	720.896	0,013000	0,000017	0,020000	0,040000	0,64	device
400	1.282.048	0,030000	0,000132	0,022000	0,040000	1,39	device
500	2.000.944	0,060000	0,000088	0,024300	0,038650	2,46	device
600	2.883.584	0,103700	0,000063	0,028800	0,039000	3,59	device
700	3.923.968	0,164900	0,000117	0,034700	0,039400	4,75	device
800	5.124.096	0,244400	0,000706	0,042800	0,039000	5,70	device
900	6.483.968	0,349200	0,000184	0,054150	0,037900	6,45	device
1000	8.003.584	0,463500	0,217890	0,095000	0,039580	4,84	device
1500	18.001.920	1,593400	0,065900	0,356500	0,040000	4,46	device
2000	32.002.048	3,990000	0,067000	0,883700	0,040000	4,51	device
3000	72.003.584	13,799000	0,014200	3,159000	0,042000	4,36	device
4000	128.004.096	32,940000	0,074500	7,641000	0,048800	4,31	device

Tabela 6.2: Resultados do Experimento com o programa paralelizado com OpenACC que implementa a Decomposição LU

```
1           #pragma acc data copy(A)
2             {
3               #pragma acc parallel
4               {
5                 #pragma acc loop
6                 for (k = 0; k < _PB_N; k++)
7                   {
8                     #pragma acc loop
9                     for (j = k + 1; j < _PB_N; j++)
10                      A[k][j] = A[k][j] / A[k][k];
11                    for(i = k + 1; i < _PB_N; i++)
12                      #pragma acc loop
13                      for (j = k + 1; j < _PB_N; j++)
14                        A[i][j] = A[i][j] - A[i][k] * A[k][j];
15                  }
16              }
17            }
```

Figura 6.1: Fragmento de código paralelizado da Decomposição LU extraído do benchmark Polybench.

6.3 Experimentos com a Matriz de Toeplitz

Matriz de Toeplitz ou matriz de diagonais constantes, chamada assim em homenagem a Otto Toeplitz, é uma matriz em que cada diagonal descendente da esquerda para a direita tem valor constante.

A solução de sistema com matriz de Toeplitz possui complexidade O(n^2). Dessa forma, a função do *benchmark* EPCC que serviu como base foi ATAX, que possui a memória alocada

de 2.221.840 bytes, conforme a tabela 5.1.

Os resultados do experimento estão na tabela 6.3 e evidenciam uma grande diferença entre a memória utilizada como parâmetro de decisão e aquela na qual o *speedup* passa ser maior que 1. Tendo isso em vista, o valor do parâmetro de decisão deve ser alterado, para o seu valor real conforme a tabela 6.3. Quando isto ocorre, o valor de referência obtido na tabela 5.1 serve apenas como um parâmetro inicial.

Matriz de Toeplitz (segundos) - Memória alocada.(bytes)							
Matriz	Memória	Host	Desvio Padrão	Device	Desvio Padrão	Speedup	Decisão
300	1.451.584	0,0006986618	5,44875005849967E-006	0,082954	0,0051654069	0,0084222798	host
500	4.022.080	0,002606536	0,0004831992	0,08145776	0,0024246522	0,0319986211	device
700	7.870.528	0,005980012	2,78005974755938E-005	0,0839134	0,0028321808	0,0712640889	device
1000	16.039.360	0,01449374	4,81453839947299E-005	0,0858894	0,0016308677	0,1687488794	device
2000	64.068.288	0,07813078	0,0003268185	0,1020812	0,0004014196	0,7653787377	device
3000	144.103.360	0,2091566	0,0011995861	0,1491462	0,0014241172	1,402359564	device
4000	256.136.384	0,4126156	0,0006340196	0,2007422	0,0038076167	2,0554502242	device
5000	400.167.360	0,7038638	0,0004667946	0,2680216	0,0031605229	2,6261458032	device
6000	576.196.288	1,065334	0,0008013301	0,3507512	0,0011974138	3,037292531	device
7000	784.231.360	1,524834	0,0018259874	0,4720872	0,0033481232	3,2299837827	device
8000	1.024.264.384	2,097168	0,0041384019	0,5803428	0,0017377121	3,6136710923	device
9000	1.296.295.360	2,890634	0,00152569	0,7084416	0,0030116745	4,0802714013	device
10000	1.600.324.288	3,8507833333	0,0045307211	0,8478778	0,0021574009	4,5416725539	device
11000	1.936.359.360	5,103684	0,0064821123	1,04816	0,0011358257	4,869184094	device
11500	2.116.369.984	5,7988975	0,0082768326	1,1493925	0,0021051742	5,0451847389	device

Tabela 6.3: Resultados do Experimento com o programa paralelizado com OpenACC que implementa a Solução da Matriz de Toeplitz, utilizando o compilador pgcc, uma CPU Xeon com GPU Tesla 2050

6.4 Discussão

A eficácia da ferramenta decisão, que tem por base a análise de um *benchmark* é restringida aos algoritmos que possuem complexidade computacional no tempo similar ao do *benchmark*. As diferenças dos valores da memória alocada pelo *benchmark* e o programa paralelizado devem-se a parâmetros que não são facilmente mensuráveis, com por exemplo a dependência entre as variáveis. Portanto recomenda-se que a escolha do valor da memória utilizada como critério de decisão seja feita através de um processo iterativo, tomando como parâmetro inicial o valor obtido na análise do *benchmark*.

```
1          #pragma acc data copyin(y,sum,alpha,beta,r) copyout(out)
2           {
3             y[0][0] = r[0];
4             beta[0] = 1;
5             alpha[0] = r[0];
6             #pragma acc parallel
7             {
8               #pragma acc loop
9               for (k = 1; k < _PB_N; k++)
10              {
11                beta[k] = beta[k-1] - alpha[k-1] * alpha[k-1] * beta[k
                     -1];
12                sum[0][k] = r[k];
13                #pragma acc loop
14                for (i = 0; i <= k - 1; i++)
15                  sum[i+1][k] = sum[i][k] + r[k-i-1] * y[i][k-1];
16                alpha[k] = -sum[k][k] * beta[k];
17                #pragma acc loop
18                for (i = 0; i <= k-1; i++)
19                  y[i][k] = y[i][k-1] + alpha[k] * y[k-i-1][k-1];
20                y[k][k] = alpha[k];
21              }
22              #pragma acc loop
23              for (i = 0; i < _PB_N; i++)
24              out[i] = y[i][_PB_N-1];
25            }
26          }
```

Figura 6.2: Fragmento de código paralelizado da Solução da Matriz de Toeplitz.

7 CONCLUSÃO

A obtenção do aumento de desempenho através da execução em paralelo de um sistema é inerente à natureza do algoritmo e suas principais particularidades tais como, tamanho da entrada de dados, complexidade computacional no tempo e espaço, relações de dependência entre as variáveis, o dispositivo ou a combinação de dispositivos na qual o sistema executa, o compilador utilizado e a linguagem computacional escolhida, cada uma com impacto diferente.

O desenvolvimento de sistemas paralelos, sempre acrescenta algum custo adicional seja em recursos humanos ou físicos, sendo necessário fazer uma previsão do desempenho com a finalidade de verificar a viabilidade do empreendimento. Este trabalho inicialmente realizou uma análise detalhada de *benchmarks* que realizaram a análise de desempenho entre processamento sequencial e paralelo com dispositivos GPU. Os programas integrantes dos *benchmarks* utilizados nos experimentos apresentaram tempo de execução diferentes conforme suas características particulares. A partir deste fato investigou-se quais as principais fontes de variação de desempenho. Quando características como linguagem de programação, compilador e ausência dependências entre variáveis são iguais para os programas, a variação do tempo de execução sequencial e paralelo é na maioria dos casos devido ao tamanho da entrada de dados e a complexidade computacional no tempo e no espaço. Esta peculiaridade é determinante de tal maneira que por simples análise visual, onde observa-se *loops* aninhados com complexidade menor ou igual O(n^2) a possibilidade da paralelização ser eficiente é bastante reduzida. A partir do ponto que a complexidade no tempo é superior a O(n^2) o fator determinante do desempenho passa ser a quantidade e dados e a complexidade no espaço ou seja a quantidade de memória alocada. Estes resultados são comuns aos sistemas paralelizados de modo geral, e significam que quando o tempo gasto entre a comunicação e trasporte de dados, entre as threads ou dispositivos de hardware, é superior ao tempo de execução em um único dispositivo ou thread, a paralelização nem sempre é viável.

Com dispositivos para processamento tais como GPUs, além do exposto acima outro fator determinante e fundamental é a largura da banda de transporte de dados (*Bandwidth*). Embora a GPU utilizada seja sempre superior em capacidade de processamento da CPU, caso contrário esta não teria sentido de existir na arquitetura proposta, este fator torna o limites em que a execução na GPU é vantajosa ainda mais elevados, tornando viabilidade da execução na GPU mais restrita.

Em face da previsibilidade do desempenho dos sistemas paralelizados, desenvolveu-se uma metodologia para escolha da forma com que um sistema poderia ser executado, CPU ou na GPU, tendo por critério de decisão a memória alocada pelas variáveis paralelizadas. Tendo em vista o OpenACC ser uma tecnologia nova em ascensão, foi desenvolvido neste trabalho uma ferramenta que implementa a decisão automatizada de forma transparente para o usuário em sistemas paralelizados com OpenACC. Testes realizados como esta ferramenta, evidenciam que sua eficiência é condicionada a existência prévia de uma base de dados com programas, pertencentes a um *benchmark*, com complexidade no tempo similar ao do programa paralelizado. Outro fator vital para o sucesso da ferramenta é a obrigação do programa escolhido do *benchmark* possuir o mesmo grau de dependência entre as suas varáveis. Um experimento realizado com a decomposição de Cholesky com o compilador pgcc e o padrão OpenMP, conforme apresentado no apêndice B, mostram que quando um trecho de código paralelizado é dependente de outro, o compilador insere automaticamente uma cláusula de barreira para garantir a integridade da solução na execução do programa. Portanto, o programa escolhido no *benchmark* para servir como referência deve ter uma quantidade similar de cláusulas de barreiras quando paralelizado, demostrando ter o mesmo grau de dependência entre as variáveis.

No decorrer deste trabalho, fica evidente que não tem sentido eleger uma forma de execução sequencial ou paralela como mais efetiva, pois cada qual possui suas vantagens e desvantagens que dependem das particularidades inerente ao algoritmo utilizado para a solução do problema. O que este trabalho vem comprovar são as etapas fundamentais para obter um sistema que execute de forma mais eficaz possível, tais como a identificação da região com maior custo computacional, otimização matemática desta região para execução em paralelo, escolha do dispositivo de hardware CPU ou GPU, que possa ser obtido melhor desempenho levando em conta a quantidade de dados que o sistema irá manipular e a complexidade computacional no tempo e espaço. A principal contribuição de trabalho está no desenvolvimento de meios para auxiliar a escolha do dispositivo de hardware mais adequado, utilizado previsão de desempenho.

7.1 Trabalhos Futuros

O padrão OpenMP em sua versão 4.0 contempla algumas funcionalidades para utilização da GPU, de modo que seria interessante desenvolver e aplicar os conceitos desenvolvidos neste trabalho com OpenACC utilizando OpenMP pois é suportado pelo compilador gcc de código aberto. Até a presente data a clausula *reduction* não foi implementada no OpenMP 4.0.

Desenvolver a próxima versão da ferramenta de tomada de decisão utilizando pthreads, OpenMP na CPU e GPU.

Ampliar o estudo para contemplar os processadores que possuem GPU incorporada aos seus núcleos.

Realizar um estudo de forma que seja possível quantificar o grau de dependência entre as variáveis de um sistema paralelizado, estabelecendo métricas de desempenho possibilitando sua utilização para aumentar precisão nas previsões.

REFERÊNCIAS

ALI, A.; SYED, K. S. An Outlook of High Performance Computing Infrastructures for Scientific Computing. **Advances in Computers**, [S.l.], v.91, p.87–118, 2013.

ANTON, H.; BUSBY, R. **Algebra Linear Contemporânea**. [S.l.]: Bookman, 2006.

BAILEY, D. H. et al. The NAS Parallel Benchmarks&Mdash;Summary and Preliminary Results. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 1991., New York, NY, USA. **Proceedings...** ACM, 1991. p.158–165. (Supercomputing '91).

BRODTKORB, A. R. et al. State-of-the-art in Heterogeneous Computing. **Sci. Program.**, Amsterdam, The Netherlands, The Netherlands, v.18, n.1, p.1–33, Jan. 2010.

BRODTKORB, A. R.; HAGEN, T. R.; SÆTRA, M. L. Graphics Processing Unit (GPU) Programming Strategies and Trends in GPU Computing. **J. Parallel Distrib. Comput.**, Orlando, FL, USA, v.73, n.1, p.4–13, Jan. 2013.

CHE, S. et al. A Performance Study of General-purpose Applications on Graphics Processors Using CUDA. **J. Parallel Distrib. Comput.**, Orlando, FL, USA, v.68, n.10, p.1370–1380, Oct. 2008.

CHE, S.; SKADRON, K. BenchFriend: correlating the performance of gpu benchmarks. **Int. J. High Perform. Comput. Appl.**, Thousand Oaks, CA, USA, v.28, n.2, p.238–250, May 2014.

COOK, S. **CUDA Programming**: a developer's guide to parallel computing with gpus. 1st.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

DAGUM, L.; MENON, R. OpenMP: an industry standard api for shared-memory programming. **Computational Science & Engineering, IEEE**, [S.l.], v.5, n.1, p.46–55, 1998.

DELL. **Multi-Core Technology Brief.** [Online; accessed 20-Julho-2014].

DIAZ, J.; MUNOZ-CARO, C.; NINO, A. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. **IEEE Transactions on Parallel and Distributed Systems**, Los Alamitos, CA, USA, v.23, n.8, p.1369–1386, 2012.

FANFARILLO, A. et al. Coarrays in GNU Fortran. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION, 23., New York, NY, USA. **Proceedings...** ACM, 2014. p.513–514. (PACT '14).

FARBER, R. **The OpenACC Execution Model. Dr Dobb's. Disponível em**: http://www.drdobbs.com/parallel/the-openacc-execution-model/240006334#disqus_thread. 2012.

FLYNN, M. J. Some Computer Organizations and Their Effectiveness. **IEEE Trans. Comput.**, Washington, DC, USA, v.21, n.9, p.948–960, Sept. 1972.

FREDRICKSON, N. R.; AFSAHI, A.; QIAN, Y. Performance Characteristics of openMP Constructs, and Application Benchmarks on a Large Symmetric Multiprocessor. In: ANNUAL INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 17., New York, NY, USA. **Proceedings...** ACM, 2003. p.140–149. (ICS '03).

DAYDÉ, M. et al. (Ed.). **High Performance Computing for Computational Science - VECPAR 2006**: 7th international conference, rio de janeiro, brazil, june 10-13, 2006, revised selected and invited papers. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p.39–51.

GOODRICH, M. T.; TAMASSIA, R. **Algorithm Design**: foundations, analysis and internet examples. 2nd.ed. New York, NY, USA: John Wiley & Sons, Inc., 2009.

GREGG, C.; HAZELWOOD, K. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In: PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE (ISPASS), 2011 IEEE INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2011. p.134–144.

GREWE, D.; LOKHMOTOV, A. Automatically Generating and Tuning GPU Code for Sparse Matrix-vector Multiplication from a High-level Representation. In: FOURTH WORKSHOP ON GENERAL PURPOSE PROCESSING ON GRAPHICS PROCESSING UNITS, New York, NY, USA. **Proceedings...** ACM, 2011. p.12:1–12:8. (GPGPU-4).

HAGERNÄS, P.; ANDRÉN, A. **Data-parallel acceleration of PARSEC Black-Scholes benchmarks**. [S.l.]: School of Information and Communication Technology, Royal Institute of Technology, Sweden, 2013.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture, Fifth Edition**: a quantitative approach. 5th.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

HOSHINO, T. et al. CUDA vs OpenACC: performance case studies with kernel benchmarks and a memory-bound cfd application. In: CLUSTER, CLOUD AND GRID COMPUTING (CC-GRID), 2013 13TH IEEE/ACM INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2013. p.136–143.

HOSTE, K. et al. Performance Prediction Based on Inherent Program Similarity. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 15., New York, NY, USA. **Proceedings...** ACM, 2006. p.114–122. (PACT '06).

HWANG, K.; XU, Z. **Scalable Parallel Computing**: technology,architecture,programming. New York, NY, USA: McGraw-Hill, Inc., 1998.

INTA, R.; BOWMAN, D. J.; SCOTT, S. M. The "Chimera": an off-the-shelf cpu/gpgpu/fpga hybrid computing platform. **Int. J. Reconfig. Comput.**, New York, NY, United States, v.2012, p.2:2–2:2, Jan. 2012.

JOHNSON, N. **EPCC OpenACC benchmark suite. Disponível em**: http://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openacc-benchmark-suite. 2013.

JUCKELAND, G. et al. SPEC ACCEL: A standard application suite for measuring hardware accelerator performance. In: HIGH PERFORMANCE COMPUTING SYSTEMS. PERFORMANCE MODELING, BENCHMARKING, AND SIMULATION - 5TH INTERNATIONAL WORKSHOP, PMBS 2014, NEW ORLEANS, LA, USA, NOVEMBER 16, 2014. REVISED SELECTED PAPERS. **Anais...** [S.l.: s.n.], 2014. p.46–67.

KASIM, H. et al. Survey on Parallel Programming Model. In: IFIP INTERNATIONAL CONFERENCE ON NETWORK AND PARALLEL COMPUTING, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2008. p.266–275. (NPC '08).

KENNEDY, K.; KOELBEL, C.; ZIMA, H. The Rise and Fall of High Performance Fortran: an historical object lesson. In: THIRD ACM SIGPLAN CONFERENCE ON HISTORY OF

PROGRAMMING LANGUAGES, New York, NY, USA. **Proceedings...** ACM, 2007. p.7–1–7–22. (HOPL III).

KIRK, D. B.; HWU, W.-m. W. **Programming Massively Parallel Processors**: a hands-on approach. 2nd.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

LARKIN, J. Getting Started with OpenACC. Disponível em: http://devblogs.nvidia.com/parallelforall/getting-started-openacc/.

LEPPER, A. **Accelerator weather forecasting**. 2015. Dissertação (Mestrado em Ciência da Computação) — The University of Edinburgh.

LEVITIN, A. V. **Introduction to the Design and Analysis of Algorithms**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

MAGNI, A.; GREWE, D.; JOHNSON, N. Input-aware Auto-tuning for Directive-based GPU Programming. In: WORKSHOP ON GENERAL PURPOSE PROCESSOR USING GRAPHICS PROCESSING UNITS, 6., New York, NY, USA. **Proceedings...** ACM, 2013. p.66–75. (GPGPU-6).

NUMRICH, R. W.; REID, J. Co-array Fortran for Parallel Programming. **SIGPLAN Fortran Forum**, New York, NY, USA, v.17, n.2, p.1–31, Aug. 1998.

NVIDIA. **NVIDIA's Next Generation CUDA Compute Architecture**:fermi. [Online; accessed 20-Julho-2014].

NVIDIA. **NVIDIA CUDA Programming Guide version 2.3.** [Online; accessed 20-Julho-2014].

OLUKOTUN, K. et al. The Case for a Single-chip Multiprocessor. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.30, n.5, p.2–11, Sept. 1996.

OPENACC. **The OpenACC Application Program Interface http**://www.http://www.openacc.org/. [Online; accessed 20-Julho-2014].

OpenMP.org. **OpenMP Specifications**. 2015.

PEDRETTI, K. et al. Early Experiences with Node-level Power Capping on the Cray XC40 Platform. In: INTERNATIONAL WORKSHOP ON ENERGY EFFICIENT SUPERCOMPUTING, 3., New York, NY, USA. **Proceedings...** ACM, 2015. p.1:1–1:10. (E2SC '15).

PENNYCOOK, S. J. et al. An Investigation of the Performance Portability of OpenCL. **J. Parallel Distrib. Comput.**, Orlando, FL, USA, v.73, n.11, p.1439–1450, Nov. 2013.

PGI. PGI Accelerator Compilers with OpenACC Directives. Disponível em: https://www.pgroup.com/resources/accel.htm.

REGULY, I. Z.; KEITA, A.-K.; GILES, M. B. Benchmarking the IBM Power8 Processor. In: ANNUAL INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND SOFTWARE ENGINEERING, 25., Riverton, NJ, USA. **Proceedings...** IBM Corp., 2015. p.61–69. (CASCON '15).

RENNICH, S. C.; STOSIC, D.; DAVIS, T. A. Accelerating Sparse Cholesky Factorization on GPUs. In: FOURTH WORKSHOP ON IRREGULAR APPLICATIONS: ARCHITECTURES AND ALGORITHMS, Piscataway, NJ, USA. **Proceedings...** IEEE Press, 2014. p.9–16. (IA3 '14).

RYOO, S. et al. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 13., New York, NY, USA. **Proceedings...** ACM, 2008. p.73–82. (PPoPP '08).

SAMADI, M. et al. Adaptive Input-aware Compilation for Graphics Engines. **SIGPLAN Not.**, New York, NY, USA, v.47, n.6, p.13–22, June 2012.

SKILLICORN, D. B. A Taxonomy for Computer Architectures. **Computer**, Los Alamitos, CA, USA, v.21, n.11, p.46–57, Nov. 1988.

STOJANOVIC SASA BOJIC DRAGAN, B. M. V. M. M. V. M. An overview of Selected Hybrid and Reconfigurable Architectures. **2012 IEEE INTERNATIONAL CONFERENCE ON INDUSTRIAL TECHNOLOGY (ICIT)**, [S.l.], p.444–449, 2012.

STROHMAIER, E. et al. Top500 Supercomputer Sites. Disponível em: http://top500.org.

SU, C.-L. et al. Overview and comparison of OpenCL and CUDA technology for GPGPU. In: APCCAS. **Anais...** IEEE, 2012. p.448–451.

VOLKOV, V.; DEMMEL, J. **LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs**. [S.l.]: EECS Department, University of California, Berkeley, 2008. (UCB/EECS-2008-49).

WIENKE, S. et al. OpenACC: first experiences with real-world applications. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 18., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2012. p.859–870. (Euro-Par'12).

WOLF. **The PGI Accelerator Compilers with OpenACC https**://www.pgroup.com/lit/articles/insider/v4n1a1.htm. [Online; accessed 20-Julho-2014].

WU, Q. et al. MIC Acceleration of Short-range Molecular Dynamics Simulations. In: FIRST INTERNATIONAL WORKSHOP ON CODE OPTIMISATION FOR MULTI AND MANY CORES, New York, NY, USA. **Proceedings...** ACM, 2013. p.2:1–2:8. (COSMIC '13).

XU, R. et al. NAS Parallel Benchmarks for GPGPUs Using a Directive-Based Programming Model. In: LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING - 27TH INTERNATIONAL WORKSHOP, LCPC 2014, HILLSBORO, OR, USA, SEPTEMBER 15-17, 2014, REVISED SELECTED PAPERS. **Anais...** [S.l.: s.n.], 2014. p.67–81.

YANG, D. et al. Performance Comparison of Cholesky Decomposition on GPUs and FPGAs. In: SYMPOSIUM ON APPLICATION ACCELERATORS IN HIGH PERFORMANCE COMPUTING (SAAHPC), KNOXVILLE, TN. **Proceedings...** [S.l.: s.n.], 2010.

APÊNDICES

APÊNDICE A – Ferramenta para tomada de decisão

O código fonte da ferramenta está disponível em https:bitbucket.orgppgidiss_renato ou solicitando ao autor em renato.ferrari@ufsm.br

Ortogonalização de Gram-Schmidt - alterado pela ferramenta

```
1  // Codigo fonte modificado
2  #include < malloc.h >
3  #include < stdio.h >
4  #include < stdlib.h >
5  #include < string.h >
6  #include "minhalib.h"
7  FILE * resultado;
8  struct mallinfo mi;
9  int conta_mem = 1;
10 unsigned long int mi1, mi2, mi3, mi4;
11 // Classical Gram-Schmidt Rob Farber
12 #include < stdio.h > #include < stdlib.h >
13 //#include <accelmath.h>
14 #include < omp.h >
15 #include < math.h >
16
17 #ifndef ROWS
18 #define ROWS 400
19 #endif
20 #ifndef COLS
21 #define COLS 400
22 #endif
23
24 #ifdef _OPENACC
25
26 void gramSchmidt(float Q[][COLS], const int rows, const int cols
       ) {
27     float Qt[cols][rows];
28
29     FILE * pt;
30     int memoria_benchmark;
31     mi = mallinfo();
32     if (conta_mem) {
33         mi2 = mi.uordblks;
34         mi4 = mi.hblkhd;
35     }
36     conta_mem = 0;
37     long int memoria_calculada = mi4 - mi3 + mi2 - mi1;
38     printf("->Memoria alocada: %d \n ", mi4 - mi3 + mi2 - mi1);
39     if ((pt = fopen("memoria_benchmark.txt", "r")) == NULL) {
40         printf("Erro ao abrir arquivo!!!  - ");
41         exit(1);
42     };
43     fscanf(pt, "%d", & memoria_benchmark);
44
45     printf("memoria benchmark: %d", memoria_benchmark);
46     if (memoria_calculada > memoria_benchmark) {
47         if (putenv("ACC_DEVICE=nvidia")) {
48             printf("Erro ao adicionar a variavel de ambiente
```

```
                ACC_DEVICE=nvidia\n");
49          }
50      }
51      else {
52          if (putenv("ACC_DEVICE=host")) {
53              printf("Erro ao adicionar a variavel de ambiente
                    ACC_DEVICE=host\n");
54          }
55      }
56      printf("ACC_DEVICE: %s \n ", getenv("ACC_DEVICE"));
57
58      #pragma acc data create(Qt[cols][rows]) copy(Q[0: rows][0:
            cols]) {
59          //transpose Q in Qt
60          #pragma acc parallel loop
61          for (int i = 0; i < rows; i++)
62          for (int j = 0; j < cols; j++)
63          Qt[j][i] = Q[i][j];
64
65          for (int k = 0; k < cols; k++) {
66
67              #pragma acc parallel {
68                  double tmp = 0.;
69
70                  #pragma acc loop vector reduction(+: tmp)
71                  for (int i = 0; i < rows; i++) tmp += (Qt[k][i]
                        * Qt[k][i]);
72                  tmp = sqrt(tmp);
73
74                  #pragma acc loop vector
75                  for (int i = 0; i < rows; i++) Qt[k][i] /= tmp;
76              }
77
78              #pragma acc parallel loop vector_length(128)
79              for (int j = k + 1; j < cols; j++) {
80                  double tmp = 0.;
81                  for (int i = 0; i < rows; i++) tmp += Qt[k][i] *
                        Qt[j][i];
82                  for (int i = 0; i < rows; i++) Qt[j][i] -= tmp *
                        Qt[k][i];
83              }
84          }
85
86          #pragma acc parallel loop
87          for (int i = 0; i < rows; i++)
88          for (int j = 0; j < cols; j++)
89          Q[i][j] = Qt[j][i];
90      }
91  }
92
93  #else
94
95  // OMP/serial code for performance testing
96  void gramSchmidt(float Q[][COLS], const int rows, const int cols
        ) {
97      float Qt[cols][rows];
98
99      #pragma omp parallel for
```

```
100     for (int i = 0; i < rows; i++)
101     for (int j = 0; j < cols; j++)
102     Qt[j][i] = Q[i][j];
103 
104     for (int k = 0; k < cols; k++) {
105         double tmp = 0.;
106         #pragma omp parallel for reduction(+: tmp)
107         for (int i = 0; i < rows; i++) tmp += (Qt[k][i] * Qt[k][
                i]);
108         tmp = sqrt(tmp);
109 
110         #pragma omp parallel for
111         for (int i = 0; i < rows; i++) Qt[k][i] /= tmp;
112 
113         #pragma omp parallel for reduction(+: tmp)
114         for (int j = k + 1; j < cols; j++) {
115             double tmp = 0.;
116             for (int i = 0; i < rows; i++) tmp += Qt[k][i] * Qt[
                    j][i];
117             for (int i = 0; i < rows; i++) Qt[j][i] -= tmp * Qt[
                    k][i];
118         }
119     }
120 
121     #pragma omp parallel for
122     for (int i = 0; i < rows; i++)
123     for (int j = 0; j < cols; j++)
124     Q[i][j] = Qt[j][i];
125 }
126 #endif
127 
128 void printOctave(char *var, float A[][COLS], int rows, int cols)
        {
129     //  if((rows*cols) > 1000) return;
130     printf("%s=[\n",
131     var);
132     for (int i = 0; i < rows; i++) {
133         for (int j = 0; j < cols; j++)
134             printf("%c%e", (j == 0) ? ' ' : ',', A[i][j]);
135             printf(";");
136     }
137     printf("];\n");
138 }
139 void checkOctave(float A[][COLS], int rows, int cols) {
140     int found_error = 0;
141     for (int c1 = 0; c1 < cols; c1++)
142     for (int c2 = c1; c2 < cols; c2++) {
143         double sum = 0.;
144         for (int i = 0; i < rows; i++) sum += A[i][c1] * A[i][c2
                ];
145         if (c1 == c2) { // should be near 1 (unit length)
146             if (sum < 0.9) {
147                 printf("Failed unit length: %d %d %g\n", c1, c2,
                        sum);
148                 found_error = 1;
149                 exit(1);
150             }
151         }
```

```
152            else { // should be very small (orthogonal)
153                if (sum > 0.1) {
154                    printf("Failed orthonogonal  %d %d %g\n", c1, c2
                           , sum);
155                    found_error = 1;
156                    exit(1);
157                }
158            }
159        }
160        if (!found_error) printf("Check OK!\n");
161    }
162
163    int main(int argc, char * argv[]) {
164
165        mi = mallinfo();
166        mi1 = mi.uordblks;
167        mi3 = mi.hblkhd;
168
169        int rows = ROWS;
170        int cols = COLS;
171        float( * A)[COLS] = (float( * )[COLS]) malloc(sizeof(float)
               * rows * cols);
172
173        // fill matrix A with random numbers
174        for (int i = 0; i < rows; i++)
175        for (int j = 0; j < cols; j++)
176        A[i][j] = (float) rand() / (float) RAND_MAX;
177        printf("Done with init!\n");
178
179        //printOctave("A", A, rows, cols);
180        double startTime = omp_get_wtime();
181        gramSchmidt(A, rows, cols);
182        double endTime = omp_get_wtime();
183        printf("runtime %d %d matrix %g\n", rows, cols, (endTime -
               startTime));#ifndef NO_CHECK
184        //  checkOctave(A, rows, cols);
185        //printOctave("Q", A, rows, cols);
186        #endif
187        free(A);
188    }
```

APÊNDICE B – Resultado da compilação de um programa para resolução de um sistema linear por decomposição de Cholesky com OpenMP e o compilador pgcc.

cholesky:

10, include "cholesky.h"

36, Parallel region activated

Parallel loop activated with static block schedule 40, Begin critical section

End critical section

Barrier

Parallel region terminated

49, Parallel region activated

Parallel loop activated with static block schedule

55, Begin critical section

End critical section

Barrier

Parallel region terminated

84, Memory copy idiom, loop replaced by call to c_mcopy8_bwd

solucao_chol_12:

10, include "cholesky.h"

132, Memory copy idiom, loop replaced by call to c_mcopy8

156, Memory zero idiom, loop replaced by call to c_mzero8

162, Memory copy idiom, loop replaced by call to c_mcopy8_bwd

main:

744, Zero trip check eliminated