PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Programa de Pós-Graduação em Engenharia Elétrica

PSIVIA-HP: Paralelização do algoritmo SIVIA (*Set Inversion via Intervalar Analysis*) para Plataformas Heterogêneas

Luiz Guilherme Hilel Drumond Silveira

Belo Horizonte
2013

Luiz Guilherme Hilel Drumond Silveira

PSIVIA-HP: Paralelização do algoritmo SIVIA (*Set Inversion via Intervalar Analysis*) para Plataformas Heterogêneas

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de mestre.

Orientadores:
Prof. Dr. Carlos Augusto Paiva Silva Martins
Prof. Dr. Luís Fabrício Góes

Belo Horizonte
2013

FICHA CATALOGRÁFICA
Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

Silveira, Luiz Guilherme Hilel Drumond
S587p PSIVIA-HP: paralelização do algoritmo SIVIA (*Set Inversion Via Intervalar Analysis*) para plataformas heterogêneas / Luiz Guilherme Hilel Drumond Silveira. Belo Horizonte, 2013.
89 f. : il.

Orientador: Carlos Augusto Paiva Silva Martins
Coorientador: Luís Fabrício Góes
Dissertação (Mestrado) – Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Engenharia Elétrica.

1. Programação paralela (Computação). 2. Mineração de dados (Computação). 3. Algorítmos computacionais. 4. Arquitetura de computador. I. Martins, Carlos Augusto Paiva Silva. II. Góes, Luís Fabrício. III. Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Engenharia Elétrica. IV. Título.

CDU: 681.3.066

Luiz Guilherme Hilel Drumond Silveira

PSIVIA-HP: Paralelização do algoritmo SIVIA (*Set Inversion via Intervalar Analysis*) para Plataformas Heterogêneas

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de mestre.

_____

Prof. Dr. Carlos Augusto Paiva Silva Martins – PUC Minas (Orientador)

_____

Prof. Dr. Luís Fabrício Góes – PUC Minas (Orientador)

_____

Prof. Dr. Domingos da Costa Rodrigues – UFMG (Banca Examinadora)

_____

Profa. Dra. Rose Mary de Souza Batalha - PUC Minas (Banca Examinadora)

Belo Horizonte, 30 de Agosto de 2013.

AGRADECIMENTOS

Agradeço primeiramente a minha mãe por todo o seu exemplo de caráter e força**,** que norteiam a minha vida, e pela sua ajuda incondicional durante este percurso. Agradeço também ao Professor Carlos, um dos meus orientadores, por toda a sua orientação acadêmica, pela sua amizade e por sempre ter sido um pai na dimensão plena da palavra. Agradeço também ao Professor Luís Fabrício, pela sua orientação fundamental para o andamento do trabalho, como também pela sua amizade e conselhos que fogem do âmbito acadêmico.

Meus agradecimentos também, aos amigos: Milene Barbosa pela sua ajuda no início do trabalho e pelos seus exemplos; ao amigo Igor Mota Morici pela amizade e seu exemplo de profissional acadêmico; a Ludmila Murta pela amizade e exemplo de competência.

Não posso deixar de agradecer a Fernanda Camargo pelo seu exemplo, companheirismo e carinho que me ajudaram a seguir este caminho e a Maira Corrêa pelo seu carinho em um momento difícil deste percurso.

Finalizando, gostaria de agradecer a todos do PPGEE pela amizade, especialmente a Isabel Siqueira e Isabel Novaes pelo carinho e pelos momentos agradáveis.

RESUMO

Com a diversidade de plataformas paralelas disponíveis e a possibilidade de utilização do paralelismo para viabilizar a utilização de técnicas e aplicações que antes demandavam grande potencial computacional, é fundamental a possibilidade de poder integrar as mais diversas plataformas paralelas. Para tal, foi desenvolvido um modelo de programação paralela que provê interoperabilidade do seu código em plataformas paralelas heterogêneas denominada por OpenCL. Uma dessas técnicas numéricas que demanda grande poder computacional é o algoritmo SIVIA (*Set Inversion Via Interval Analysis*). Este algoritmo requer um alto nível de processamento estar baseado na matemática intervalar. O presente trabalho apresenta uma proposta de paralelização do algoritmo SIVIA em OpenCL para ser executado em plataformas heterogêneas. A abordagem adotada não foi a paralelização de algum módulo do algoritmo, mas a decomposição do espaço de busca, permitindo assim que desde a primeira iteração hajam intervalos sob avaliação. Para validação foram utilizadas três funções de complexidades distintas, a saber as funções polinomial, logarítmica e exponencial. As execuções em CPU apresentaram escalabilidade para todos os tipos de funções utilizadas, sendo a menor escalabilidade de 4 *Work-Items* na função exponencial. Em GPU o espaço de busca reduzido e o tamanho de grupo igual ao número de *Stream Processors* em um *Stream MultiProcessor* alcançaram os melhores resultados. Os ganhos máximos alcançados pelo *Kernel* do PSIVIA-HP em relação à execução sequencial foi de 51 vezes em CPU na função exponencial e 41 vezes em GPU na função logarítmica. Porém, aplicações em OpenCL não dependem apenas do desempenho do *Kernel*, pois há o tempo de preparação do dispositivo. Dessa maneira, com os compiladores e os dispositivos utilizados, o tempo de preparação do dispositivo em CPU foi maior do que em GPU, caracterizando uma dependência relativa aos compiladores OpenCL dos fabricantes dos dispositivos. Mesmo com esta dependência, o PSIVIA-HP se apresentou com uma abordagem que possibilita alto desempenho em relação a abordagens sequenciais.

Palavras-chave: Processamento Paralelo. Computação em Plataformas Heterogêneas. *Mulicore*. GP-GPU. OpenCL. Matemática Intervalar. Análise Intervalar.

ABSTRACT

With the diversity of parallel platforms available and the possibility of using parallelism to enable the use of techniques and applications that previously demanded great computational potential, the possibility of integrating several parallel platforms is essential. Therefore, parallel programming model was developed in order to provide interoperability of your code in heterogeneous parallel plataforms called OpenCL . One of these techniques that requires a large computational power is the algorithm Sivia (Set Inversion Via Interval Analysis) that requires high processing level since it is an algorithm based on interval mathematics. Thus, this monograph proposes a parallelization of the algorithm Sivia in OpenCL so it can be run on heterogeneous platforms. The chosen approach of the parallelization of the algorithm Sivia was not the parallelization of some modulus of algorithm, but the search space of the algorithm, enabling that since the first iteration there are intervals being evalueted. In the results were used three distinct functions with diverse complexities, being chosen the polynomial functions, logarithmic and exponential. For the polynomial function, performance results only in kernel CPU presented scalability up to 4-Work Items, in the exponential function scalability up to 8 Work-Items and scalability up to 8 Work-Items for the logarithmic function, and all of them in a search space starting with the lowest number of intervals that do not belong to the function. For GPU, with all the functions used, the best results of running kernel were for reduced search space and group size equal to the number of Stream Processors in a Stream MultiProcessor. Regarding the overall performance of the application for the polynomial function utilized, the gain obtained by using the PSIVIA-HP was 25 times in CPU and 51 times in GPU, to the exponential function the gain was of 26 times on CPU and GPU. For the logarithmic function the gain was of 14 times in CPU and 41 times in GPU. However, OpenCL applications do not depend only on the performance of the Kernel, because there is a time for the preparation of the device. In the compilers and devices used, the cost is in CPU is higher than in GPU regarding the preparation of the device for executing the code in OpenCL, leading to dependence of the performance of compilers OpenCL. Despite the dependence, the PSIVIA-HP presents itself with an approach that brings high performance in relation to sequential approaches.

Keywords: Parallel Processing. Computing in Heterogeneous Platforms. Multicore. GPGPU. OpenCL. Interval Mathematics. Interval Analysis.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE EQUAÇÕES

LISTA DE ABREVIATURAS

- GHz(*Giga Hertz*)
- EP (Elementos de processamento)
- GB (Giga Bytes)
- KB (Kilo Bytes)

LISTA DE SIGLAS

- GP-GPU(G*eneral-purpose computing on graphics processing units*);
- GPU (*Graphic Processor Unit*);
- OpenCL (*Open Computer Language*);
- SIVIA (*Set Inversion via Interval Analysis*);
- FPGAs (*Field-programmable gate array*);
- WSCAD(Workshop de Sistemas Computacionais de Alto Desempenho);
- ISCA (*International Symposium on Computer Architecture*);
- MPI(*Massage Passing Interface)*;
- DSM(*Distributed shared memory*);
- MIMD (*Multiple Instruction Multiple Data*);
- SIMD (*Singe Instruction Multiple Data*);
- CPU (*Central Processor Unit*);
- MIPS (*Microprocessor without Interlocked Pipeline Stages*);
- ISA (*Instruction Set Architecture*);
- LSDC (Laboratório de Sistemas Digitais e Computacionais);
- PPGEE (Programa de Pós Graduação em Engenharia Elétrica);
- GW (*Global Work Size*);
- LW (*Local Work Size*);
- WI (*Work-Item*);
- ULA (Unidade Lógico Aritmética);
- ILP (*Instruction Level Paralelism*);
- SISD (*Single Instruction Single Data*);
- SIMD (*Single Instruction Multiple Data*);
- MISD (*Multiple Instruction Single Data*);
- API (*Application Programming Interface*);
- MPI(Message Passing Interface);
- CUDA (*Compute Unified Device Architecture*);

SUMÁRIO

1. INTRODUÇÃO

Com a grande variedade de máquinas paralelas disponíveis e a crescente demanda de desempenho nas aplicações da classe de otimização e tomada de decisão, é necessário identificar qual máquina paralela é mais adequada e qual o melhor método de paralelização dos algoritmos. Por esta razão, nesta pesquisa, foi proposto, desenvolvido e verificado a paralelização do algoritmo SIVIA (*Set Inversion via Intervalar Analysis*) para a utilização em plataformas paralelas heterogêneas. A utilização de algoritmos em plataformas paralelas tem como objetivo melhorar o desempenho através da redução do tempo de execução por meio do uso do paralelismo. O paralelismo pode ser explorado em diferentes tipos de plataformas paralelas, porém com a diversidade de plataformas paralelas disponíveis é necessário identificar a mais adequada para cada aplicação.

Este primeiro capítulo é destinado a apresentar os tópicos relativos ao contexto, a justificativa, os objetivos, o escopo e a organização do texto.

1.1 Contexto

Até o final do século XX, o paralelismo era mais presente em centros de pesquisa e corporações que utilizavam supercomputadores pela alta demanda de processamento. Por outro lado, a comercialização de máquinas para usuários domésticos era predominante monoprocessadas. A evolução dos processadores, utilizados em centros de pesquisa e computadores domésticos, na época em questão, era o aumento da frequência dos processadores (HELD, BAUTISTA e KOEHL, 2006).

No final do século XX, os engenheiros começaram a lidar com problemas físicos, como a distância dos transistores na pastilha de silício e a dissipação do calor produzida pelos processadores. O aumento de transistores em uma mesma área de silício reduz a distância entre a fonte e o dreno, e a partir de uma determinada distância podem ocorrer flutuações na corrente, mesmo quando não houver tensão aplicada, levando o transistor a deixar de ser um dispositivo de processamento confiável (IRWIN, 2005). A solução encontrada pelos projetistas foi denominada de arquitetura de múltiplos núcleos, que diferem das máquinas monoprocessadas pela menor quantidade de transistores por núcleo, menor quantidade de

energia gerada pelo processador, maior quantidade de núcleos de processamento e frequência menor de cada núcleo. Apesar da frequência menor, o ganho no uso de arquiteturas de múltiplos núcleos se deve a possibilidade de processos ou partes dos processos, possam ser processados em núcleos distintos ao mesmo tempo.

No começo da década passada, a companhia de processamento de áudio BionicFX iniciou pesquisas em programação de propósito geral em placas gráficas para a sua arquitetura de software denominada AVEX (*Audio-Video EXchange*) (INTEL, 2010). A partir da possibilidade apresentada pela BionicFX, as empresas fabricantes de GPUs se voltaram para a perspectiva de utilizar o processamento vetorial de suas placas gráficas para processamentos de propósito geral. Esse novo modelo de paralelismo foi denominado de GP-GPU (*General Purpose – Graphic Processor Unit*) (INTEL, 2010).

Nos dias de hoje as máquinas de múltiplos núcleos e as GPUs são mais utilizadas por serem financeiramente mais acessíveis. Porém, cada máquina provê um tipo de paralelismo, sendo necessário o desenvolvedor identificar qual é o tipo necessário para a sua aplicação. Outra métrica importante nessa análise é a granularidade do processo, sendo que, arquiteturas de múltiplos núcleos são indicadas para paralelismo de grão grosso, onde o desenvolvedor deve analisar o seu código, identificar as partes não dependentes e dividi-las para serem processadas simultaneamente. Já às GPUs são indicadas para paralelismo de grão fino, onde uma mesma instrução é aplicada a um conjunto de dados diferentes (EL-REWINI e ABD-EL-BARR, 2005).

Cada fabricante disponibiliza uma linguagem de programação específica para o seu periférico, sendo necessário o desenvolvedor adquirir conhecimento dessa linguagem para desenvolver suas aplicações. Quando se deseja desenvolver para *desktops* projetados com arquiteturas de múltiplos núcleos, este problema não é expressivo, pois grande parte dos compiladores são voltados para as arquiteturas x86 e x64, comuns pelos fabricantes de processadores dominantes do mercado. Porém, em GPUs não há compartilhamento de instruções. Caso se deseje trabalhar com duas GPUs de diferentes fabricantes, será necessário desenvolver o mesmo algoritmo para as linguagens de cada fabricante. Além, que o próprio desenvolvedor deverá criar a forma de comunicação do mesmo algoritmo sendo processado em placas de fabricantes distintos. Este problema permanece quando se deseja trabalhar com plataformas heterogêneas (CPU (*Central Processing Unit*) e GPU (*Graphic Processor Unit*) ao mesmo tempo), onde, além do problema das linguagens supracitado, deve se ater a sincronização, comunicação, modelo arquitetural e divisão da carga de trabalho.

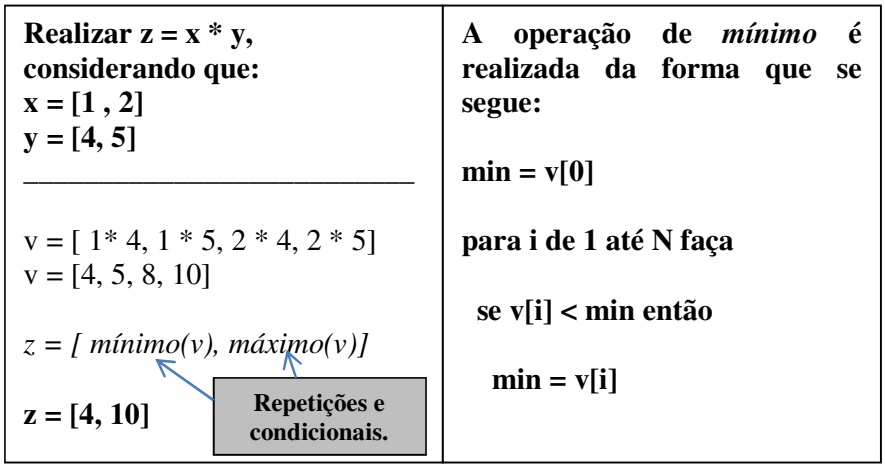
1.2 Justificativa

Com a dificuldade de se desenvolver uma aplicação que trabalhe em plataformas heterogêneas, no ano de 2008 um consórcio foi formado para desenvolver um padrão de programação que solucionasse esta questão. Este consórcio denominado *KHRONOS GROUP* (KHRONOS, 2013) formado por grandes empresas fabricantes de periféricos, desenvolveram o OpenCL (*Open Computer Language*), um padrão de programação paralela para plataformas heterogêneas. Inicialmente, o foco do OpenCL era em *multicores* ou CMPs (*Chip-level Multiprocessing*) e GP-GPUs, sendo a sua principal característica a possibilidade de um mesmo código desenvolvido em OpenCL poder ser executado em CPU, GPU ou em ambas as plataformas simultaneamente, sem a necessidade de recodificação independentemente do fabricante. Apesar do foco inicial ter sido nas plataformas mencionadas, é válido ressaltar que já há resultados de pesquisas de execução de códigos OpenCL em FPGAs (*Field-programmable gate array*), como CHEN e SINGH (2013) que realizaram uma avaliação de Compressão de Fractal em OpenCL nas plataformas CPU, GPU e FPGA.

Técnicas paralelas podem ser empregadas na classe de algoritmos baseada na matemática intervalar, a qual demanda alto processamento. A matemática intervalar é uma área da matemática voltada para problemas, onde, não é possível determinar valores pontuais, sendo necessário trabalhar com intervalos. A matemática intervalar provê uma aritmética que permite realizar operações básicas em intervalos como a soma, subtração, multiplicação e divisão e funções mais complexas como exponencial, logarítmica, entre outras. A aritmética intervalar provê também operações com caixas (conjunto de intervalos), necessárias para algoritmos que trabalhem com múltiplas dimensões. A principal operação realizada em caixas é a bisseção, responsável por dividir o intervalo pela metade, gerando novos intervalos.

As operações da matemática intervalar são compostas por conjuntos de operações da matemática clássica, implicando então em uma maior necessidade de processamento. Por esta razão, algoritmos intervalares como o SIVIA (*Set Inversion via Interval Analysis*) demandam maior processamento do que algoritmos pontuais. Um exemplo que pode ser utilizado para ilustrar esta alta demanda de processamento é a operação básica de multiplicação intervalar, que além de ser composta de multiplicações simples, possuem ainda condicionais e repetições (JOULIN, 2004). A figura 1 apresenta um exemplo da operação de multiplicação e os testes realizados.

Figura 1. Exemplo de operação de multiplicação intervalar.



Como é possível perceber no exemplo da figura 1, são realizadas multiplicações entre os valores de mínimo e máximo dos dois intervalos. Porém, é necessário encontrar o mínimo e o máximo destas multiplicações, sendo necessário realizar o percorrimento do vetor *v* através de uma repetição e a utilização de uma condicional para determinação os valores. É válido ressaltar que, em execução, caso ocorra uma predição equivocada em um *pipeline* quando estas condicionais estão sendo processadas, o *pipeline* terá de ser esvaziado e o desempenho será prejudicado (PATTERSON e HANNESSY, 2005). A figura 1 omite a operação de máximo, pois a diferença fundamental é apenas o sinal de maior ao invés do menor no teste do SE.

Apesar da sua demanda de processamento, Algoritmos Intervalares como o SIVIA já foram utilizados em pesquisas para navegação de veículos inteligentes (DREVELLE e BONNIFAIT, 2013) e em conjunto com *Fuzzy Sets* (MAZEIKA, JAULIN e OSSWALD, 2007). O SIVIA original apresentado em JOULIN (1992) e a sua utilização nos trabalhos descritos é uma abordagem sequencial do algoritmo. A sua utilização pode ser inviável com problemas de alta exatidão, que demandam alto processamento, justamente pela sequencialidade de sua abordagem.

1.3 Objetivos

Baseado nos problemas apresentados anteriormente, o objetivo geral da pesquisa é propor e desenvolver um algoritmo SIVIA paralelo em OpenCL que permita a execução em plataformas paralelas heterogêneas (processadores de propósito geral e processadores gráficos) elevando o desempenho em relação às abordagens sequenciais disponíveis.

1.3.1 Objetivos Específicos

Com vista nos problemas apresentados destacam-se os seguintes objetivos principais da pesquisa exposta nesta dissertação.

a) Definir, propor e desenvolver uma biblioteca com os operadores e funções intervalares básicas em C99 permitindo compatibilidade com o OpenCL;
b) Desenvolver, avaliar e analisar o algoritmo SIVIA paralelo em OpenCL para plataformas heterogêneas.

1.4 Escopo

Os resultados deste trabalho serão focados no ganho de desempenho em relação ao tempo de execução de uma instância do problema. Não será abordada nesta dissertação a vazão, ou seja, a quantidade de instâncias do problema que podem ser solucionadas por execução do PSIVIA-HP (*Parallel Set Inversion via Intervalar Analysis – Heterogeneous Platform*).

Toda a avaliação será direcionada para identificar qual dispositivo paralelo é mais adequado para a execução do PSIVIA-HP, sendo que, a avaliação do desempenho em plataformas heterogêneas simultâneas não será abordada.

1.5 Organização do Texto

O restante do texto desta dissertação está organizado da seguinte forma: no capítulo 2 é apresentada a revisão da literatura sobre os tipos de paralelismo, sobre as Arquiteturas Paralelas, os modelos de programação paralela e os trabalhos relacionados; no capítulo 3 são apresentados os principais conceitos subjacentes ao algoritmo SIVIA sequencial e a implementação do PSIVIA-HP que é a proposta de solução desta dissertação; no capítulo 4 são apresentadas as etapas da pesquisa, a configuração dos experimentos e a verificação dos experimentos; no capítulo 5 são apresentados os resultados e análise de desempenho do PSIVIA-HP; no capítulo 6 são apresentados a conclusão acerca dos principais resultados obtidos, as principais contribuições da pesquisa e os possíveis trabalhos futuros. Após o capítulo 6 são descritas as referências bibliográficas utilizadas na pesquisa.

2. REVISÃO DA LITERATURA

Este capítulo é destinado à apresentação da revisão da literatura sobre os tipos de paralelismo, a classificação de arquiteturas paralelas, os modelos de programação paralela e os trabalhos relacionados a esta pesquisa.

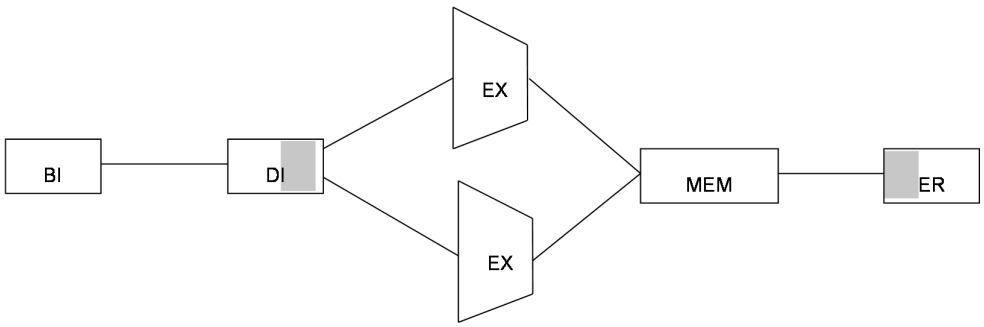
2.1 Tipos de Paralelismo

Em computação paralela existem diferentes níveis de paralelismo e para cada nível uma abordagem pode ser aplicada para elevar o desempenho. O primeiro nível de paralelismo é denominado ILP (*Instrucition-Level Parallelism*) e pode ser aplicado tanto em hardware como em software (PATTERSON e HANNESSY, 2005).

O uso do ILP em hardware é comum nos processadores de hoje, estando presente em todos os projetos de processadores criados a partir da década de 90. Uma das técnicas de ILP mais comum é o *pipeline*. O *pipeline* consiste na divisão do processador em estágios, permitindo que mais de uma instrução seja processada por unidade de tempo (PATTERSON e HANNESSY, 2005).

Uma das evoluções da técnica do *pipeline* é denominada *pipeline* superescalar, que foca na replicação dos recursos para permitir que mais de uma instrução seja executada no mesmo estágio de processamento. A figura 1 representa um exemplo de *pipeline* superescalar (PATTERSON e HANNESSY, 2005) (EL-REWINI e ABD-EL-BARR, 2005).

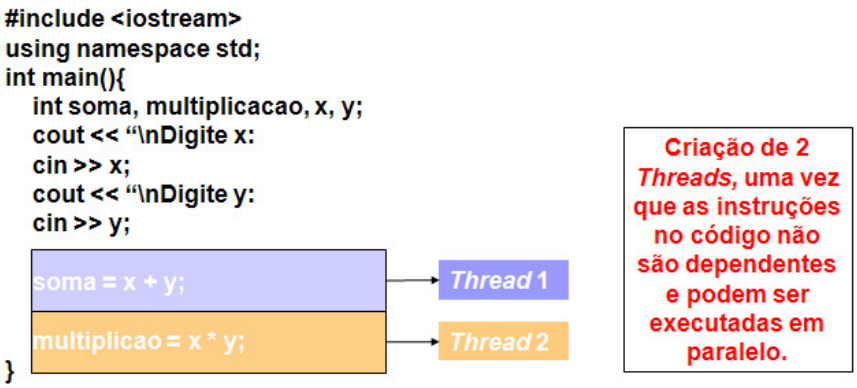
Figura 2. Exemplo de Pipeline Superescalar em MIPS.



Na figura 1 o estágio de EX (Execução) foi replicado, permitindo, por exemplo, que um recurso seja alocado para cálculos de números inteiros e o segundo para cálculos de ponto flutuante. Sendo assim, seria possível que uma instrução inteira e outra de ponto flutuante fossem processadas ao mesmo tempo. É importante ressaltar que as implementações de arquiteturas Superescalares devem permitir a busca de mais de uma instrução e a leitura de vários operandos do banco de registradores por cada pulso de *clock*. Contudo, as limitações das arquiteturas Superescalares estão nas ocorrências de dependências diretas e em falhas de predição de desvios, que são mais complexos de serem tratados do que em arquiteturas não Superescalares. Isso implica adicionar ao projeto mais hardware, tornando mais complexo, para o tratamento dessas questões (PATTERSON e HANNESSY, 2005).

Na perspectiva da aplicação existem dois conceitos fundamentais denominados de paralelismo de tarefas e paralelismo de dados. Porém, antes de abordar ambos os conceitos há a necessidade de conceituar *threads* (processos leves), fundamentais na implementação de aplicações paralelas. Caso, no momento da concepção do código, o desenvolvedor identificar porções que não são dependentes, será possível implementar o código através de partes menores, por consequência mais leves (*threads*), que podem ser executadas paralelamente (TANENBAUM, 2010). A figura 2 apresenta um exemplo de código que pode ser implementado com *threads*.

Figura 3 Exemplo de um código que pode ser paralelizado com threads em C++.



No código apresentado na figura 2, as variáveis soma e multiplicação utilizam os operandos x e y, mas não há dependência direta, uma vez que a operação de multiplicação não utiliza o valor da soma calculada da linha anterior. Como não há dependência direta, as operações podem ser processadas paralelamente. Sendo assim, cada operação (soma e multiplicação) podem ser programadas em *threads* e cada *thread* ser processada em núcleos de processamento distintos. Caso a aplicação não seja desenvolvida com este paradigma, todo

o processo será atribuído para apenas um núcleo de processamento, não utilizando todo o recurso que as máquinas de múltiplos núcleos dispõem. É válido ressaltar dois aspectos: o primeiro é que a figura 2 é apenas um exemplo de possibilidade de criação de *threads* já que não existe dependência, pois a criação de *threads* gera custo, sendo o desenvolvedor responsável por avaliar se a aplicação demanda processamento suficiente para que tenha validade a sua paralelização. O outro aspecto que deve ser ressaltado é que apesar de *threads* ser um conceito comum no paralelismo de dados e tarefas, o exemplo da figura 2 é sobre a aplicação no paralelismo de tarefas.

Diante dos conceitos sobre *threads* apresentados, paralelismo de tarefas pode ser definido como porções de códigos distintas, sem dependência, que podem ser processadas paralelamente.

O paralelismo de dados é caracterizado por uma grande massa de dados a ser processada com poucas operações a serem aplicadas a esta massa. Um exemplo que pode auxiliar na compreensão deste conceito seria a soma de duas matrizes com alta ordem de grandeza, onde apenas uma operação, soma, é aplicada a cada elemento das matrizes. Para cargas de trabalho com esta característica é indicado à utilização de arquiteturas vetoriais, como GPUs, que possibilitam a aplicação de uma mesma instrução em um conjunto de dados distintos. (AMD,2010) (EL-REWINI e ABD-EL-BARR, 2005).

2.2 Arquiteturas Paralelas

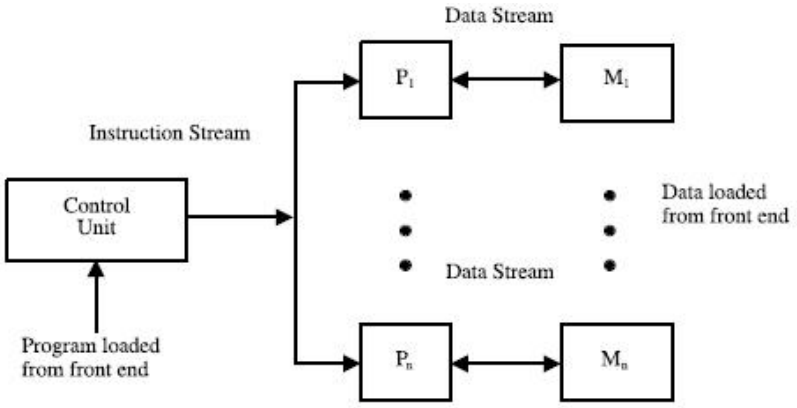
Em arquitetura de computadores há uma classificação de máquinas paralelas baseada no fluxo de instruções e dados do processador, denominada Taxonomia de Flynn apresentada por Flynn (1966). Nessa Taxonomia as máquinas paralelas são classificadas em quatro categorias: SISD (*Single Instruction Single Data*); SIMD (*Single Instruction Multiple Data*); MISD (*Multiple Instruction Single Data*) e MIMD (*Multiple Instruction Multiple Data*).

A classificação SISD é considerada a máquina de Von Neumann tradicional, composta por apenas um caminho de dados, uma unidade de controle e memória (FLYNN, 1966) (EL-REWINI e ABD-EL-BARR, 2005) (TANENBAUM, 2007), como eram as máquinas monoprocessadas até o final do século passado. As máquinas SISD consistem em aplicar apenas uma instrução em um dado por ciclo de *clock*. Porém, é válido ressaltar que essas

máquinas não eram compostas de nenhum nível de paralelismo, pois como já citado neste texto, a técnica de pipeline se tornou comum nos projetos de processadores na década de 90 do século passado.

Máquinas SIMD são adequadas para cargas de trabalho com características de paralelismo de dados, onde uma operação é aplicada a uma grande quantidade de dados, como o exemplo de soma de matrizes já mencionado. Para que seja possível o processamento, máquinas SIMD são compostas por apenas uma unidade de controle, responsável por decodificar a instrução e de um hardware que permita múltiplas leituras do banco de registradores (FLYNN, 1966). Cada EP em máquinas SIMD possui uma pequena quantidade de memória rápida e limitada de acesso exclusivo do EP. Há também em sua arquitetura uma memória global que é compartilhada por todos EPs, mais lenta e de maior capacidade quando comparada à memória exclusiva de cada EP (EL-REWINI e ABD-EL-BARR, 2005) (TANENBAUM, 2007). A figura 3 apresenta um diagrama da composição de máquinas SIMD.

Figura 4. Diagrama de uma máquina SIMD de (EL-REWINI e ABD-EL-BARR, 2005).



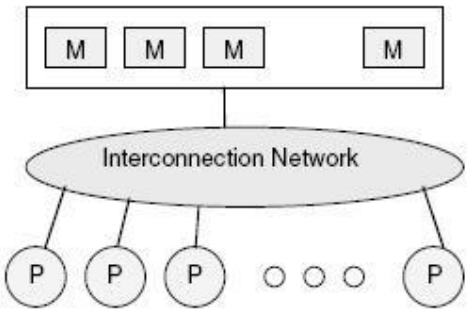
Cargas de trabalho da área de computação gráfica e processamento digital de imagens possuem alto nível de paralelismo de dados (AMD, 2010), por esta razão, as GPUs foram concebidas nesta arquitetura. A operação de renderização, responsável por atribuir camadas em objetos, por exemplo, é uma típica aplicação de paralelismo de dados, já que a renderização é aplicada a todos os objetos no cenário em 3D.

As principais denominações da arquitetura SIMD em GPU são: cada EP ((Pn) na figura 3) é denominado de SP (*Stream Processor*), sendo que o agrupamento de SPs é denominado de SMPs (*Stream Multiprocessors*) (EL-REWINI e ABD-EL-BARR, 2005) (AMD, 2010).

Máquinas denominadas MISD são caracterizadas por um fluxo de dados passando por um vetor de processadores, executando diferentes fluxos de instruções (FLYNN, 1966). Na prática não existem máquinas MISD, porém alguns autores consideram máquinas paralelas montadas para trabalhar como *pipeline* como máquinas MISD (EL-REWINI e ABD-EL-BARR, 2005). É válido ressaltar que cada unidade em uma máquina MISD pode ser composta por máquinas SISD, SIMD e MIMD (EL-REWINI e ABD-EL-BARR, 2005) (TANENBAUM, 2007).

Máquinas de múltiplos núcleos e *clusters* são consideradas máquinas MIMD, que são utilizadas quando a carga de trabalho é caracterizada com paralelismo de tarefas. Máquinas MIMD podem ser ainda classificadas nos modelos, memória compartilhada e memória distribuída (EL-REWINI e ABD-EL-BARR, 2005) (TANENBAUM, 2007). Nas máquinas de memória compartilhada cada EP possui o seu próprio caminho de dados e uma memória que é compartilhada por todos os EPs. Assim, a comunicação entre processos ou *threads* nesse modelo é realizada através de escrita e leitura na memória global (EL-REWINI e ABD-EL-BARR, 2005) (TANENBAUM, 2007). A figura 4 apresenta o modelo da memória compartilhada.

Figura 5. Arquitetura MIMD - memória compartilhada (EL-REWINI e ABD-EL-BARR, 2005).



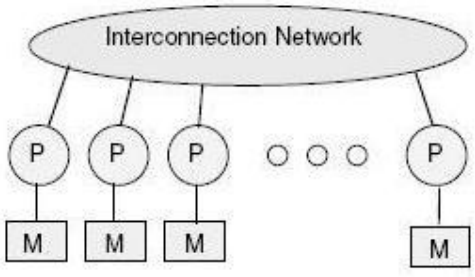
Shared Memory MIMD Architecture

A comunicação no modelo memória compartilhada, conforme apresentado na figura 4, é através de uma rede de interconexão, permitindo assim que todos os EPs possuam iguais chances de acesso e velocidade à memória. O gargalo deste modelo é baseado na velocidade da memória e a quantidade de requisições, já que às requisições são controladas pela unidade de controle da memória (EL-REWINI e ABD-EL-BARR, 2005).

Na arquitetura de memória distribuída, cada EP é composto por conjuntos de processadores e memórias privativas, sendo interligados através de uma rede de interconexão sem a presença de uma memória global. Como a memória é privativa de cada EP, para um EP

requisitar um dado que não esteja em sua memória, é necessário o envio de mensagens através da rede de interconexão requisitando o dado presente na memória de outro EP. As mensagens são enviadas e recebidas através de comandos *send/receive* explícitos no código da aplicação (EL-REWINI e ABD-EL-BARR, 2005). A figura 5 apresenta o modelo de memória distribuída.

Figura 6. Arquitetura MIMD – Memória Distribuída (EL-REWINI e ABD-EL-BARR, 2005).



Message Passing MIMD Architecture

Um exemplo desse tipo de modelo são os *clusters*, que apresentam alta possibilidade de escalabilidade, pois a adição de EPs se baseia na inclusão de novas máquinas. O gargalo desse modelo está na vazão da rede de interconexão que conecta os EPs. Caso o tráfego seja alto, técnicas que melhoram a comunicação devem ser empregadas para possibilitar o uso deste modelo (EL-REWINI e ABD-EL-BARR, 2005). Os *Clusters*, por exemplo, utilizam uma rede de interconexão (*ethernet*, *gigabits*, *infiniband*) para interligar os EPs, caso a comunicação seja alta, as técnicas podem ser empregadas em qualquer camada do protocolo de comunicação de rede utilizado, ou até mesmo no nível da aplicação.

2.3 Modelos de Programação Paralela

Em programação de arquiteturas paralelas, o desenvolvedor deve entender qual arquitetura irá executar o seu código, para que assim possa escolher o modelo adequado para a programação, já que cada arquitetura possui seus modelos exclusivos. Para a programação em memória compartilhada, existe a programação de *threads* por suporte da própria linguagem, onde o desenvolvedor explicitamente descreve todo o código da *thread*. Por não ser trivial esse tipo de programação, foi desenvolvida uma API (*Application Programming Interface*) para C e Fortran que permite maior facilidade de programação para o modelo de memória compartilhada, denominado de OpenMP (*Open Multi-Processing*) (OPENMP,

2013). Apesar de utilizar alguns métodos para propiciar certos aspectos dinâmicos, a programação do OpenMP é baseada em diretivas de compilação. O aspecto que deve ser mais ressaltado na programação em OpenMP, que propicia maior facilidade na programação é a possibilidade da paralelização da estrutura de repetição *for*, comum em cargas de trabalho com característica de paralelismo de dados. A figura 6 apresenta um trecho de código de exemplo de paralelização do *for* contido em OpenMP (2013).

Figura 7. Exemplo de paralelização de um *for* (OpenMP, 2013).

```
#pragma omp parallel shared(a,b,c,chunk) private(i)
  {

  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

  }  /* end of parallel section */
```

No trecho de código da figura 6, um vetor resultante recebe a soma de outros dois vetores de ordem *n*. A facilidade de programação é demonstrada pela inclusão de apenas duas linhas de código, sendo que a segunda contém a variável *chunk* que informa qual a quantidade de iterações que cada *thread* irá processar (OPENMP, 2013).

Quando se pensa na programação no modelo de passagem por mensagem, o padrão mais conhecido em C é o MPI (*Message Passing Interface*). Todas as mensagens enviadas e recebidas de cada instância da aplicação devem ser explicitamente codificadas pelo desenvolvedor. No MPI há obrigatoriedade da utilização de um aplicativo para a configuração do *cluster* e o disparo da aplicação para execução, como o MPICH (MPICH, 2013). Apesar da obrigatoriedade da utilização de um aplicativo, sua utilização pode propiciar que cada máquina que compõe o *cluster* possua qualquer sistema operacional.

O primeiro modelo de programação para GPU amplamente conhecido foi o CUDA (*Compute Unified Device Architecture*) para programação de GPUs da fabricante NVidia (NVIDIA, 2013). A facilidade de programação em CUDA é um de seus pontos positivos, onde o *Kernel* (código que será executado na GPU) é uma função dentro de um código em C, sendo apenas necessário chamar a função para a sua execução. Outra grande vantagem do CUDA em relação a sua programação é a possibilidade de alocação dinâmica dentro do código do *Kernel*, não sendo necessário prever qual é a quantidade de memória que irá ser utilizada, levando a um melhor aproveitamento dos recursos. Em relação ao tempo de execução de uma aplicação em CUDA há um fator positivo, pois o código de preparação do dispositivo e transferência de dados para a GPU é compilada juntamente com o *Kernel*.

Contudo, quando à análise é sobre a disponibilidade de dispositivos paralelos, o CUDA possui a desvantagem de que seus códigos podem ser executados apenas em placas da Nvidia. Essa desvantagem aumenta caso seja necessária uma aplicação que execute em CPU e GPU, pois será necessário integrar o CUDA com outro modelo de programação para memória compartilhada, como o OpenMP.

Até o final do século passado, as máquinas paralelas mais difundidas eram as máquinas multiprocessadas e os *clusters*, porém com o passar dos anos surgiram as GPUs com a possibilidade de programação para propósito geral e os FPGAs. Com essa diversidade de plataformas, uma máquina paralela que fosse composta por CPU, GPU e FPGA, por exemplo, seria necessário desenvolver três aplicações na linguagem específica de cada dispositivo e ainda um mecanismo de comunicação entre essas aplicações. Diante dessa diversidade de plataformas, o consórcio *Khronos Group* formado pelas principais fabricantes de periféricos do mercado, desenvolveu um padrão de programação para plataformas heterogêneas denominado OpenCL (KHRONOS, 2013).

O OpenCL tem como objetivo a portabilidade, ou seja, um código desenvolvido em OpenCL será processado em qualquer dispositivo independente do fabricante. Essa independência se restringe apenas entre os fabricantes que desenvolvem compiladores e suporte para OpenCL para os seus dispositivos (KHRONOS, 2013).

Para a programação em OpenCL é necessário trabalhar com os conceitos de *Kernel e Host,* que serão descritos em sequência.

O *Kernel* é o código em OpenCL que será executado no dispositivo, independente se for GPU, CPU, FPGA, etc. Da mesma maneira que um programa quando executado passa a ser denominado como processo (TANNENBAUM, 2007), o *Kernel* em execução passar a ser denominado como WI (*Work-Item*). A quantidade de WIs que serão executados é um dos parâmetros do OpenCL determinado pelo desenvolvedor. É válido ressaltar que o termo WI é utilizado apenas em OpenCL, sendo que na literatura as instâncias de códigos a serem processadas em paralelo são denominadas *threads* (TANENBAUM, 2007). Essa nomeclatura não ocorre apenas em arquiteturas de memórias compartilhadas, uma vez que a Nvidia denomina cada instância do seu *Kernel* como *thread* (NVIDIA, 2013).

Diferindo do CUDA, em que o *Kernel* é compilado juntamente com o código de preparação do dispositivo, o *Kernel* do OpenCL é compilado em tempo de execução. Ou seja, após o código de preparação do dispositivo ser compilado em qualquer API, quando esse código executável está em execução, é chamado o compilador OpenCL para a compilação do *Kernel*. É válido ressaltar que o OpenCL permite a utilização de um binário de um *Kernel* já

compilado para diminuir o tempo de execução da aplicação. Porém, esse binário só é criado após uma compilação em tempo de execução (AMD, 2010).

O *Host* é um código em C com métodos da biblioteca do OpenCL responsável por preparar o dispositivo para a execução. Essa preparação é necessária pelo fato de o OpenCL prover portabilidade do seu código, sendo necessário identificar qual o dispositivo e qual a plataforma OpenCL estão disponíveis para compilação e execução. Até o momento da escrita desta dissertação a versão disponível do OpenCL está na versão 1.2, porém a versão da plataforma implementada depende do fabricante do dispositivo. É necessário ressaltar que o *Host* é executado na CPU e é no *Host* que está à função de chamada para a execução do *Kernel* (AMD, 2010).

Para a execução do *Kernel* em OpenCL é necessário definir os parâmetros GW (*Global Work size*) e o LW (*Local Work Size*). O GW indica qual é a quantidade total de WIs que serão executados, sendo este número geralmente o total de EPs em CPU e por haver um *pipeline* implementando em cada SP, quatro vezes a quantidade de EPs em GPU (AMD, 2010). A quantidade de WIs igual ao número de EPs disponíveis, são para CPUs dedicadas para processar a aplicação em OpenCL, pois valores de GW superiores ao número de EPs irão gerar concorrência entre os WIs por recursos. Para as CPUs que possuam grande quantidade de processos, é válido atribuir mais WIs do que EPs disponíveis. Esse aumento, eleva a probabilidade dos WIs da aplicação em OpenCL serem escalonados para serem processados em algum núcleo da CPU. Os WIs são processados em grupos, onde o tamanho desses grupos são determinados pelo valor do LW. Ao ser processado, cada grupo é atribuído para um SMP na GPU ou para um núcleo de processamento da CPU. A determinação do valor do LW para GPU depende da especificação da placa da quantidade de SPs nos SMPs, já em CPU o LW sempre é 1, uma vez que, por núcleo da CPU só há um caminho de dados (AMD, 2010). A quantidade de grupos criados é dada pela razão do GW pelo LW.

O OpenCL trabalha com quatro modelos de memória: *Global*, *Local*, *Private* e *Constant*. Toda a alocação de *buffers* do *Host* para o dispositivo é realizada na área de memória *Global* do OpenCL, sendo que fisicamente essas alocações são realizadas na memória *Global* da GPU e na memória primária da CPU. A área de memória *Global* pode ser acessada por todos os WIs, sendo esta a forma de comunicação entre WIs de diferentes grupos. A área de memória *Local* é efetiva apenas em GPU, já que é uma memória acessada apenas pelos WIs de um grupo, lembrando que o tamanho de um grupo em CPU é sempre 1, sendo esta a razão de não ser efetivo em CPU. A área de memória *Private* é acessível apenas pelo próprio WI, sendo que qualquer variável declarada no corpo do *Kernel* é considerada

como *private*. A área de memória destinada ao *Constant* é utilizada para trabalhar com constantes dentro do *Kernel* e é visível por todos os WIs (AMD, 2010).

Em seguida serão apresentados os trabalhos relacionados à pesquisa desenvolvida.

2.4 Trabalhos Relacionados

Nesta seção serão apresentados os principais trabalhos relacionados ao tema de pesquisa desta dissertação, sendo apresentado primeiro o artigo original sobre o algoritmo SIVIA. Após, trabalhos com temas que permeiam a aplicação do SIVIA em versões paralelas, trabalhos que visam ganho com a utilização do OpenCL, trabalhos sobre ganho na utilização de GPU e trabalhos com a aplicação do SIVIA sequencial.

Jaulin e Walter (1993) aplicaram e apresentaram o conceito de inversão de conjuntos através da análise de intervalos (SIVIA) para o problema de estimativa de erro de fronteiras não lineares. Na época da publicação do artigo de Jaulin e Walter (1993), o problema de estimativa de erro para equações lineares já se encontrava solucionado, porém não para sistemas não lineares. Neste documento Jaulin e Walter (1993) propõem um algoritmo baseado em análise intervalar capaz de determinar partes internas e externas das caixas (conjunto de intervalos) através da operação de união. A partir deste algoritmo é que foi desenvolvido o PSIVIA-HP, uma vez que este trabalho é a apresentação original do SIVIA sequencial.

Drevelle e Bonnifait (2013) pesquisaram um mecanismo de navegação de veículos inteligentes, sendo que o principal problema desse tema é a exatidão da posição do veículo para que possa trafegar sem a possibilidade de colidir com obstáculos. Para tal o algoritmo SIVIA foi aplicado como parte de uma técnica proposta, sendo responsável pela computação do posicionamento do objeto. Para que fosse possível a utilização da técnica em tempo real, o SIVIA foi implementado *multithreaded* em C++. Porém, é válido e necessário ressaltar que no artigo não é apresentada nenhuma análise de desempenho da versão do SIVIA *multithreaded*, apenas cita que o SIVIA na técnica proposta foi implementada *multithreaded*. No artigo os resultados apresentados em relação à utilização do SIVIA é em relação à exatidão de posicionamento em uma simulação de um veículo em três voltas no trajeto fechado de 1 Km de uma vila de Paris.

Fractais é uma técnica de compressão efetiva que apesar do resultado final de qualidade, demanda altos recursos computacionais na parte de decodificação do algoritmo. Justamente em relação ao seu alto custo computacional essa técnica passou a não ser escolhida para utilização, já que o algoritmo *Fractal* original é sequencial. Mas, com a possibilidade de execução de algoritmos em plataformas heterogêneas, Chen e Singh (2013), propuseram um algoritmo de compressão de vídeo em OpenCL utilizando *Fractais* em tempo real. Os experimentos foram realizados com uma CPU Intel Xeon W3690 de 6 núcleos de 3.46GHz cada com *hyperthreading*, uma GPU NVIDIA C2075 Fermi e um FPGA Stratix V 5SGXA7. Os principais resultados alcançaram um ganho de 3 vezes da utilização do FPGA utilizado em relação à GPU utilizada e de 114 vezes do FPGA em relação a CPU utilizada. Porém, não há nenhuma análise sobre o tempo de preparação dos dispositivos para execução em OpenCL, apenas o tempo de *Kernel* e transferência.

Fraire, Ferreyra e Marques (2013) apresentam uma visão geral sobre OpenCL e suas implementações, e para tal, utilizaram como estudo de caso uma multiplicação de matrizes na ordem de 1024 x 1024. O ambiente para geração dos resultados foi composto por uma CPU Intel(R) Core(TM) i7 CPU Q 720 @ 1.60GHz de 8 núcleos e uma GPU ATI Radeon 5870 Mobility com 7 *Compute Units*. Os resultados foram comparados com um código sequencial implementado em C++ chegando a alcançar um ganho de 2 vezes com a versão em OpenCL executada em CPU e um ganho de 10000 vezes com a utilização do código em OpenCL executado na GPU. Para a CPU utilizada que possui quatro núcleos físicos, simulando 2 lógicos cada, totalizando oito núcleos para execução, o ganho conseguido pelos autores é considerado baixo. Provavelmente, por alguma otimização implícita na geração do código C++ do compilador utilizado ou pela implementação do *Kernel* gerado tenha sido focado para a execução em GPU e não em CPU.

Rocha e outros (2008) apresentaram em seu trabalho uma avaliação de desempenho da utilização de GPU. A avaliação de desempenho foi realizada comparando códigos de multiplicação de matrizes, sendo executados em uma GPU GeForce8800 GTS, com 12 SMPs possuindo 8 SPs cada, e em um processador AMD M2X2 64bits 6000+ 3000 GHz.

Os resultados de Rocha e outros (2008) mostram que é real o ganho do uso de arquiteturas paralelas, uma vez que o ganho no processamento foi aproximadamente de 75 vezes em relação à execução sequencial quando utilizado um processador *dual-core*. Quando executado em GPU, o ganho foi de 76 vezes. Os resultados, como supracitado, confirmam o uso de GPU para cargas de trabalho caracterizadas como paralelismo de dados e com alto volume de dados para processamento. Entretanto, cargas sem estas características não

alcançarão ganho com a utilização de GPU, já que há o custo para carregar a memória da GPU com instruções e dados, sendo este um dos limitadores para a utilização de GPU.

O SIVIA foi utilizado por Mazeika, Jaulin e Oswald (2007) para trabalhar como parte de um método que visa a utilização de conjuntos *Fuzzy* para funções multidimensionais contínuas, expressões não lineares, entre outras. O SIVIA foi utilizado na avaliação e classificação de fronteiras de conjuntos. Não há nenhuma descrição do código da implementação ou do ambiente em que os resultados foram gerados. Não sendo o objetivo do trabalho de Mazeika, Jaulin e Oswald (2007) o ganho de desempenho de toda a aplicação é justificada apenas a apresentação dos resultados da simulação em relação à melhora na classificação. Porém, isto impede que profissionais da área da computação paralela avaliem o que foi realizado para propor melhorias futuras de desempenho.

3. PARALLEL SIVIA PARA PLATAFORMAS HETEROGÊNEAS

Este capítulo é destinado a apresentar, conceitos iniciais sobre matemática intervalar, as principais características do algoritmo SIVIA sequencial de Joulin (1993) e a descrição da abordagem paralela adotada para gerar o PSIVIA-HP.

3.1 Conceitos Iniciais de Matemática Intervalar

Toda esta seção é baseada no documento de Joulin (2004).

O conceito fundamental na matemática intervalar é a noção de intervalo. Um intervalo é um subconjunto do domínio em $\mathbb{R}$, sendo que o conjunto de todos os intervalos de $\mathbb{R}$ é denotado como $\mathbb{IR}$.

Um intervalo $[x]$ é delimitado por limites, inferior e superior, que são definidos por:

$$x^- = \text{limite\_inferior}([x]) = \inf \{x | x \in [x] \};$$
$$x^+ = \text{limite\_inferior}([x]) = \inf \{x | x \in [x] \}.$$

Equação 1. Definições de Intervalo.

Por convenção o limite inferior e superior de um intervalo vazio $\{\emptyset\}$ é considerado infinito, como se segue:

$$\text{limite\_inferior}(\emptyset) = -\infty \text{ e limite\_superior}() = (\emptyset) = +\infty$$

Equação 2. Definição para intervalos vazios.

O tamanho de um intervalo é dado pela subtração do limite superior pelo limite inferior:

$$w(x) = x^+ - x^-$$

Equação 3. Tamanho de um intervalo.

Do mesmo modo da matemática clássica, a matemática intervalar possui as suas operações básicas que são constituídas por soma, subtração, multiplicação, divisão, máximo e mínimo. Logo, se $\Diamond \in \{+,-,*,/,\max,\min\}$, onde * é a multiplicação e se $[x]$ e $[y]$ são dois intervalos é possível descrever a relação através de:

$$[x] \Diamond [y] = [\{x \Diamond y \mid x \in [x], y \in [y]\}]$$

Equação 4. Definições de operações em intervalos.

Dessa maneira as operações básicas intervalares podem também ser definidas como:

$$[x^-, x^+] + [y^-, y^+] = [x^- + y^-, x^+ + y^+];$$

$$[x^-, x^+] - [y^-, y^+] = [x^- - y^+, x^+ - y^-];$$

$$[x^-, x^+] * [y^-, y^+] = \min[(x^-, y^-, x+, y^-, x^-, y^+, x^+, y^+),$$
$$\max(x^-, y^-, x+, y^-, x^-, y^+, x^+, y^+);$$

$$[x^-, x^+] / [y^-, y^+] = 1)\ a = inv(y),$$
$$2)\ [x^-, x^+] * [a^-, a^+];$$

$$inv([x^-, x^+]) = se\ \ (x^- = \emptyset\ \ e\ \ x^+ = \emptyset\ )\ \ então\ \ [\emptyset, \emptyset]$$
$$senão\ \ se\ \ \ (x \in [0, x])\ então\ \ \text{[-infinito,+infinito]}$$
$$senão\ \ \ \ [\frac{1}{x^+}, \frac{1}{x^-}];$$

$$\min([x^-, x^+], [y^-, y^+]) = [\min(x^-, y^-), \min(x^+, y^+)];$$

$$\max([x^-, x^+], [y^-, y^+]) = [\max(x^-, y^-), \max(x^+, y^+)].$$

Equação 5. Operações básicas elementares.

Todas as definições apresentadas são considerando apenas uma dimensão. Quando há necessidade de trabalhar com um maior número de dimensões é necessário definir uma caixa, ou vetor de intervalos, que é um produto cartesiano de *n* intervalos, onde, *n* é a quantidade de dimensões.

$$[x] = [x_1^-, x_1^+] * \ldots * [x_n^-, x_n^+] = [x_1 * \ldots * x_n]$$

Equação 6. Operações básicas elementares.

O conjunto de todas as caixas de $\mathbb{R}^n$ é definido como $I\mathbb{R}^n$.

A principal operação realizada em caixas é a operação de bisseção. A bisseção é a divisão por um plano na dimensão de maior tamanho da caixa [*x*]. Exemplo, considerando a caixa [*x*] = {**[1,5]**,[1,2]}, após a bisseção serão gerada as caixas [*x*](1) = {**[1,3]**,[1,2]} e [*x*](2) = {**[3,5]**,[1,2]}. A primeira dimensão por ser maior foi dividida, conforme destacado.

3.2 SIVIA Sequencial

O algoritmo SIVIA é um algoritmo de busca determinístico proposto por Joulin e Walter (1993), como hipótese para diminuir a taxa de erro em estimativas de fronteiras não lineares através da análise de intervalos.

O artigo original do SIVIA não apresenta qualquer descrição de implementação computacional do SIVIA, contudo, Joulin em 2004 descreve uma implementação de uma biblioteca de matemática intervalar e uma implementação do algoritmo SIVIA sequencial, ambos em SCILAB no seu curso de matemática intervalar. Nesse documento do curso (JOULIN,2004) são baseadas as implementações dos algoritmos sequenciais e do PSIVIA-HP descrito na seção 3.2..

O algoritmo SIVIA sequencial se inicia com um intervalo de entrada que indica qual será o espaço de busca. Essa busca é realizada até que todos os intervalos gerados pelo algoritmo tenham sido avaliados e, por consequência, tenha ocorrido à delimitação entre as regiões válida e inválida.

Para as descrições seguintes é apresentada a figura 7 que possui o pseudocódigo do algoritmo SIVIA sequencial.

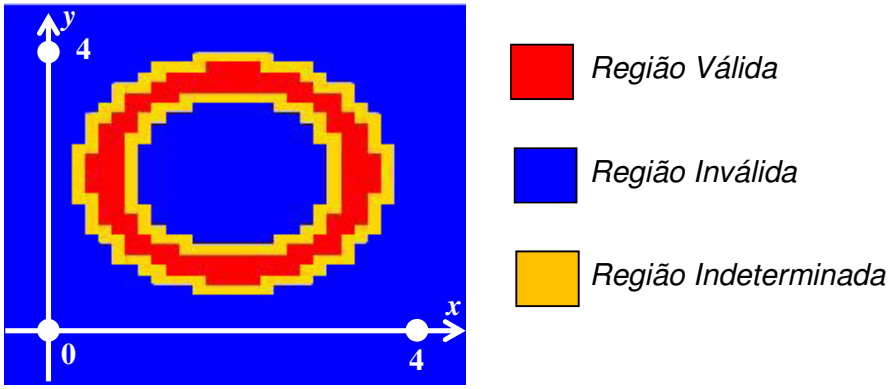
Figura 8. Pseudocódigo do algoritmo SIVIA sequencial.

```
Enfileirar (intervalo_inicial)
Enquanto (lista_intervalos != vazia) faça
   Desenfileirar(intervalo)
   Avaliar(intervalo)
     Se(avaliação = válido) então
       Enfileirar_lista_válidos(intervalo)
     Senão se (avaliação = inválido) então
       descartar(intervalo)
     Senão se (tamanho (intervalo)  < exatidão)
       descartar(intervalo)
     Senão se (avaliação = indeterminado ) então
      início
        Bissecionar(intervalo)
        Enfileirar(intervalo_gerado1)
        Enfileirar(intervalo_gerado2)
      Fim
```

Como é possível perceber na figura 7, a cada iteração um intervalo é retirado da lista de intervalos para ser avaliado pela função de pertinência, que é responsável por classificar o intervalo entre válido, inválido ou indeterminado. Os resultados dessa classificação podem ser de incluir o intervalo na lista de intervalos válidos, descartar o intervalo ou aplicar a operação de bisseção que gera novos intervalos menores. A figura 8 apresenta um exemplo de execução do algoritmo descrito na figura 7 para uma função de pertinência polinomial $z = (x-2)^2 + y^2$

, com intervalo inicial em x = [0,4] e y = [0,4]. Sendo que na figura 8, é destacado em azul os intervalos não pertencentes à função, em vermelho os pertencentes à função e em amarelo intervalos com tamanho menor que a exatidão (indeterminada). É válido ressaltar que este será o padrão adotado por toda a dissertação nas apresentações das funções.

Figura 9. Arruela impressa em SCILAB com o SIVIA Sequencial.



O intervalo inicial sendo x = [0,4] e y = [0,4] fixa o espaço de busca da execução do algoritmo. Logo, um intervalo de busca inicial que não seja amplo o suficiente, impedirá que se encontrem todos os intervalos válidos. Considerando que o intervalo inicial para a mesma função polinomial apresentada na figura 8 fosse em x = [2,4] e y = [0,4], apenas metade da arruela seria encontrada na execução do algoritmo.

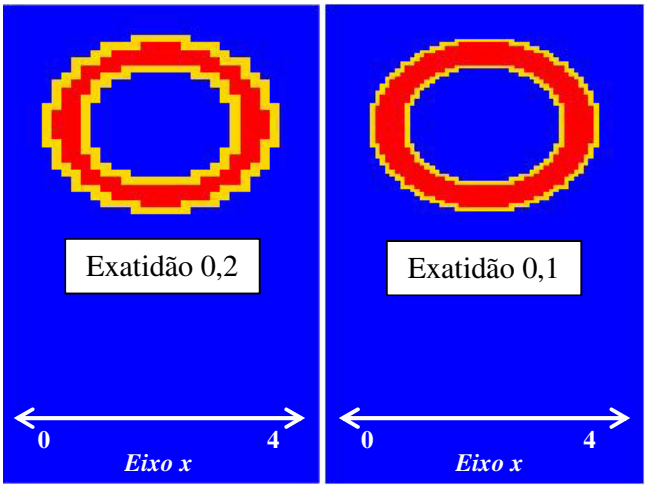
Um intervalo considerado válido na classificação indica que o seu intervalo é um subconjunto de toda a extensão válida da função de pertinência (em vermelho na figura 8), sendo assim ele é enfileirado na lista de intervalos válidos. Já um intervalo considerado inválido (em azul na figura 8), não pertence a nenhuma parte da função de pertinência, sendo descartado.

Um intervalo classificado como indeterminado indica que em sua extensão há subintervalos pertencentes à função de pertinência e subintervalos não pertencentes. Um exemplo, para melhor compreensão seria considerar o intervalo x = [3,25; 3,75] e y = [2,35; 2,4] para uma função qualquer. É possível que o subintervalo x = [3,25; 3,5] seja parte dessa função e o subintervalo x = [3,5; 3,75] não seja parte. A primeira ação realizada em intervalos indeterminados é avaliar se o seu tamanho é menor do que o parâmetro exatidão, caso seja maior o intervalo continua na execução do algoritmo, caso contrário é descartado. Esta ação é parte do critério de parada do algoritmo, pois a bisseção divide intervalos pela metade e caso não houvesse essa avaliação o algoritmo iria sempre estar gerando novos intervalos menores

e nunca finalizaria, pois estaria sempre avaliando. Sendo assim, o parâmetro exatidão pode ser definido como o tamanho mínimo para um intervalo ser parte da execução do algoritmo. Para realizar esta avaliação é necessário determinar o tamanho do intervalo, através da subtração do limite superior pelo limite inferior, da dimensão de maior tamanho. Um exemplo, para melhor compreensão seria determinar o tamanho do intervalo x = [-4,4] e y = [0,4], que baseado na regra mencionada possui tamanho 8.

Os intervalos de menor tamanho que a exatidão se localiza na fronteira entre o conjunto de intervalos válidos e o conjunto de intervalos inválidos. Estes intervalos estão em amarelo na figura 8 e 9.

Figura 10. Exatidões distintas para a função Polinomial para o intervalo [0,4]



A figura 9 apresenta um exemplo de duas execuções com valores distintos de exatidão. Como é possível perceber, a borda para a exatidão 0,2 possui maior área do que com a exatidão 0,1, já que a exatidão 0,1 possibilita que intervalos menores sejam avaliados e classificados como válidos ou inválidos. A especificação do valor da exatidão é dependente do problema e é determinada pelo especialista que irá utilizar o algoritmo, como Mazeika, Joulin e Osswald (2007) que utilizaram a exatidão de 0,05 no seu trabalho.

Para os intervalos classificados como indeterminados e com tamanho superior a exatidão, é aplicado o método de bisseção. Esse método consiste na divisão pela metade da dimensão de maior tamanho do intervalo. Exemplificando, considerando o intervalo x = [1; 10] e y = [4; 5], o tamanho das dimensões são respectivamente 9 para x e 1 para y; logo a divisão por 2 será realizada na dimensão x, gerando os intervalos: (1) x = [1; 5,5] e y = [4; 5]; (2) x = [5,5;10] e y = [4; 5]. É possível perceber que o eixo y não foi modificado na geração

dos dois novos intervalos, já que o método é aplicado apenas na dimensão de maior tamanho. Esses dois novos intervalos menores gerados são enfileirados para serem avaliados em futuras iterações, conforme apresentado na figura 7 da descrição do algoritmo.

A quantidade de bisseções é um indicador do custo de processamento do algoritmo, uma vez que, quanto maior o número de bisseções maior será a quantidade de intervalos gerados que devem ser classificados posteriormente. Como o SIVIA é um algoritmo determinístico o caminho que o algoritmo percorre depende da sua entrada e essa entrada (o intervalo inicial) determina a quantidade de bisseções na execução. Para melhor compreensão, é apresentado um cenário simulado com uma execução de 7 bisseções e os seguintes intervalos de 1 dimensão: [-4,4], [0,4] e [-4,8].

Tabela 1. Simulação de Bisseções.

Intervalo Inicial	Bisseções						
	1	*2*	*3*	*4*	***5***	*6*	*7*
[-4,4]	[0,4]	[0,2]	[1,2]	[1.5,2]	**[1.750,2]**	[1.875,2]	[1.9375,2]
[0,4]	[0,2]	[1,2]	[1.5,2]	[1.750,2]	**[1.875,2]**	[1,9375,2]	[1.96875,2]
[-4,8]	[-4,2]	[-1,2]	[0.5,2]	[1.25,2]	**[1.625,2]**	[1.8125, 2]	[1.90625, 2]

Com a sequência de bisseções apresentadas na tabela 1 os resultados das bisseções para os intervalos iniciais [-4,4] e [0,4] diferem em apenas uma iteração. Uma vez que o resultado da primeira bisseção do intervalo inicial [-4,4] é o intervalo [0,4] e nas bisseções seguintes com a diferença de 1 iteração produzem os mesmos intervalos. Já as bisseções do intervalo [-4,8] produzem intervalos diferentes, pois o tamanho do intervalo [-4,8] é diferente e não é múltiplo em relação ao tamanho dos intervalos [-4,4] e [0,4]. Com esse exemplo é possível apresentar a diferença do número de bisseções em relação ao intervalo de entrada. Pois, tomando o momento 5 da tabela 1, considerando que um intervalo válido fosse [1.625,2], na quinta bisseção este intervalo seria encontrado com o intervalo de entrada [-4,8]. Porém, não é o que ocorre para os intervalos [-4,4] e [0,4], pois analisando o momento 4 destes dois intervalos, ambos os intervalos são maiores do que o intervalo [1.625,2], sendo necessário realizar mais uma bisseção. Já no momento 5 eles são menores, logo será necessário realizar bisseções suficientes que gerem intervalos que representem o intervalo [1.625,2]. É válido ressaltar que o exemplo apresentando é para expressar a diferença do número de bisseções em relação ao intervalo de entrada e não que intervalos maiores alcançam melhores resultados. Visto que

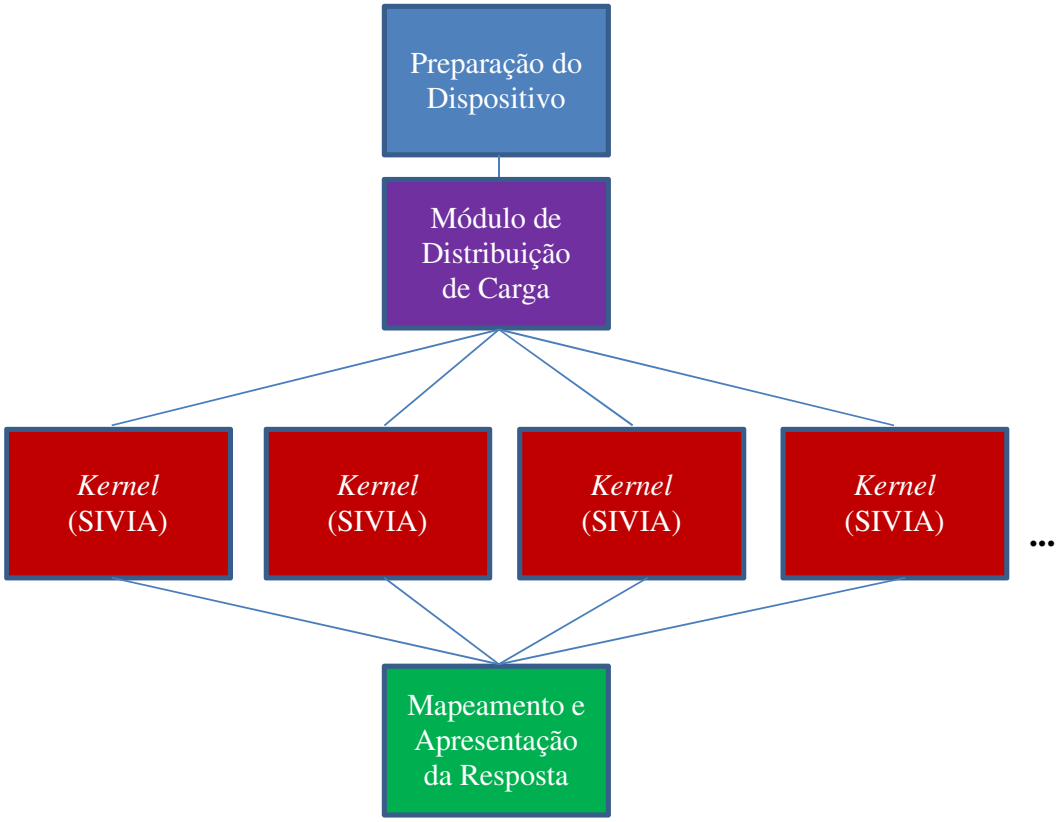
uma avaliação com este objetivo teria que ser realizada considerando também a função de pertinência.

Como mencionado, a cada bisseção dois novos intervalos são gerados e adicionados à lista de intervalos a serem avaliados. Essa lista de intervalos a serem avaliados recebe o intervalo inicial no início do algoritmo e apenas cresce com a inserção de intervalos gerados pelas bisseções. Sendo assim, o critério de parada do algoritmo é quando a lista de intervalos a serem avaliados estiver vazia.

3.2 PSIVIA-HP

A versão paralela desenvolvida do SIVIA nesta dissertação foi projetada com dois módulos principais necessários para suprir as demandas de distribuição de cargas (os intervalos) e execução paralela do algoritmo. A figura 10 apresenta um diagrama de composição do PSIVIA-HP.

Figura 11. Diagrama dos módulos do PSVIA-HP



O módulo de preparação do dispositivo é comum a todas as aplicações em OpenCL, como já mencionado na seção 2.3. Esse módulo possui as funções necessárias de preparação da plataforma para a execução do *Kernel*. Porém, foi adicionado a este módulo uma parametrização, permitindo ao usuário informar os parâmetros relacionados na tabela 2 necessários para à execução do PSIVIA-HP.

Tabela 2. Parâmetros para execução do PSIVIA-HP.

Parâmetros para Execução do PSIVIA-HP	
Parâmetro	Descrição
GW	Quantidade de WIs que serão executados.
LW	Tamanho do Grupo de WIs.
Quantidade de Instâncias do Problema.	O PSIVIA-HP permite que mais de um problema seja solucionado no mesmo instante.
Quantidade de Dispositivos	Quantos dispositivos que serão utilizados na execução. Ex.: ((1) – GPU ou CPU | (2) CPU + GPU)
Exatidão	Parâmetro utilizado tanto no *Host* quanto no *Kernel* que determina o tamanho mínimo que uma caixa deve ter para fazer parte da execução do algoritmo.
Seleção da função	É possível descrever em um arquivo várias funções de pertinência, este parâmetro seleciona dentro do arquivo qual função será utilizada.
Modo de saída dos resultados	Vídeo, arquivo para ser importado para o *excel* ou arquivo para ser utilizado no SCILAB.
Modo do *Kernel*	Compilar o *Kernel* ou utilizar um binário de um *Kernel* já compilado.

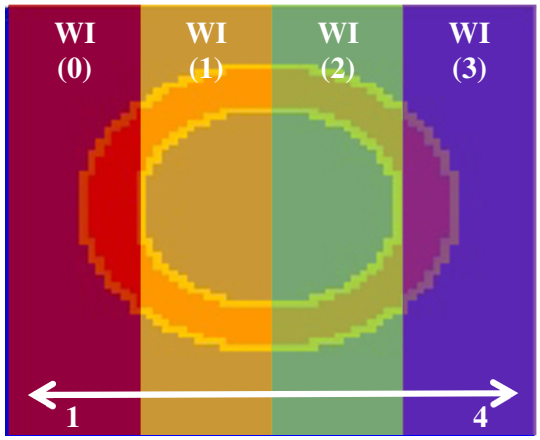
A hipótese da divisão da carga de trabalho implementada neste módulo, foi a divisão do eixo *x* pela quantidade de WI que será utilizado na execução. A hipótese desenvolvida se deve ao fato de que o SIVIA se inicia com apenas um intervalo de entrada e apenas após certo número de iterações (dependente da função de pertinência utilizada), é que irá ocorrer um número de bisseções suficientes para atribuir um intervalo por EP. Por essa razão, atribuindo-se uma porção do intervalo inicial de entrada, não será necessário esperar certa quantidade de iterações para haver carga de trabalho para todos os EPs. É válido ressaltar que a quantidade

de iterações não é alta quando executado em CPU, pois as CPUs disponíveis no mercado, até a escrita desta dissertação, chegam a oito núcleos de processamento. Entretanto, as GPUs disponíveis no mercado são de centenas e/ou milhares de EPs

A implementação do módulo de distribuição de carga foi realizada no *Host*, uma vez que os WIs é que necessitam das porções dos intervalos para iniciar a execução. A execução deste módulo, defini e atribui em um *buffer* as porções dos intervalos de cada WI e copia para o *buffer* da plataforma que irá executar a aplicação.

Para melhor compreensão, a figura 11 é apresentada para demonstrar a divisão da carga de trabalho quando há apenas uma instância do problema a ser solucionado. Porém, é válido ressaltar que o PSIVIA-HP possibilita a execução de mais de uma instância do problema como descrito na tabela 2.

Figura 12. Divisão do intervalo inicial [0,4] para uma execução com 4 WIs.



No exemplo da figura 11, a função polinomial utilizada é a $z = (x - 2)^2 + (y - 2)^2$ para um intervalo inicial de entrada sendo x = [1,4] e y = [0,4] em uma execução com 4 WIs. Com esta configuração o módulo de distribuição de carga iria atribuir para o *buffer* os intervalos apresentados na tabela 3.

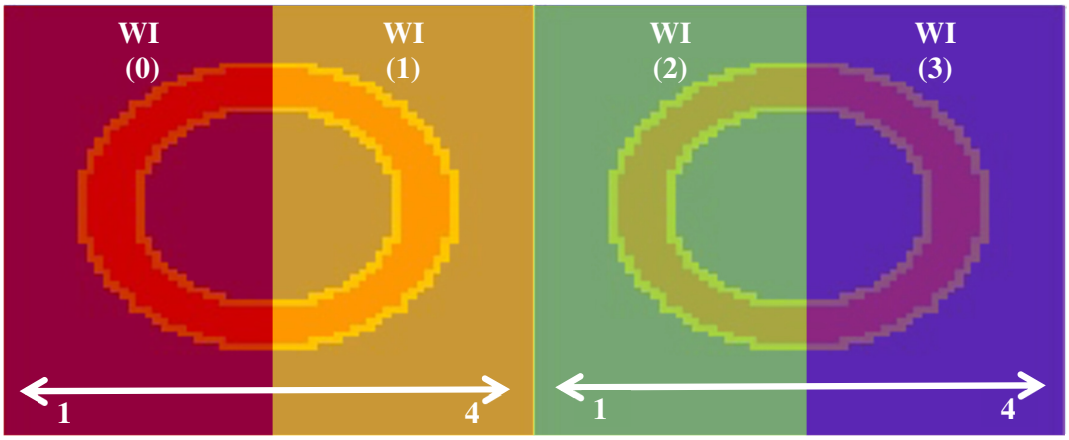
Tabela 3. Divisão do intervalo inicial [0,4] para uma execução com 4 WIs.

WI (0)	WI (1)	WI (2)	WI (3)
x = [1,1.75]; y = [0,4]	*x = [1.75,2.5]*; y = [0,4]	*x = [2.5,3.25]*; y = [0,4]	*x = [3.25,4]*; y = [0,4]

Como a hipótese de divisão da carga de trabalho é a divisão do eixo *x,* as bisseções iniciais serão aplicadas no eixo *y* para criação dos novos intervalos, já que as bisseções são realizadas nas dimensões de maior tamanho.

O PSIVIA-HP quando configurado para solucionar mais de uma instância do problema, o módulo de distribuição de carga divide o espaço de busca da primeira instância entre os primeiros WIs e assim sucessivamente. A divisão é realizada somando as extensões dos espaços de busca de todas as instâncias do problema e dividindo entre os WIs. Um exemplo apresentado na figura 12, considera duas instâncias da função polinomial apresentada na figura 11 para ser dividida entre 4 WIs, para um intervalo inicial em *x* sendo [1,4].

Figura 13. Divisão do intervalo inicial [0,4] para uma execução com 4 WIs.



Para o exemplo da figura 12, o módulo de divisão de carga somou o espaço de busca das duas instâncias, (tamanho([1,4]) => 3) (+) (tamanho([1,4]) => 3) = 6. Essa soma é então dividida pela quantidade de WIs. Como são 4 WIs, cada WI irá trabalhar com uma porção de 1,5 do eixo *x*. Sendo que, os primeiros WIs (0 e 1) irão trabalhar com a porção de 1,5 apenas da primeira instância e os WIs restantes (2 e 3) com a porção de 1,5 da segunda instância. O cálculo da porção de cada WI é apresentado na equação 1.

$$(1) \quad p = \frac{\sum_{i=0}^{n} w([x_i])}{n}$$

Onde,
p *é a porção calculada;*
x *a instância;*
n *total de instâncias;*
w(x) *tamanho da instância.*

Caso haja mais instâncias do problema do que EPs disponíveis o módulo de divisão de carga atribuí uma instância inteira para cada WI e a instância restante é dividia entre todos os WIs. A tabela 4 apresenta um exemplo desse cenário contendo 3 instâncias, 2 WIs, um intervalo inicial de *x* em [0,4] e a mesma função polinomial trabalhada na figura 12.

Tabela 4. Exemplo de intervalos atribuídos para 2 WIs e 3 instâncias do problema.

	Instância 1	*Instância 2*	*Instância 3*
WI(0)	[0,4]	-	[0,2]
WI(1)	-	[0,4]	[2,4]

O módulo de distribuição de carga não foi paralelizado, pois em resultados iniciais o tempo de execução médio foi de 15 micros segundos. Como há custo computacional no emprego de técnicas paralelas, este tempo médio é baixo para aplicação de tais técnicas.

O *Kernel* do PSIVIA-HP é a implementação descrita em (JOULIN, 2005) descrita na seção 3.1 com adequação ao padrão de programação do OpenCL. Porém, é necessário ressaltar as seguintes características importantes da implementação desenvolvida:

1. Apesar dos exemplos e resultados desta dissertação serem apresentados para problemas em duas dimensões, o PSIVIA-HP permite trabalhar com problemas N-dimensionais.
2. Foi implementado para permitir a execução em dispositivos distintos no mesmo instante, sendo possível compartilhar o mesmo contexto caso esses dispositivos sejam do mesmo fabricante, como descrito em (AMD, 2010);
3. Toda a biblioteca intervalar com os principais operadores e as principais operações, tanto para intervalar quanto para caixas (conjunto de intervalos), foram preparados visando a execução em OpenCL para o PSIVIA-HP;
4. Permite a execução de funções multiobjetivos.

Nenhuma parte do algoritmo SIVIA sequencial apresentado na figura 7 foi paralelizada. Analisando o código do SIVIA sequencial, o passo com maior custo computacional é a classificação dos intervalos pela função de pertinência. O custo dessa classificação é em relação às operações que compõem a função de pertinência e a quantidade de dimensões. Paralelizar apenas o passo da avaliação não se apresentou como uma abordagem efetiva, já que todos os testes foram realizados com cenários 2 dimensões e a paralelização da classificação poderia apenas ter efeito caso a quantidade de dimensões fosse igual ao número de EPs, com funções complexas contendo desvios. Sendo assim, a abordagem adotada foi em relação ao espaço de busca para permitir que desde a primeira iteração haja intervalos sendo avaliados, sem a necessidade de esperar um número de iterações que gerem intervalos igual ao número de EPs.

A lista de intervalos a serem avaliados utilizada pelo SIVIA sequencial descrita em (JOULIN, 2004) é implementada através de uma lista encadeada. Porém, o OpenCL não possibilita trabalhar com alocação dinâmica dentro do código do *Kernel*, sendo necessária a alocação de um *buffer* antes da execução. Como já mencionado, alocações fora do *Kernel* estão obrigatoriamente na área de memória *Global* do OpenCL e são fisicamente armazenadas na memória *global* da GPU que é uma memória maior e mais lenta. Porém, o PSVIA-HP foi implementado com a possibilidade de trabalhar com um *buffer* declarado na área de memória *Private*, que pode ser utilizado quando o problema não exige uma grande quantidade de intervalos e é possível alocar esses intervalos na área de memória rápida e limitada.

Em resultados iniciais, o PSIVIA-HP apresentou um comportamento que foi denominado como redundância. A redundância pode ser definida como um conjunto de intervalos que poderiam ser encontrados em apenas um intervalo, caso fosse utilizado um menor número de WIs, contudo levaria a um nível de paralelismo menor. Conjecturando uma execução para auxiliar a compreensão, considerando uma função de pertinência qualquer com 2 WIs, o WI (0) poderia encontrar o intervalo em x = [1,3] e o WI (1) poderia encontrar o intervalo em x = [3,5], porém em uma execução sequencial o intervalo x = [1,5] poderia ser encontrado. Sendo que este intervalo x = [1,5] é a união dos dois intervalos encontrados na execução paralela e como resultado final é a mesma representação. Entretanto, vale ressaltar que isto não se caracteriza como um problema, pois a redundância é dependente da forma da função de pertinência e da quantidade de WIs que estão em execução. Sendo assim, não há garantia que irá ocorrer à redundância e, caso ocorra, qual seria a quantidade.

O módulo de Mapeamento e Apresentação da Resposta realiza a leitura dos *buffers* de resposta das plataformas de execução e exibe o resultado. Esse módulo possui as opções de exibir os resultados em vídeo ou em arquivo. Caso o usuário opte por exportar em arquivo há duas opções: criar o arquivo em um formato que seja possível importar dados externos para o *excel* ou no formato de intervalos para serem manipulados no SCILAB.

4. METODOLOGIA

Esta seção tem como objetivo apresentar as etapas da pesquisa, as configurações dos experimentos e os resultados iniciais que foram necessários para validar os algoritmos.

4.1 Etapas da Pesquisa

Para o desenvolvimento da pesquisa foi necessário cumprir as seguintes etapas:

- Construção da biblioteca intervalar no padrão C99 contendo os operadores intervalares fundamentais;
- Construção do algoritmo SIVIA sequencial em C baseado no código descrito em JOULIN (2004) em SCILAB com lista encadeada;
- Construção do algoritmo SIVIA sequencial com um vetor alocado estaticamente, porém logicamente trabalhando como dinâmico, já que o OpenCL não permite alocação dinâmica dentro do *Kernel*;
- Implementação do PSIVIA-HP descrito na seção 3.2;
- Validação das versões através da comparação dos intervalos encontrados. A comparação dos algoritmos sequenciais e do PSIVIA-HP foi realizada sobre o a versão sequencial implementada em SCILAB considerando uma execução com exatidão de 0,2 e a função polinomial descrita em JOULIN (2004);
- Definição dos parâmetros dos experimentos (funções de pertinência e valores de exatidão) através de testes primários;
- Geração e análise dos resultados;
- Escrita da dissertação.

.

4.2 Configuração dos Experimentos

Esta seção é destinada a apresentar todo o ambiente no qual os resultados foram gerados, sendo desde os recursos computacionais, até os parâmetros fundamentais para execução do SIVIA sequencial e do PSIVIA-HP.

Para a geração dos resultados inicialmente foi definido o ambiente físico a partir da disponibilidade de recursos do LSDC (Laboratório de Sistemas Digitais Computacionais) do PPGEE (Programa de Pós-Graduação de Engenharia Elétrica), que é apresentado na tabela 5.

Tabela 5. Ambiente da geração dos Resultados.

Software	
Sistema Operacional:	Windows 7 64 bits
IDE:	VISUAL STUDIO 2010
Hardware	
Processador:	Intel(R) Core(TM) i7-3820QM CPU @ 2.70GHz com 4 núcleos físicos simulando 2 núcleos lógicos cada.
Cache L1:	64 KB por núcleo físico, sendo 32 KB para a cache de instruções e 32 KB para cache de dados.
Cache L2:	64 KB por núcleo físico.
Cache L3:	Cache L3 de 8192 KB
Memória Principal:	16 GB
GPU:	NVIDIA GTX 670M com 7 *Stream Multprocessor*, sendo que cada *Stream Multprocessor* contém 48 *Stream Processor*, totalizando 336 elementos de processamento.

A definição do intervalo de entrada inicial, o valor da exatidão e a função de pertinência são obrigatórios por serem parâmetros originários do SIVIA, porém é necessário definir outros parâmetros além dos mencionados para a execução do PSIVIA-HP. A tabela 6 apresenta a escolha dos parâmetros para a coleta dos resultados para cada versão dos algoritmos.

Tabela 6. Variação dos parâmetros utilizados nos experimentos.

Parâmetro	Variação	SIVIA Sequencial	PSIVIA-HP
Intervalos	[0,4][-4,4], [-4,4][-4,4] e [-4,8][-4,4].	**X**	**X**
GW em GPU	1344.		**X**
GW em CPU	1, 2, 4 e 8.		**X**
LW em GPU	48 (28 Grupos), 96 (14 Grupos), e 192 (7 Grupos).		**X**
LW em CPU	1.		**X**
Funções	Polinomial: $z = (x-2)^2 + (y-2)^2$; Exponencial: $z = e^{x-2} + e^{y-2}$; Logarítmica: $z = \log(x) + \log(y)$.	**X**	**X**
Exatidão	0,00001	**X**	**X**
Algoritmos			
Versões e dispositivos de execução do Algoritmo	Lista Encadeada (CPU); Alocação de vetor (CPU); PSIVIA-HP(CPU e GPU).		

Os itens não marcados para o SIVIA sequencial na tabela 6 é por serem itens exclusivos de aplicações em OpenCL. O valor de GW ser 1344 (336 EPs x 4) é em razão da especificação da placa utilizada (336 EPs) e por uma regra mencionada em AMD (2010) que aconselha a atribuir quatro vezes mais WIs do que EPs disponíveis por haver um *pipeline* em cada SP. Com o valor do GW escolhido é necessário definir o tamanho dos grupos que serão atribuídos para os SMPs, que por consequência irá determinar a quantidade de grupos criados. O LW definido como 48 implicará na criação de 28 grupos com 48 WI (valor do LW) cada. Como a GPU utilizada possui 7 SMPs e cada grupo é atribuído para cada SMP, gera-se concorrência por haver mais grupos do que SMPs. É necessário ressaltar que apesar da concorrência pode ocorrer dos WIs em um grupo assumir intervalos iniciais que na primeira iteração já serão classificados como inválidos; logo este grupo sairá do SMP sendo atribuído outro grupo para o SMP. Esse valor foi escolhido para justamente verificar se este

escalonamento irá propiciar melhor desempenho. Já com o LW sendo 96 serão criados 14 grupos, gerando concorrência como mencionado. Contudo, com cada grupo possuindo o dobro de WIs do que EPs disponíveis, parte do *pipeline* do SP será utilizado. Com o LW 192 serão criados 7 grupos, não gerando concorrência entre SMPs e utilizando todo o recurso do *pipeline* de cada SP. Porém, caso 1 WI em um grupo não finalizar na primeira iteração, irá haver perda de recurso. Pois, como mencionado na seção 2.3, WIs que já finalizaram o seu processamento em grupos que possuam WIs com trabalho real, processam operações *nop* até que todos finalizem.

O valor máximo de GW da CPU foi escolhido pela quantidade de núcleos do processador utilizado e a variação de 1 até o máximo, em potência de 2, para avaliar a escalabilidade da aplicação. Já o valor do LW foi determinado como 1 pelas razões descritas na seção 2.3.

Para uma maior compreensão do ganho e do comportamento do PSIVIA-HP, os resultados foram analisados em três partes, a primeira avaliando o tempo de execução e a esacalabilidade apenas do *Kernel*, a segunda análise em relação à escalabilidade e o ganho de toda à aplicação. Por fim, a terceira análise foi em relação ao custo de preparação do dispositivo nas plataformas utilizadas.

O tempo total da aplicação é composto pelas métricas apresentadas na equação 2.

$$T_{TOTAL} = T_K + T_D + T_{LB}$$

Equação 2. Cálculo de Tempo da Aplicação.

As métricas apresentadas na equação 2 são: T_K o tempo do *Kernel*, T_D o tempo de preparação do dispositivo e T_{LB} o tempo de leitura do *Buffer*.

4.3 Verificação dos Parâmetros Experimentais

Esta seção tem como objetivos, apresentar e justificar as funções de pertinência e os valores dos parâmetros utilizados, e a escolha dos métodos para análise dos resultados. É parte também dos objetivos desta seção apresentar resultados iniciais que validam os algoritmos SIVIA sequenciais e o PSIVIA-HP.

Para utilizar o SIVIA é fundamental o conhecimento da função de pertinência que é o modelo matemático do problema que se deseja definir as fronteiras. Conforme já mencionado

no capítulo 3, a função de pertinência influência diretamente na quantidade de trabalho do SIVIA. Sendo assim, foram selecionadas 3 funções de complexidades matemáticas distintas que geram quantidades de trabalho diferentes. As funções escolhidas para os experimentos são apresentadas a seguir:

- Polinomial: $z = (x-2)^2 + (y-2)^2$;
- Exponencial: $z = e^{x-2} + e^{y-2}$;
- Logarítmica: $z = \log(x) + \log(y)$;

A função polinomial (sem deslocamento) é utilizada como exemplo no curso de SIVIA em SCILAB no documento de JOULIN (2004). Sendo assim, esta função foi utilizada como base para validar todas as aplicações desenvolvidas e por consequência na geração dos resultados. Contudo, foi preciso selecionar outras duas funções de pertinência que gerem uma carga de trabalho diferente da função polinomial. Sendo assim, foram selecionadas as funções exponencial e logarítmica apresentadas. Outro fator importante para a seleção destas duas funções é o fato de que GPUs foram desenvolvidas para processamento de aplicações gráficas e lidam com funções matemáticas mais complexas em razão do seu hardware otimizado.

Nos exemplos do documento de JOULIN (2004) foi utilizado o intervalo inicial x = [-4,4] e y = [-4,4], sendo então o primeiro intervalo utilizado na validação dos algoritmos e coleta dos resultados. Porém, como apresentado na tabela 7 o tamanho do intervalo inicial influi na quantidade de intervalos encontrados, na quantidade de iterações e por consequência a quantidade de bisseções na execução do algoritmo. Assim, foram selecionados os intervalos [0,4] [-4,4] e [-4,8] [-4,4], que são as variações de 50% para mais e para menos do intervalo [-4,4][-4,4] considerando apenas o eixo x. Não foi realizada variação no eixo y, permanecendo constante [-4,4] para todos os intervalos, pois o método de paralelização do PSIVIA-HP implementado consiste na divisão do eixo x entre os EPs.

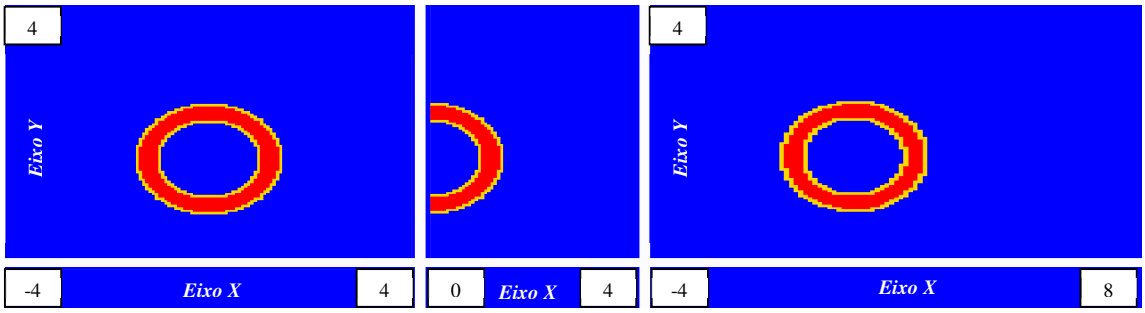
Nos resultados variando o intervalo de entrada inicial, foi possível constatar a influência deste intervalo no caminho de execução do algoritmo, como apresentado na tabela 7.

Tabela 7. Resultados Iniciais do algoritmo SIVIA e do PSIVIA-HP com variação dos Intervalos para Exatidão de 0.1 sem deslocamento da função.

Quantidade de Intervalos Encontrados Polinomial			
Intervalos	SIVIA Scilab	SIVIA Sequencial – Lista	PSIVIA-HP em CPU
X = [-4,4], Y= [-4,4]	140	140	140
X = [0,4], Y= [-4,4]	70	70	70
X = [-4,8], Y= [-4,4]	108	108	108

A tabela 7 evidencia dois aspectos que podem ser melhor descritos com a ajuda da figura 13, que foi construída com a função polinomial utilizada em Joulin (2004), similar a função da tabela 6, pois não possui deslocamento.

Figura 14. Variação dos intervalos no eixo *X* com a função polinomial e exatidão 0.1.

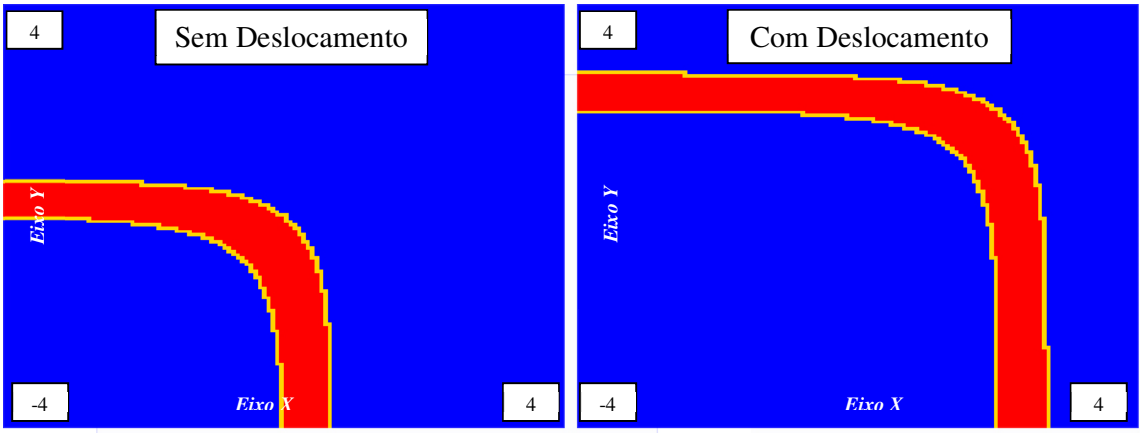


É possível perceber na figura 13 que o intervalo de entrada sendo x = [0,4] e y = [-4,4] foi encontrado apenas metade da arruela. Sendo assim, ao executar o algoritmo SIVIA deve-se trabalhar com um intervalo inicial que seja suficiente para encontrar todos os intervalos da função utilizada. Caso não se conheça a forma da função, principalmente se a execução considerar funções multiobjetivos, mais de uma execução será necessária. Mas, a análise mais importante é em relação aos outros dois intervalos de entrada, pois ambos possuem tamanho suficiente para encontrar todos os intervalos da arruela. Sendo que, o intervalo x = [-4, 8] encontrou uma quantidade menor de intervalos do que o intervalo x = [-4,4] o que não caracteriza erro. Uma vez que, como a bisseção divide pela metade o eixo de maior tamanho do intervalo, alguns intervalos podem ter sido encontrados sem redundância na execução do intervalo x = [-4,8].

Por ter sido encontrado apenas metade da arruela na execução do intervalo x = [0,4], conforme apresentado na figura 13, a função polinomial foi deslocada em -2 no eixo x e -2 no eixo y para geração dos resultados. Assim, é possível encontrar toda a arruela com os três intervalos de entrada selecionados, já apresentados na tabela 6.

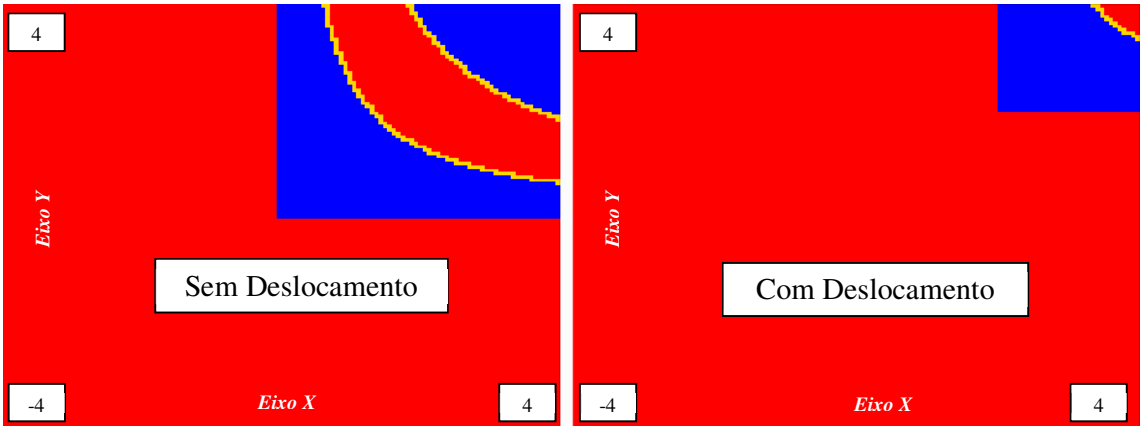
À função exponencial foi também deslocada em -2 no eixo x e -2 no eixo y, como estratégia para aumentar a carga de trabalho para os intervalos iniciais de entrada selecionados. Pois, conforme apresenta a figura 14, pelo formato da função e os intervalos de entrada selecionados, são encontrados mais intervalos finais com o deslocamento.

Figura 15. Variação dos intervalos no eixo *X* com a função exponencial e exatidão 0.1.



Na função logarítmica não foi realizado deslocamento em nenhum dos eixos, já que o deslocamento gera menos intervalos a serem encontrados, como é possível perceber na figura 15.

Figura 16. Variação dos intervalos no eixo *X* com a função logarítmica e exatidão 0.1.



Para o desenvolvimento da pesquisa foi necessário à construção do SIVIA sequencial em C. A primeira versão foi à transcrição do código em SCILAB do SIVIA descrito em JOULIN (2005) que utiliza uma estrutura de dados em lista. Logo, a transcrição foi realizada utilizando alocação dinâmica a cada inserção. Um fator relevante sobre essa versão é que a cada inserção ou remoção na lista, um acesso à memória principal obrigatoriamente é gerado. Mesmo que a cache contenha a política de escrita *write-back*, a cache apenas irá auxiliar na pesquisa, uma vez que a cada inserção um comando de alocação de memória é gerado e não um comando de alteração de bloco em cache. Vale ressaltar que a implementação utilizando lista encadeada é ensinada nas universidades e é o modelo mais utilizado pela facilidade de manipulação, mesmo não levando a um melhor desempenho.

O OpenCL não permite alocação dinâmica no interior do *Kernel,* sendo necessário alocar todo *buffer* no *Host* e copiar para o dispositivo anteriormente a execução. Por esta razão, foi desenvolvida também uma versão sequencial em C com esse princípio, utilizando um vetor alocado dinamicamente anterior à execução do código do SIVIA presente no código da aplicação. Assim, toda inserção de intervalos no vetor será apenas uma cópia entre espaços de memória. E caso o bloco esteja na cache, será apenas necessário acessar o endereço de memória da posição do vetor que já foi alocada e alterá-la.

Em resultados iniciais, apresentados na tabela 8, é possível perceber a diferença de desempenho entre as duas versões sequenciais. Os resultados apresentados são a média aritmética simples de 10 execuções, utilizando a função polinomial apresentada na tabela 6 sem deslocamento e o intervalo de entrada sendo x = [-4,4] e y = [-4,4].

Tabela 8. Tempos de execução (em segundos) das versões sequenciais implementadas para função polinomial e intervalo [-4,4].

Algoritmo	Tempo Total
Lista	103.291
Vetor	17.503

O ganho na utilização do vetor alocado é de 5,09 vezes em relação à lista encadeada, porém é válido ressaltar que desenvolver um código com lista encadeada é mais simples do que com vetor alocado, ficando a cargo de o desenvolvedor escolher entre facilidade de desenvolvimento e desempenho. É importante salientar que estes resultados são apresentados nesta seção apenas para validação dos algoritmos, pois estes resultados serão apresentados novamente em conjunto com os resultados do PSIVIA-HP no capítulo 5.

Na área de Computação de Alto Desempenho a métrica utilizada para medir desempenho é o tempo de execução (HENESSY e PATTERSON, 2003). Por esta razão,

foram medidos o tempo de execução de toda a aplicação, o tempo de execução do *Kernel* e o tempo de leitura de *buffer* de resposta. O tempo de alocação e transferência da carga de trabalho inicial não foi medido, pois em resultados iniciais a alocação e a transferência de dados em GPU foram em média 18 micros segundos, sendo irrelevante no tempo total da aplicação. Estes 18 micros segundos são da transferência entre a memória da CPU (*Host*) para a memória global da GPU através do barramento PCI-EXPRESS. Na CPU a transferência também se apresentou inexpressiva, uma vez que a transferência do *Host* para o dispositivo é apenas uma cópia de dados na própria memória principal.

Para a coleta do tempo, foram realizadas 10 execuções combinando os parâmetros apresentados na tabela 6 e calculada a média aritmética simples.

A proporção de ganho é realizada através do cálculo de *Speedup* descrito em (HENESSY E PATTERSON, 2003a) apresentado na equação 2:

$$Speedup = \frac{T_{SM}}{T_{CM}}$$

Equação 7. Cálculo de *Speed Up*.

Na equação 2 T_{SM} é o *tempo sem melhoria* e T_{CM} o *tempo com melhoria*. Para todos os cálculos foi utilizado no *Tempo Sem Melhoria* o tempo médio de execução da versão lista encadeada sequencial. Já para o *Tempo Com Melhoria* foi utilizado o tempo médio de execução das combinações apresentadas na tabela 6.

Conforme descrito na seção 3.1, a exatidão é o parâmetro que juntamente com a função de pertinência determina a quantidade de trabalho que irá ser exigida para a conclusão do algoritmo. Sendo que, fixando a função de pertinência e aumentando a exatidão aumente-se também a quantidade de intervalos gerados, conforme resultados iniciais coletados com a função polinomial e o intervalo [-4,4] contido em JOULIN (2004) apresentado na tabela 9:

Tabela 9. Quantidade de intervalos encontrados para a função polinomial e o intervalo [-4,4] de Joulin (2004).

Exatidão	Intervalos Encontrados
0,2	56
0,1	140

Para definir a exatidão a ser utilizada, foram realizados experimentos iniciais para identificar uma exatidão que demandasse processamento suficiente para haver ganho com a paralização. No primeiro experimento utilizou-se a exatidão de 0,2, intervalo em x = [-4,4] e y = [-4,4] e a função polinomial que é apresentada no documento de JOULIN (2004). Os resultados iniciais apresentaram tempo de execução médio de 2,65 micros segundos. Em

seguida, foi realizado um experimento com a exatidão de 0,1 com tempo de execução médio de 3,1 micros segundos. Diante destes resultados foi se variando a exatidão sempre na ordem de 10^{-1}. Os resultados destas variações e os tempos de execução são apresentados na tabela 10.

Tabela 10. Verificação de Exatidão em Segundos.

Verificação de Precisões						
Exatidão	**0,2**	**0,1**	**0,01**	**0,001**	**0,0001**	**0,00001**
Execução	**Tempo**					
1	0,000016	0,000031	0,000156	0,001139	11,872000	120,075000
2	0,000016	0,000031	0,000156	0,000998	16,957000	99,778000
3	0,000031	0,000031	0,000156	0,001014	12,854000	124,410000
4	0,000031	0,000031	0,000156	0,001092	16,333000	86,284000
5	0,000031	0,000031	0,000156	0,001123	15,272000	111,259000
6	0,000031	0,000031	0,000140	0,001108	17,035000	88,483000
7	0,000031	0,000031	0,000156	0,001123	16,442000	91,868000
8	0,000031	0,000031	0,000156	0,001061	16,318000	112,148000
9	0,000031	0,000031	0,000140	0,001061	12,199000	99,606000
10	0,000016	0,000031	0,000156	0,000998	14,118000	98,998000
Média	***0,000027***	***0,000031***	***0,000153***	***0,001072***	***14,940000***	***103,290900***
Desvio Padrão	***0,000007***	***0,000000***	***0,000006***	***0,000049***	***1,824472***	***11,881997***

Como é possível perceber, a exatidão de 0,00001 demandou maior tempo para ser processada com a média de tempo de 103,29 segundos. É possível perceber também que a partir da exatidão 0,01, variando a exatidão, o desvio padrão se modificou expressivamente, indicando maior influência das trocas de contexto de processos, acessos à memória principal, falhas na cache, entre outras razões de execuções em Sistemas Operacionais. Pelas razões acima apresentadas sobre o tempo médio de execução e o valor do desvio padrão, os resultados serão limitados apenas para o valor de exatidão 0,00001.

Diante do exposto desta seção, os algoritmos SIVIA sequenciais e o PSIVIA-HP foram verificados para a geração dos resultados. Vale ressaltar que apesar do PSIVIA-HP nesta seção ter sido executado apenas em CPU para validação, ele foi implementando em OpenCL que, como já descrito, foi concebido para trabalhar em plataformas heterogêneas sem a necessidade de recodificação.

No capítulo seguinte são apresentados os resultados dos experimentos e as respectivas análises.

5. RESULTADOS EXPERIMENTAIS

Este capítulo é destinado a apresentar os resultados e as análises de desempenho do *Kernel*, a análise do desempenho global do PSIVIA-HP, e uma análise sobre o tempo de preparação do dispositivo em relação ao tempo de *Kernel*.

5.1 Análise de Desempenho do *Kernel* do PSIVIA-HP

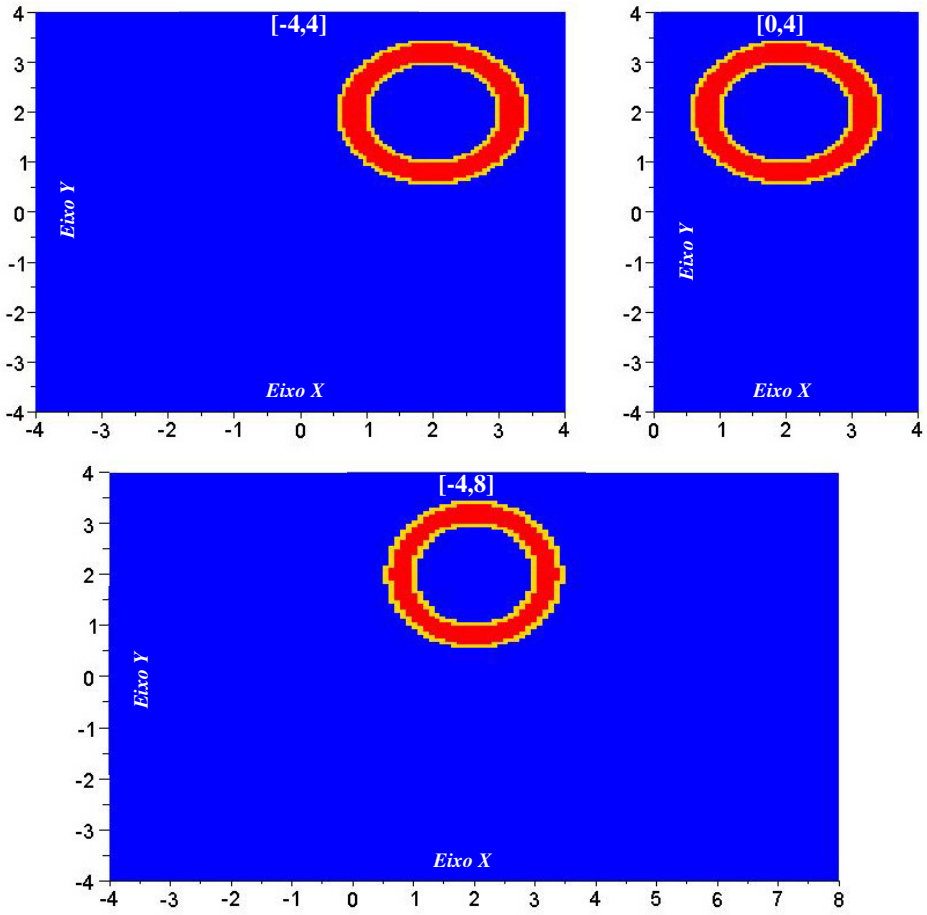
Nesta seção será analisada a escalabilidade e o ganho de desempenho na variação de WIs em CPU. Já em GPU será analisado o desempenho na variação do valor do LW. Os resultados e análises apresentadas nesta seção são apenas do tempo do *Kernel*.

5.1.1 Análise do Tempo do Kernel em CPU

Para analisar o desempenho do *Kernel* tanto em GPU quanto em CPU é necessário entender a posição da função no espaço de busca. Por esta razão, antes de cada análise será apresentada a posição da função no espaço de busca, para cada intervalo utilizado na geração dos resultados. Nos gráficos são apresentados o termo *thread* ao invés de WI, pois *thread* é o termo utilizado em Computação de Alto Desempenho.

Como a primeira análise é em relação à função polinomial, a figura 16 apresenta a posição da função polinomial no espaço de busca.

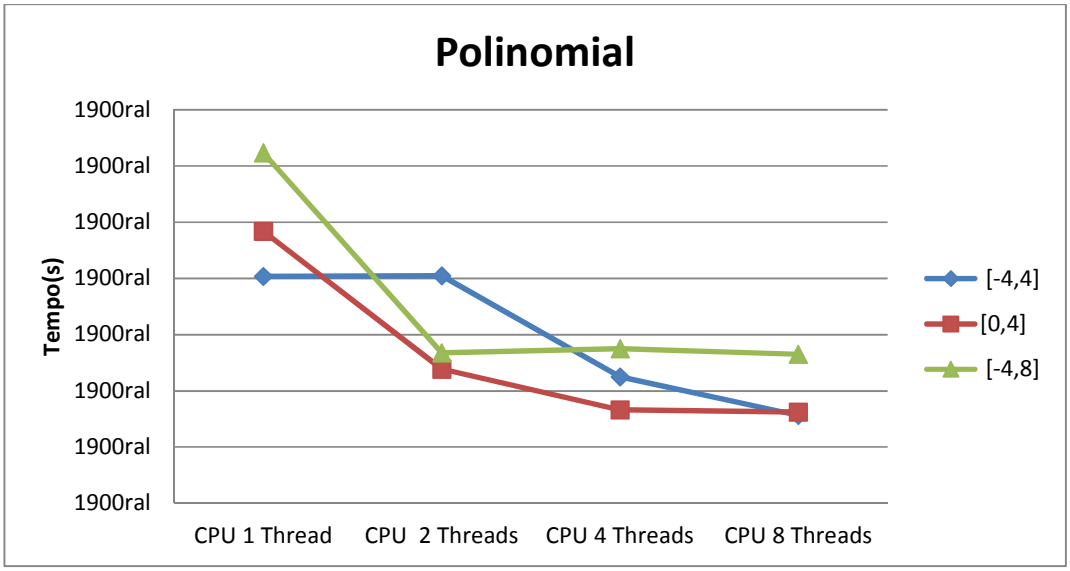
Figura 17. Espaço de busca para a função polinomial para todos os intervalos utilizados.



A posição da função no espaço de busca é importante, pois os WIs irão receber porções dos intervalos e alguns WIs podem receber porções que não pertencem à função.

A figura 17 apresenta os valores de tempo médio de execução do *Kernel* para a função polinomial.

Figura 18. Tempo médio de execução do *Kernel* em CPU com variação de WI para a função polinomial.



A primeira análise que pode ser realizada pela figura 17 é em relação à execução do intervalo [-4,4]. Ao variar de 1 para 2 WIs o desempenho ficou similar, pois como o PSIVIA-HP divide o eixo *x* entre os WIs, o primeiro WI ficou com o intervalo [-4,0] (não pertencente à função) e o segundo WI com o intervalo [0,4]. O WI com o intervalo [-4,0] finaliza na primeira iteração, ficando o WI com o intervalo [0,4] processando toda a função sequencialmente. Dessa maneira, é como se apenas um WI fosse lançado para a execução. Outro fator que é pertinente é o custo de criação de WIs; criar um WI (execução sequencial) possui um custo menor do que criar dois ou mais WIs.

No intervalo [-4,4] a aplicação é escalável apenas a partir de 2 WIs, pois só há nível de paralelismo a partir de 4 WIs, pois 2 WIs finalizam na primeira iteração e 2 WI processam o intervalo [0,4] que possui a função.

Exceto para execução com 2 WIs, onde foi superior apenas sobre a execução do intervalo [-4,4], o intervalo [-4,8] é que obteve desempenho inferior. Com 2 WIs, os tempos médios de execução de 0,477 segundos para o intervalo [-4,8] e 0,809 segundos para o intervalo [-4,4], ocorreu porque na divisão do intervalo [-4,8], o primeiro WI fica com o intervalo [-4,2] que contém a primeira metade da função e o segundo WI com o intervalo [2,8] que possui a segunda metade da função para processar. No intervalo [-4,4], como já mencionado, o primeiro WI não conterá nenhuma parte da função para processar, finalizando na primeira iteração.

Conforme já mencionado, intervalos que possuem grandes porções que não pertencem à função e que a função está localizada no centro do intervalo necessitam de um maior número

de bisseções para eliminar esses espações não pertencentes. Por esta razão, o intervalo [-4,8] obteve desempenho inferior na execução com 1 WI. Nas execuções paralelas, houve ganho em relação a execução com 1 WI, porém o fato da execução com 2 WIs ser superior à execução com 4 WIs é que na execução com 4 WIs, o primeiro e o último WI finalizam na primeira iteração, ficando apenas 2 WIs processando a função. Com o mesmo número de WIs processando a função como na execução com 2 WIs, a queda de desempenho na execução com 4 WIs é pelo custo da criação de um maior número de WIs. Já com 8 WIs, o mesmo comportamento ocorre, porém, são 5 WIs que finalizam na primeira iteração, sendo que o WI (5) processa pouca parte da função, ficando os WI (3) e (4) com a maior parte do processamento.

Tabela 19. Tempos de Execução e Ganhos na execução do intervalo [0,4] para a função Polinomial.

Quantidade de WIs	Tempo	Ganho
1	0,9673s	x
2	0,4772s	2,02
4	0,332s	2,91
8	0,3245s	2,98

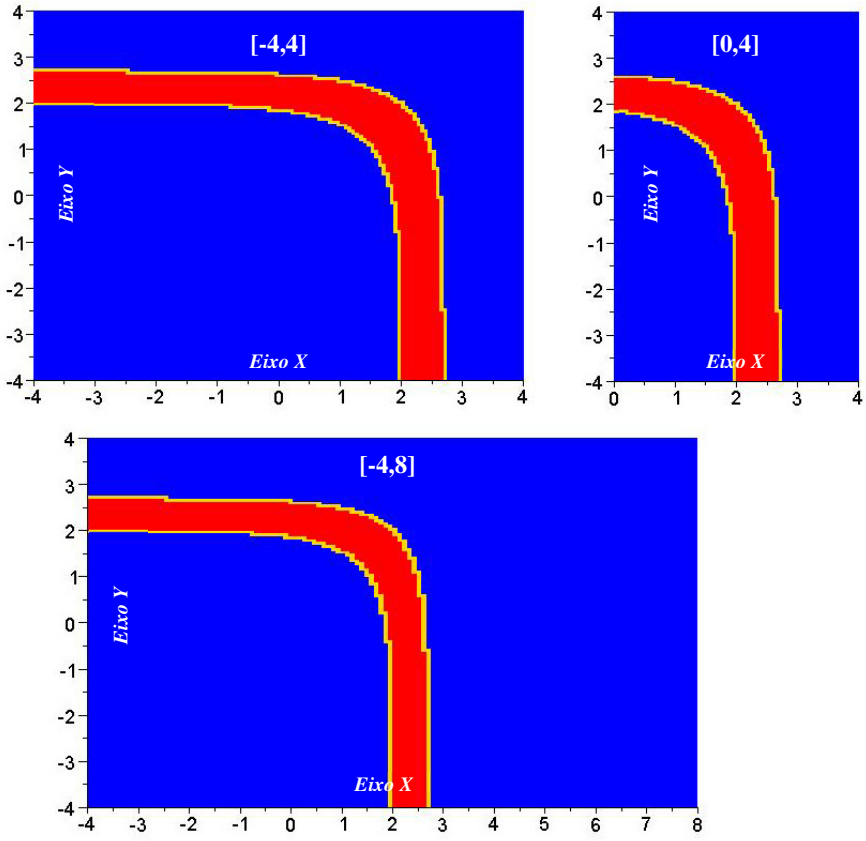
Para o intervalo [0,4], o PSIVIA-HP foi escalável até 8 WIs conforme é possível perceber na figura 17 e na tabela 11. Não sendo o tempo com 4 e 8 WIs proporcional ao aumento de WIs utilizados, a escalabilidade não é linear. Não há melhora proporcional, pois a função não é uniforme considerando o eixo *x*, levando o primeiro e o último WI processarem as bordas da função, e o segundo e terceiro WIs processarem o centro da função que é a maior parte do processamento quando a execução é com 4 WIs. Já com 8 WIs o comportamento é similar, pois os WIs (0),(1),(6) e (7) que estão nas bordas do espaço de busca, processam intervalos que não são da função finalizando na primeira iteração.

Os melhores desempenhos alcançados foram com 8 WIs, sendo que os tempos de execução dos intervalos [-4,4] e [0,4] foram equivalentes. A equivalência se deve a ambos estarem trabalhando com 4 WIs após a primeira iteração, pelas razões já citadas.

A análise final do desempenho do *Kernel* do PSIVIA-HP em CPU processando a função polinomial é que, o desbalanceamento de carga prejudica a utilização do nível máximo de paralelismo, porém mesmo com o desbalanceamento há ganho de desempenho válido. A posição da função no espaço de busca e o tamanho do espaço de busca, também influenciam no desempenho, devido as porções não pertencentes a função.

Para a avaliação de desempenho do *Kernel* da função exponencial é apresentado na figura 18 o posicionamento da função no espaço de busca.

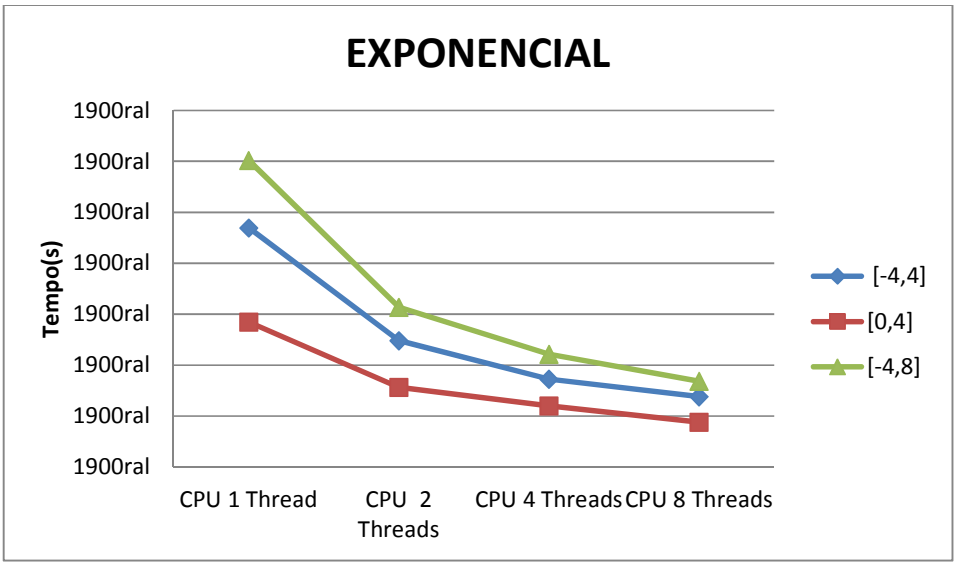
Figura 20. Espaço de busca para a função exponencial para todos os intervalos utilizados.



Em relação à função polinomial, a função exponencial possui uma maior quantidade de intervalos válidos pertencentes à função, segundo o eixo *x*. Antes de iniciar as análises, é válido ressaltar que as análises dos intervalos [-4,4] e [-4,8] serão distintas em relação ao intervalo [0,4]. Já que o intervalo [0,4] possui uma menor porção da função, como é possível perceber na figura 18.

A figura 19 apresenta os valores de tempo de execução do *Kernel* do PSIVIA-HP para a função exponencial.

Figura 21. Tempo de execução médio em CPU com variação de WI para a função exponencial.



Na função exponencial houve ganho para os três intervalos com o aumento de WIs, sendo mais expressivo quando o aumento foi de 1 para 2 WIs. Logo, é possível afirmar que para esta carga há escalabilidade não linear. Para os intervalos [-4,4] e [-4,8], o ganho foi menor com o aumento do grau de paralelismo do que em relação à mudança da execução sequencial para a paralela. A razão, como é possível perceber na figura 18, ao se dividir o eixo *x* entre os WIs, os WIs responsáveis pela curva da função terão maior processamento do que aqueles que não estão com parte da curva.

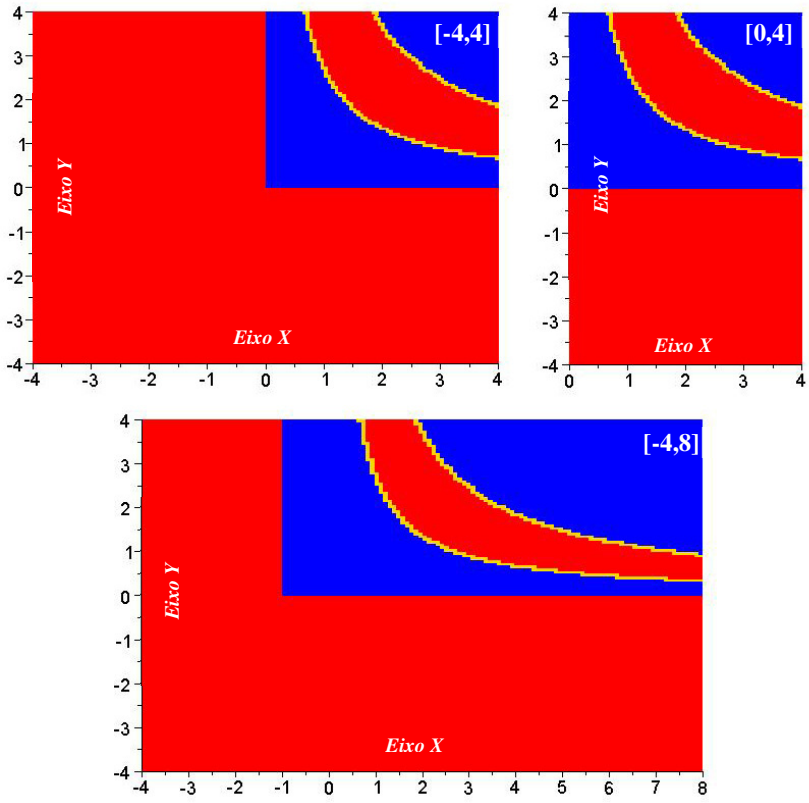
A razão de o intervalo [-4,4] obter 0,693 segundos em relação ao intervalo [-4,8] que obteve 0,844 segundos, ambos com 8 WIs, se deve ao fato que o intervalo [-4,8] possui maior parcela de intervalos que não pertencem à função, sendo atribuídos aos WIs (5), (6) e (7) que acabam finalizando na primeira iteração.

O menor nível de ganho com o aumento do nível de paralelismo de 4 para 8 WIs no intervalo [0,4], é pelos WIs (6) e (7) processarem intervalos que não pertencem à função.

Um fator importante na análise dos resultados da função exponencial, além da influência do intervalo inicial mencionado, é o desbalanceamento no processamento da curva. Pois, o desbalanceamento em relação à WIs que finalizam na primeira iteração pode ser solucionado inicialmente por lançar mais WIs do que EPs disponíveis. Porém, WIs que processam a curva, por exemplo, irão processar mais do que outros WIs que possuem parte da função, sendo necessária outra abordagem para balanceamento da carga. E caso não se saiba o formato da função, será necessário uma avaliação dinâmica para balanceamento da carga.

A função logarítmica utilizada possui a característica que, parte do espaço de busca é composto por intervalos válidos (em vermelho na figura 20) que podem ser encontrados na primeira iteração. A figura 20 evidencia este fato, exibindo o formato da função para os três intervalos utilizados.

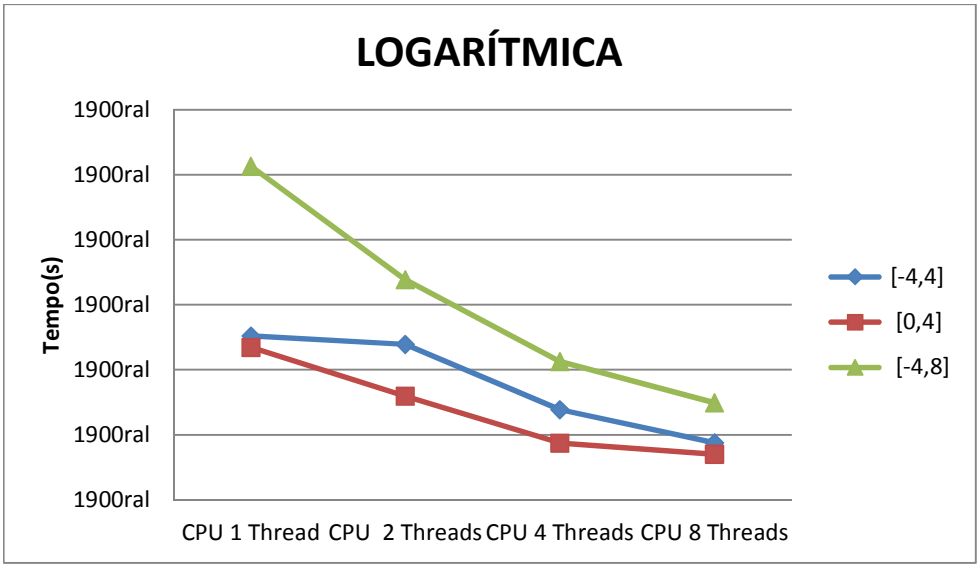
Figura 22. Espaço de busca para a função logarítmica para todos os intervalos utilizados.



Dois aspectos devem ser ressaltados antes de apresentar as análises da função logarítmica. O primeiro é que com a divisão do intervalo de entrada, o intervalo [-4,4] e [-4,8] podem levar WIs a serem finalizados na primeira iteração. Pois, a porção do intervalo já pode ser considerada como válida na primeira iteração. O segundo é que, como na função exponencial, na função logarítmica não pode ser comparado diretamente o desempenho do intervalo [-4,8] com os intervalos iniciais [0,4] e [-4,4], pois o intervalo [-4,8] possui uma curva maior, por consequência maior quantidade de intervalos a serem encontrados.

A figura 21 apresenta os resultados de tempo de execução do PSIVIA-HP para a função logarítmica.

Figura 23. Tempo médio de execução em CPU com variação de WI para a função logarítmica.



Com os resultados apresentados na figura 21 é possível determinar que os intervalos [4,4] e [-4,8] são escaláveis e o intervalo [-4,4] é escalável a partir da utilização de 2 WIs.

De acordo com a figura 21 é possível perceber, também, que o tempo de execução com 1 WI para os intervalos [-4,4] e [0,4] estão próximos, sendo 1,258 segundos para [-4,4] e 1,173 segundos para o intervalo [0,4]. A parte do espaço de busca que demandará maior processamento é comum entre os intervalos, de acordo com a figura 20. Porém, há um espaço em vermelho que está compreendido no intervalo [-4,0] para o intervalo inicial de entrada [-4,4], que é encontrado na segunda iteração, sendo esta a diferença de demanda de processamento entre os intervalos. O intervalo [-4,0] é encontrado na segunda iteração, pois na primeira iteração o intervalo [-4,4] sofre bisseção gerando os intervalos [-4,0] e [0,4] e na iteração seguinte o intervalo [-4,0] é tido como válido.

Com o aumento do número de WIs para 2 no intervalo [0,4], a curva é dividida entre os 2 WIs. Porém, não é o mesmo que ocorre quando é utilizado como entrada o intervalo [-4,4], pois o primeiro WI irá trabalhar com o intervalo [-4,0] que é considerado válido, finalizando a sua execução na primeira iteração, deixando então o processamento da curva apenas para o segundo WI.

Utilizar 4 WIs garante melhora no tempo de execução para ambos os intervalos, como apresentado na figura 21. No intervalo [0,4] isso ocorre, pois a divisão realizada atribui a curva para todos os WIs. No entanto, no intervalo [-4,4] apesar da melhora, apenas 2 dos 4 WIs processam a curva, uma vez que os 2 primeiros WIs são atribuídos para o intervalo [-4,0] que finalizam na primeira iteração.

Na execução com 8 WIs o intervalo [-4,4] é executado em 0,438 segundos e o intervalo [0,4] é executado em 0,351 segundos. Este melhor desempenho para o intervalo [0,4], é por ter durante toda a execução 8 WIs processando a curva, o que não acontece com o intervalo [-4,4], pois após a primeira iteração apenas 4 WIs estarão processando a curva. É válido ressaltar que os valores de tempo de execução com 8 WIs são mais próximos do que os tempos de execução com 2 e 4 WIs para os intervalos citados, pois a execução com 8 WIs no intervalo [0,4], gerou redundância, comportamento já descrito na seção 3.2.

Dos intervalos utilizados, aquele que apresentou maior escalabilidade foi o intervalo [-4,8], justamente por possuir uma curva maior para processar e, sendo assim, é mais influenciado pelo aumento do nível de paralelismo. O efeito de WIs serem subutilizados por processarem porções de intervalo que finalizam na primeira iteração, começa a ocorrer a partir de 4 WIs, com o primeiro WI processando o intervalo [-4,-1]. Com 8 WIs em execução serão apenas os 2 primeiros WIs que irão finalizar na primeira iteração. Sendo assim, o nível de subutilização de WIs foi menor do que nas funções polinomiais e exponenciais.

O aumento da quantidade de WIs para todas as funções permitiu decrescer o tempo de execução, porém em alguns cenários houve pouco ganho com o aumento de WIs, por existir porções de intervalos que levem WIs finalizarem na primeira iteração. O intervalo [0,4] para a função polinomial e logarítmica, quando comparados com intervalos que geram o mesmo esforço computacional, obteve melhor desempenho por possuir apenas porções de intervalos pertencentes à função. Logo, é necessário eliminar os espaços que levam a finalizar na primeira iteração antes da execução paralela, para que todos os WIs possam processar apenas porções que sejam partes da função.

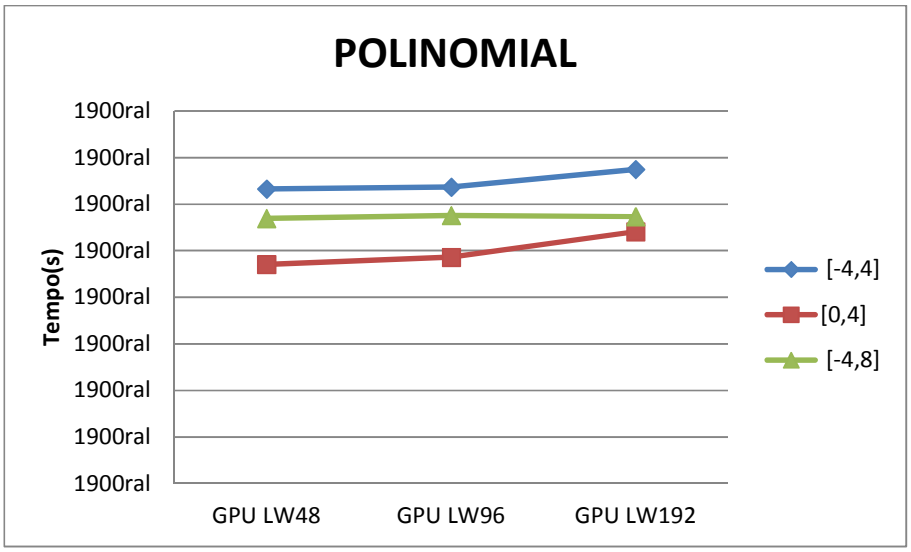
5.1.2 Comportamento do PSIVIA-HP em GPU

Das análises realizadas, a análise de escalabilidade do PSIVIA-HP para GPU não foi realizada, já que o nível de paralelismo (GW = 1344) foi o mesmo para todas as execuções. Como o parâmetro variado foi o LW, a análise é direcionada para identificar o tamanho de grupo mais adequado para ser processado em cada SMP. Essa melhor adequação é pelo fato da natureza SIMD da GPU, em que todos os WIs dentro de um SMP executam a mesma instrução. Logo, caso um WI em um grupo tome um caminho maior no código, todos os WIs

do grupo executam esse caminho (AMD, 2010). Sendo assim, WIs que finalizam na primeira iteração irão deixar SPs ociosos, caso algum WI dentro de um grupo tenha alguma parte da função para processar. Por essa razão, o tamanho do grupo é fundamental na execução, pois determina o nível de ociosidade na utilização dos SPs.

A figura 22 apresenta os resultados de tempo para a função polinomial.

Figura 24. Tempo de execução médio do *Kernel* em GPU com variação de LW para a função polinomial.



Um fator importante que deve ser analisado antes dos resultados apresentados na figura 22 é a variância média dos tempos de execução do *Kernel* em GPU. Para essa execução a variância foi de 0,00255, que é estatisticamente irrelevante. Este valor de variância é justificado pelo fato que a GPU é dedicada apenas para a execução do *Kernel*, que não é o caso da CPU que há concorrência com os processos do sistema.

De acordo com a figura 22, o tempo do intervalo [-4,4] foi superior ao do intervalo [-4,8] para todas as variações do LW. O tempo superior é em virtude da quantidade de intervalos encontrados, onde com o intervalo [-4,4] foram encontrados 2092032 intervalos e com o intervalo [-4,8] foram encontrados 1739964 intervalos. Como ambos os intervalos produzem redundância pelo fato de serem divididos para 1344 WIs, a diferença do número de intervalos encontrados é pela quantidade de bisseções realizadas em virtude do intervalo inicial de entrada, como descrito na seção 3.1.

No intervalo de [-4,8] a variação do LW não influenciou no tempo de execução do *Kernel*, isto indica que WIs que finalizam na primeira iteração, ou estavam todos em um grupo e foram escalonados e/ou estes WIs estão em menor número em grupos com WIs com porções da função para processar.

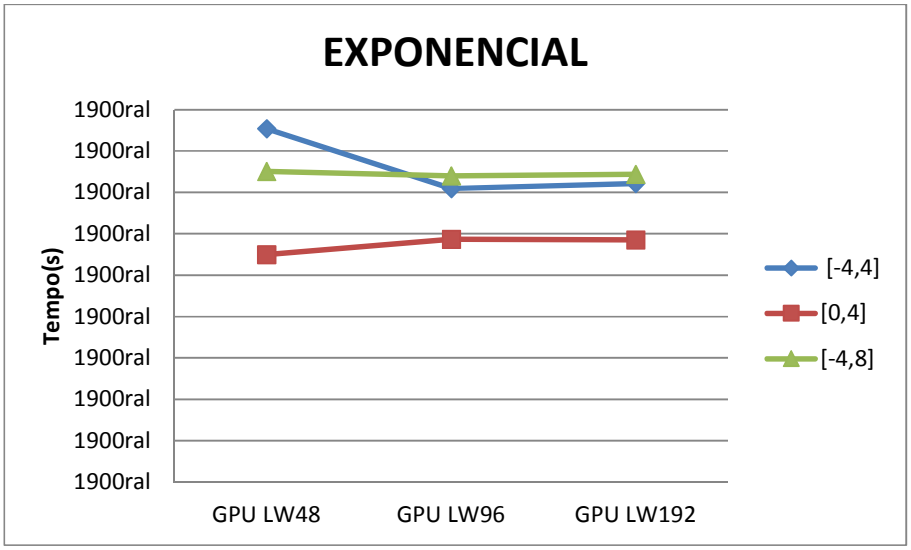
A função no intervalo [-4,4] conforme exibe a figura 13 na seção 5.1.1, se encontra a direita do espaço de busca, como consequência WIs a partir do WI 672 é que serão responsáveis pelo processamento da função. Não houve variação no tempo de execução médio com a mudança do LW de 48 para 96, porém o tempo de execução médio aumentou quando o valor de LW foi 192. Como o LW igual a 192 o tamanho do grupo é maior, WIs que finalizam na primeira iteração estavam aglomerados nos três primeiros grupos e foram escalonados. Entretanto, o quarto e o último grupo (que processam as bordas da função em relação ao eixo *x*) possuem WIs que finalizam na primeira iteração ocupando SPs, enquanto WIs que possuem partes da função estão processando.

Para as execuções da função Polinomial em GPU o intervalo [0,4] obteve os melhores resultados, por possuir menor quantidade de intervalos que não pertencem à função. O melhor resultado foi para o grupo com menor tamanho, LW igual a 48, que pelo fato de ser menor, reduz a probabilidade de WIs que finalizam na primeira iteração estarem em grupos com WIs que processam a função. Com o aumento do LW há uma pequena piora no desempenho, pois WIs com porções inválidas estavam contidos em grupos com partes da função para processar.

Intervalos reduzidos, com menor quantidade de intervalos que não são partes da função, como o intervalo [0,4], obtêm melhores resultados para funções com as características da função polinomial utilizada. Porém, caso não seja possível determinar um intervalo de entrada inicial, a variação do parâmetro LW não implica em ganho expressivo, uma vez que para todas as execuções o desvio padrão foi menor que 0,05, sendo estatisticamente desconsiderado. Esta pequena variação do tempo de execução com a mudança do parâmetro LW, também se apresenta inexpressiva quando considerado o tempo de preparação de dispositivo, que é apresentado na seção 5.3.

A função exponencial apresenta alguns comportamentos similares na execução em GPU, cuja análise e resultados são apresentados em sequência.

Figura 25. Tempo de execução médio em GPU com variação de LW para a função exponencial.



Para a função exponencial os tempos de execução médios permaneceram constantes com o aumento de LW de 96 para 192 para os intervalos [0,4] e [-4,8], e uma pequena piora para o intervalo [-4,4]. A razão é pelo fato da função, como se pode perceber na figura 18, possuir intervalos a serem avaliados do ponto -4 até aproximadamente o ponto 3 e, assim, o aumento do LW apenas diminuiu a quantidade de grupos, concatenando os WIs que efetivamente possuem intervalos a avaliar.

O intervalo [0,4] obteve o melhor desempenho, porém deve ser ressaltado que esse intervalo como apresentado na figura 18, possui uma menor quantidade de intervalos pertencentes à função do que os intervalos [-4,4] e [-4,8]. Mas, ao analisar o seu comportamento, há uma queda no desempenho quando se eleva o LW de 48 para 96. Novamente, em razão de WIs que finalizam na primeira iteração estarem em grupos que processam a função.

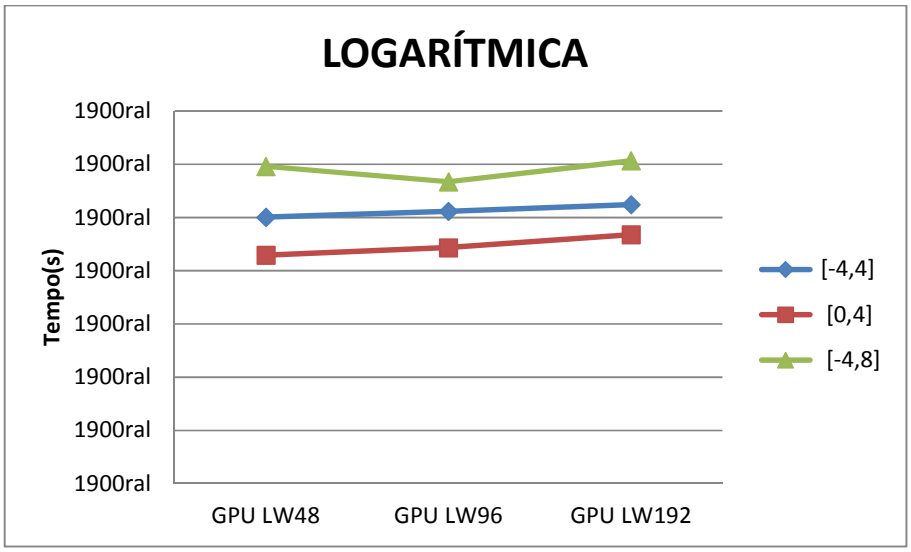
Já os intervalos [-4,4] e [-4,8] cobrem a mesma extensão da função, conforme apresentado na figura 18. Para o intervalo [-4,8] modificar o LW não altera o tempo de execução médio, pois aumentar o LW apenas concatena os grupos que possuem porções da função para processar. Já o intervalo [-4,4] o valor de LW igual a 48 alcançou o desempenho mais baixo, uma vez que dos 28 grupos criados, apenas os 5 últimos grupos é que irão possuir intervalos que não são parte da função, os demais WIs estarão divididos em 23 grupos que serão escalonados e não utilizarão o *pipeline* de cada SP.

Para as funções processadas em GPU que possuam as características da função exponencial utilizada, a variação do LW de 96 para 192 não exerce influência, como comprova o desvio padrão calculado inferior a 0,05. Caso a função possua na extensão do

eixo *x* grande parte a ser processada, é válida a configuração com grupos maiores para utilizar o potencial do *pipeline* nos SPs. Conforme a melhora alcançada com a mudança do LW de 48 para 96 no intervalo [-4,4], apresentada na figura 23.

A figura 24 representa o comportamento da função logarítmica.

Figura 26. Tempo de execução médio em GPU com variação de LW para a função logarítmica.



A função logarítmica possui grande espaço de intervalo contínuo pertencente à função, como apresentado na figura 20, que leva WIs finalizarem na primeira iteração. Essa análise é importante para a discussão dos resultados em seguida.

Especificamente sobre os resultados de cada intervalo, o intervalo [-4,8] foi o mais lento, porém, como também apresentado na figura 20, este intervalo possui uma curva maior para processar. Os valores próximos de tempo para o LW igual a 48 e 192, 0,5958 segundos e 0,6068 segundos respectivamente, apresentam menor adequação do que a execução com o LW igual a 96. O desempenho inferior para o LW igual a 48 são pelos 28 grupos criados, sendo que, apenas grupos com índice superior a 14 é que irão processar a curva; os demais irão finalizar na primeira iteração. Já para o LW igual a 192, dos 7 grupos criados, os grupos de índices 0 e 1 irão finalizar na primeira iteração, o grupo de índice 2 conterá 30% de WIs que irão finalizar na primeira iteração e os grupos restantes irão processar o restante da curva.

O melhor desempenho alcançado com o LW igual a 96 foi que dos 14 grupos criados, metade finalizou na primeira iteração e a outra metade processou a curva. Sendo assim, todos os 7 SMPs estavam com grupos processando a curva na maior parte do tempo.

O comportamento em relação à variação do LW para os intervalos [-4,4] e [0,4] foi o mesmo, conforme a figura 24. A diferença de tempo entre os dois intervalos é pela metade

dos WIs na execução com o intervalo [-4,4], finalizar na primeira iteração. O declínio do desempenho com a variação do LW se deve aos WIs que não processam a curva e finalizam na terceira iteração (metade azul e metade em vermelho no intervalo [0,4] sem a curva na figura 20, quando analisado o eixo *y*). E os WIs que processam parte da curva com maior inclinação (a inclinação demanda maior quantidade de bisseções para determinar um intervalo). Os WIs que processam parte da curva com maior inclinação também levam a SPs a ficarem ociosos, uma vez que os WIs que já finalizaram o seu processamento tomam o maior caminho dentro do SMP por haver ainda um WI processando parte da curva. Por todas as razões citadas, a variação do LW leva à piora do tempo de execução, já que grupos menores diminuem a probabilidade de ocorrer ociosidade dos SPs.

A execução de funções com as características da logarítmica utilizada em GPU, deve ser realizada com tamanho de grupos menores para diminuir o nível de ociosidade dos SPs.

Apesar de alguns resultados a variação do LW não apresentar mudança no tempo médio de execução, grupos menores com LW igual a 48 alcançaram melhor desempenho, excetuando apenas no intervalo [-4,4] da função polinomial.

A seção seguinte é destinada à análise de desempenho global de tempo da aplicação, não apenas do *Kernel* do PSIVIA-HP.

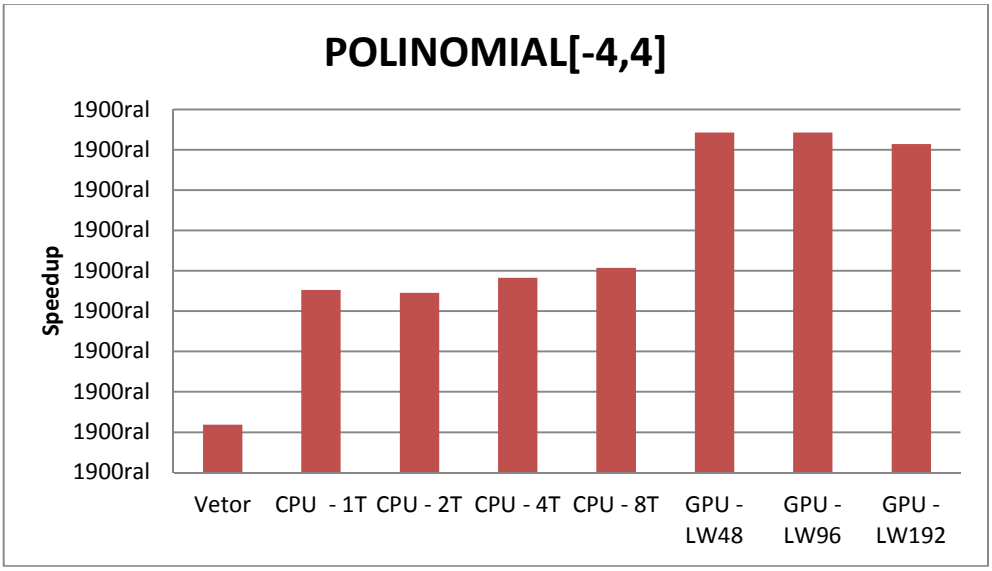
5.2 Análise de Ganho do PSIVIA-HP

Nesta seção será analisado o desempenho global do PSIVIA-HP através do cálculo de *speed up* (equação 1), apresentada na seção 4.2. Para que fosse possível analisar o desempenho global e utilizar a equação de *speed up*, o tempo médio de execução foi calculado através da equação 2, tempo total, também apresentada na seção 4.2. É válido ressaltar que não foi apresentado o ganho de desempenho do SIVIA sequencial em lista encadeada, pois o seu tempo médio de execução foi utilizado no numerador da equação de *speed up* (*tempo sem melhoria).* Sendo assim, a proporção de ganho de desempenho de todas as demais versões dos algoritmos é em relação ao seu tempo médio de execução.

Nos gráficos são apresentados também os ganhos de uma versão sequencial do SIVIA que é a mais próxima da implementação do *kernel* do PSIVIA-HP, implementada através de um vetor alocado simulando uma lista dinâmica.

Na figura 25 é exibido o gráfico com os valores de ganho da função polinomial para o intervalo [-4,4].

Figura 27. Resultados de *speedup* para função polinomial [-4,4].



De acordo com a figura 25, não aplicando nenhuma técnica paralela, apenas utilizando o SIVIA sequencial com vetor alocado ao invés do SIVIA sequencial com lista encadeada, foi possível alcançar um ganho de 5 vezes. A razão, como já abordado na seção 4.3, é que a versão da Lista Encadeada gera mais acessos à memória principal sem utilizar o recurso da memória cache. Já com a utilização do PSIVIA-HP em CPU o ganho foi de 22 vezes com apenas 1 WI (sequencial). Logo, utilizar OpenCL traz ganho mesmo sem paralelismo, uma vez que ambos os algoritmos foram executados sequencialmente. Como não é possível analisar o compilador OpenCL de ambos os fabricantes, uma hipótese que possa ter ocasionado este ganho com a execução de apena 1 WI é que, a compilação é realizada especificamente para o periférico e não para uma família de periféricos, logo possui otimizações.

Apesar da precisão da análise, é válido ressaltar sobre a pequena queda de desempenho ao aumentar de 1 para 2 WIs. Esta redução ocorreu pelo custo de criar um maior número de WIs e que 1 dos WIs termina na primeira iteração.

Os ganhos obtidos com 4 WIs e 8 WIs foram de 24 e 25 vezes respectivamente; porém ao dobrar a quantidade de WIs esperava-se que o *speedup* também aumentasse próximo ou maior do que duas vezes. Logo, o tempo de preparação do dispositivo está com um peso maior do que as melhorias no aumento do nível de paralelismo. Isto pode ser comprovado analisando o

ganho apenas do *Kernel* com o aumento de 2 para 4 WIs, pois o tempo de execução caiu de 0,809 segundos para 0,449 segundos, uma melhora de 1,8 vezes que não foi acompanhada pela aplicação, conforme a figura 25.

A influência do tempo de preparação do dispositivo pode ser confirmada ao analisar a porção do tempo de *Kernel* em relação ao tempo total da aplicação. O tempo de *Kernel* correspondeu 17% em relação ao tempo total da aplicação com apenas 1 WI e 7% com 8 WIs. Logo, o tempo de *Host* é fixo e independe do desempenho do *Kernel*. A tabela 11 comprova este fato ao subtrair o tempo de execução do *Kernel* do tempo total da aplicação.

Tabela 11. Tempos de execução desconsiderando o tempo de *Kernel* para a função polinomial em CPU.

Algoritmo	Tempo Dispositivo
CPU 1 Thread	3.763
CPU 2 Threads	3.837
CPU 4 Threads	3.831
CPU 8 Threads	3.756
Variância	*0.002*
GPU LW48	1.187
GPU LW96	1.229
GPU LW192	1.189
Variância	*0.001*

A tabela 11 apresenta dois aspectos fundamentais: 1) O aumento de WIs ou a variação do LW não influencia no tempo de preparação do dispositivo, comprovado pela variância, tanto para CPU quanto para GPU; 2) Em relação aos fabricantes dos compiladores e dispositivos utilizados nesta dissertação, o tempo de preparação da CPU é maior do que da GPU.

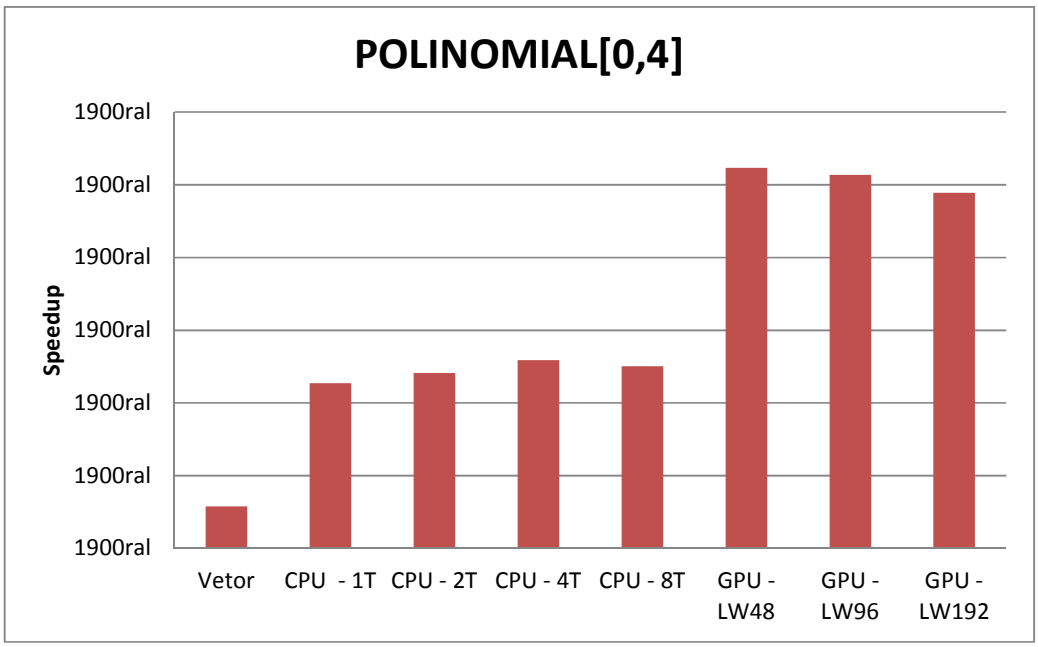
Apesar destes tempos apresentados na tabela 11 serem de resultados coletados para a função polinomial, os mesmos se mantêm para outras funções, pois a influência do intervalo e de qual função será executada só é sentida na execução do *Kernel*. Diante das análises mencionadas é possível concluir que, como há um custo de preparação do dispositivo, o compilador OpenCL deve ser superior aos compiladores de arquiteturas de propósito geral para que a execução do *Kernel* seja rápida o suficiente para ter validade a sua utilização. Os resultados obtidos comprovam isto, já que o tempo médio de execução para 1 WI (sequencial) em CPU foi de 2,39 segundos e o tempo médio de execução da lista encadeada foi de 128,6 segundos. Caso se avalie o ganho apenas do *Kernel*, este seria de 53 vezes.

Analisando o ganho da utilização da GPU, o ganho mínimo foi de 40 vezes e o ganho máximo de 42 vezes. O tempo de execução do *Kernel* em GPU é maior do que em CPU,

porém o tempo de preparação do dispositivo em GPU é menor, possibilitando ganhos superiores a CPU.

Uma análise das razões do custo de preparação dos dispositivos é apresentada na seção 5.3.

Figura 28. Resultados de *Speedup* para função polinomial [0,4].



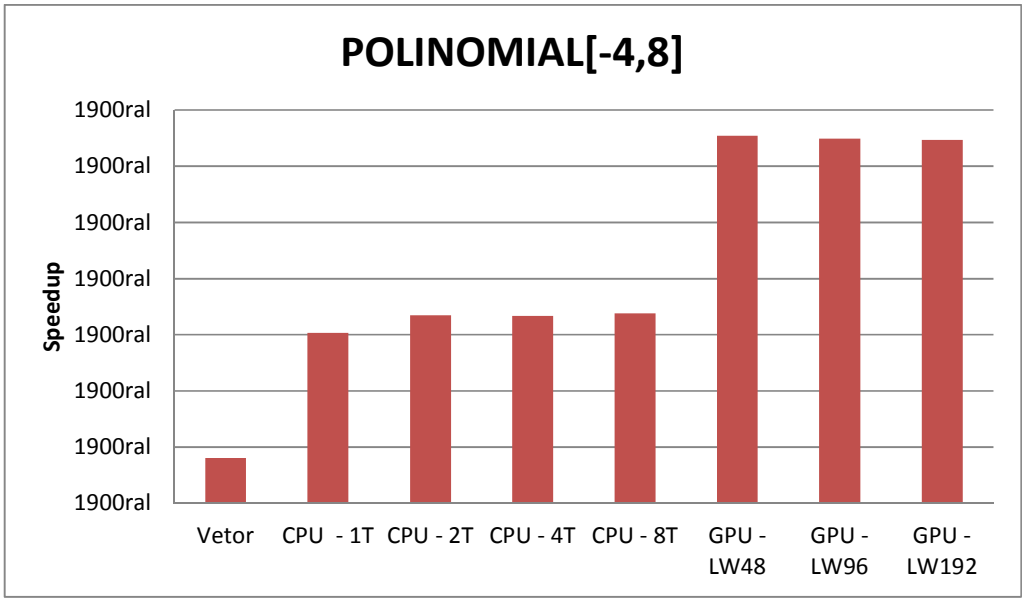
Na função polinomial com o intervalo [0,4], o ganho na utilização do OpenCL considerando apenas 1 WI (sequencial) levou a um ganho de 22 vezes em relação a lista encadeada.

O PSIVIA-HP em CPU apresentou escalabilidade pequena até 4 WIs, alcançando desempenho máximo de 25,8 vezes. Quando foi utilizado o nível máximo de paralelismo com 8 WIs, houve uma pequena queda no desempenho, porém ao se avaliar os tempos médios de execução do *Kernel* com 4 e 8 WIs, apresentado na figura 17, os tempos médios de execução são considerados os mesmos, já que a variância é 0,00002888. Sendo assim, a redução do ganho apresentada foi em relação ao tempo médio de execução do *Host* e não da execução paralela em CPU.

Ao se avaliar o ganho do PSIVIA-HP em GPU, os ganhos foram respectivamente 52 vezes para LW igual a 48, 51 vezes para LW igual a 96 e 48 vezes para LW igual a 192. A proporção do tempo de execução do *Kernel* na GPU, no melhor caso, correspondeu aproximadamente a 44,7% do tempo total da aplicação. Logo, melhorias para redução do tempo de execução do *Kernel* em GPU possuem maior peso no tempo da aplicação do que em CPU.

É válido ressaltar que as maiores taxas de ganho para este intervalo são, conforme a análise do tempo de execução do *Kernel* já apresentada, por possuir pequenas porções de intervalo que não são partes da função, ficando a maior parte do processamento de intervalos que pertencem à função.

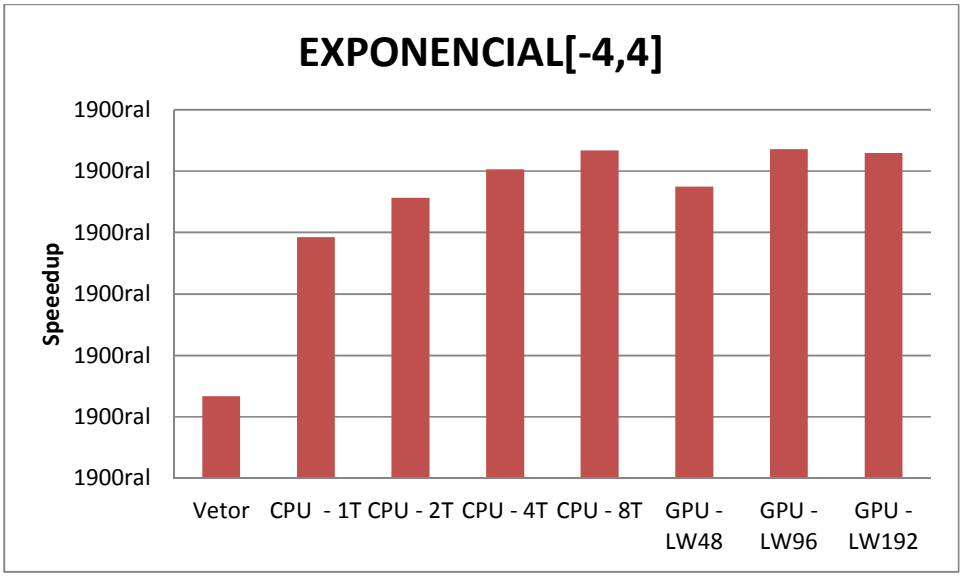
Figura 29. Resultados de *Speedup* para função polinomial [-4,8].



No intervalo [-4,8] a utilização de 1 WI levou a um ganho de 15 vezes e de 16 vezes com a utilização de 2 WIs. Porém, o aumento no nível de paralelismo para 4 ou 8 WIs não elevou o ganho. Analisando o tempo de execução do *Kernel*, houve uma pequena variação no tempo de execução quando variado o número de WIs de 2 em diante, sendo esta pequena variação suprimida pela variação média do tempo de *Host*. Este mesmo comportamento também ocorreu em GPU, onde, o ganho máximo foi de 32 vezes.

Em sequência serão analisados os resultados de ganho para a função exponencial.

Figura 30. Resultados de *Speedup* para função exponencial [-4,4].



De acordo com a figura 28, a utilização do vetor ao invés da lista encadeada levou a um ganho de 6 vezes, pelas mesmas razões citadas para a função polinomial. Em CPU a utilização de apenas 1 WI (sequencial) obteve ganho de 19 vezes e com 8 WIs (nível máximo de paralelismo) o ganho foi de 26 vezes. Estes níveis de ganhos da aplicação seguiram o mesmo comportamento do *Kernel* com o aumento do número de WIs, sendo então escalável até 8 WIs. O comportamento foi o mesmo, pois a função exponencial demanda mais processamento do que a função polinomial utilizada, isto é comprovado analisando a proporção do tempo de execução do *Kernel* em relação ao tempo de *Host*, que foi de 35,8% para 1 WI e 14% para 8 WIs.

Para os resultados apresentados, o *speedup* para CPU foi sublinear, ou seja, se houvesse um dispositivo com mais EPs disponíveis o ganho iria atingir um limite e não haveria melhora, mesmo com o aumento do número de WIs.

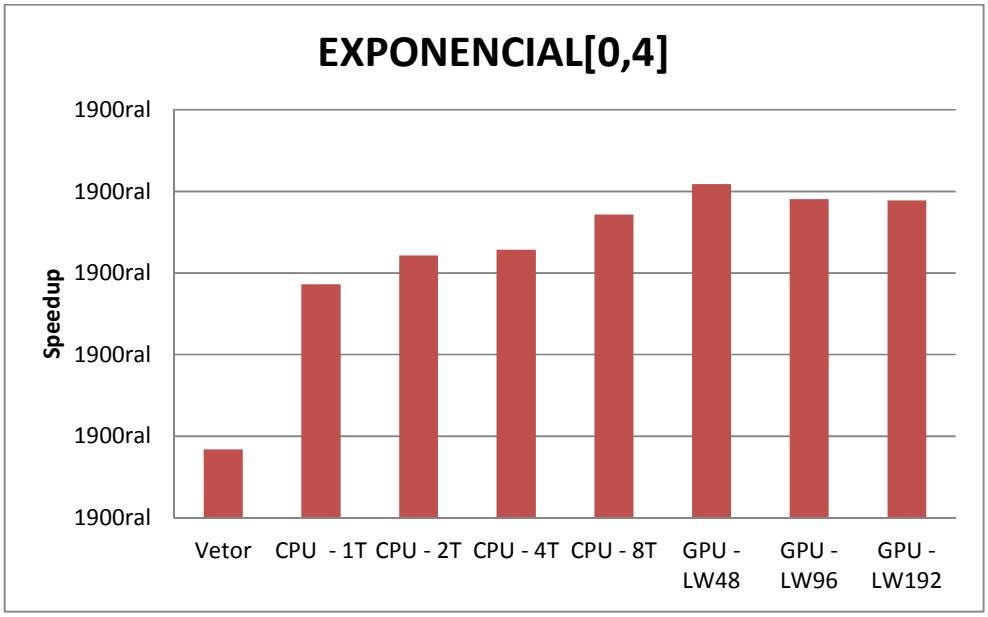
O menor ganho obtido na execução em GPU foi com LW igual a 48, em função do desempenho do *Kernel*, conforme descrito na seção 5.1.2.

A melhor execução em CPU com 8 WIs obteve ganho de 26 vezes e em GPU com LW igual a 96 e 192 também obteve ganho de 26 vezes. Sendo assim, para uma função com estas características em conjunto com este intervalo de entrada, quaisquer dos dispositivos podem ser utilizados para execução. Porém, o custo para adquirir a CPU utilizada é superior ao da GPU utilizada, sendo então mais viável à utilização da GPU, por alcançar ganho equivalente com um custo financeiro menor.

A razão para o desempenho em GPU não ter sido superior ao da CPU como na função polinomial se deve a dois fatores: 1) A curva, conforme apresentando na figura 18, é onde há maior demanda de processamento, sendo assim os WIs que processam a curva em um grupo fazem com que os demais WIs tenham que esperar a sua finalização, levando a ociosidade dos SPs; 2) Este processamento é realizado em cada EP na GPU que possui frequência de 598 MHz, diferentemente da CPU que possui frequência de 2,7 GHz em cada EP. Logo, uma técnica dinâmica para balancear os intervalos entre os EPs deve ser empregada em GPU. Porém, há a dificuldade de identificar previamente as discrepâncias de processamento de cada WI, pois os intervalos apenas são avaliados quando o algoritmo está em execução e, juntamente com este fato, cada função possui um domínio diferente. Isto pode ser confirmado através da análise do tamanho máximo da lista de intervalos a serem avaliados em um WI, em uma execução com LW igual a 48. No grupo 20, foi levantado que o WI 991 o tamanho máximo da sua lista foi de 159617 intervalos, já o WI 992 o tamanho máximo da sua lista foi de 19505 intervalos.

A figura 29 apresenta os valores de ganho para a execução da função exponencial para o intervalo [0,4].

Figura 31. Resultados de *Speedup* para função exponencial [0,4].



Os níveis de ganho para o intervalo [0,4] foram inferiores ao intervalo [-4,4] por serem cargas de trabalho diferentes, como já mencionado, pois o intervalo [0,4] é uma porção do intervalo [-4,4].

No processamento do PSIVIA-HP na CPU, houve ganho apenas com a utilização do OpenCL, conforme apresentado na figura 29. Para este intervalo o comportamento escalável do *Kernel* se manteve para aplicação, pela mesma razão citada no intervalo anterior.

O aumento do nível de paralelismo em CPU de 2 para 4 WIs levou a um ganho de apenas 0,4 vezes, pois na execução com 4 WIs um WI finalizou na primeira iteração, conforme levantamento do tamanho das listas de intervalos a serem avaliados, apresentado na tabela 12.

Tabela 12. Tamanho máximo da lista de intervalos a serem avaliados com 2 e 4 WIs.

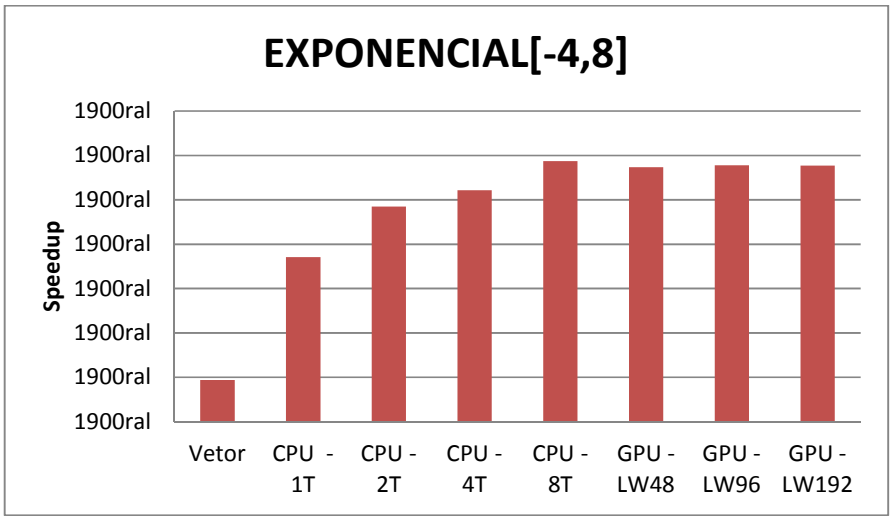
	Dois WIs	Quatro WIs
WI(0)	1.943.790	582.808
WI(1)	1.928.416	575.962
WI(2)	X	549.124
WI(3)	X	1

Como o espaço de busca é reduzido e quase a metade desse espaço possui intervalos não pertencentes à função, a utilização de um maior número de grupos em GPU, possibilitou segmentar WIs em grupos que possuem parte da função para processar e WIs que não possuem partes da função. Essa segmentação possibilita que grupos que não possuam parte da função sejam escalonados, deixando os recursos dos SPs apenas para os WIs que irão processar a função. Por esta razão, o LW igual a 48 (maior quantidade de grupos criados) alcançou ganho de 25 vezes, sendo superior inclusive aos ganhos da CPU.

Os ganhos da aplicação seguiram o mesmo padrão do *Kernel*, não havendo modificação dos níveis de ganho com a variação do LW. Vale mencionar novamente que a proporção do tempo do *Kernel* em relação ao tempo total da aplicação em GPU é maior. Sendo assim, o tempo do *Kernel* em GPU exerce maior influencia no tempo total da aplicação do que em CPU.

Em seguida serão apresentados os resultados de ganho para o intervalo [-4,8].

Figura 32. Resultados de *speedup* para função exponencial [-4,8].



As principais análises de acordo com a figura 32 são que, em CPU houve ganho de 18 vezes com apenas 1 WI, a aplicação foi escalável até 8 WIs alcançando ganho máximo de 29 vezes e o desempenho da execução em CPU foi superior a da GPU.

Ao se avaliar os níveis de ganho para este intervalo na CPU em comparação com o intervalo [-4,4], este intervalo alcançou maiores níveis de ganho. Porém, o tempo médio de execução do *Kernel* no intervalo [-4,4] foi inferior ao intervalo [-4,8], sendo então a diferença em relação ao tempo de *Host*. Esta afirmação fica mais clara analisando a tabela 13.

Tabela 13. Desvio padrão entre as execuções do tempo de *Kernel* dos intervalos [-4,4] e [-4,8].

Tempo			
	Intervalo		
Quantidade de Wis	[-4,4]	[-4,8]	Desvio Padrão
1	2.348(s)	3.009(s)	0.467397582
2	1.2431(s)	1.569(s)	0.2304461
4	0.8626(s)	1.1076(s)	0.173241161
8	0.6938(s)	0.8443(s)	0.106419571

Na tabela 13 os valores de desvio padrão indicam que a diferença entre os tempos de execução são próximos, logo a diferença de ganho entre os intervalos para aplicação é pela execução sequencial do *Host* que pode ser influenciada por trocas de contexto de processos e demais razões de administração do S.O..

Já em GPU, a alteração do valor do LW não variou o nível de ganho, já que todas as execuções alcançaram ganho de 28 vezes em relação à lista encadeada. Como ocorreu para os

outros intervalos na função exponencial, os níveis de ganho da aplicação seguiram o padrão dos níveis de desempenho do *Kernel*, pelas razões já citadas. Vale ressaltar que, como já mencionado, apesar da CPU utilizada ter alcançado ganho superior a GPU utilizada, o uso da GPU é mais aconselhável por ter um custo financeiro inferior.

Foi realizado um rastreio para identificar quais grupos possuíam porções de intervalos inválidos para processar e, assim, foi identificado que com o LW igual a 48 são 12 dos 28 grupos que possuem porções de intervalo inválido. Com o LW igual a 96 são seis grupos e com LW igual a 192 três grupos. Logo, com o LW igual a 48 todos os SMPs estarão com WIs, porém com apenas um por SPs não utilizando o recurso de *pipeline* dos SPs. A atribuição do valor 96 para LW implica em atribuir 2 WIs por SPs utilizando o recurso do *pipeline*, porém existem neste cenário apenas oito grupos com porções de intervalos a serem avaliados por mais de uma iteração. Assim, na primeira atribuição de trabalho para SMPs todos estarão processando por mais de uma iteração e na segunda atribuição apenas um SMP dos sete estarão processando por mais de uma iteração. Com o LW igual a 192 cada SP possuirá quatro WIs para processar por SMPs, porém apenas cinco dos SMPs estarão processando porções de intervalos que irão gerar mais de uma iteração do algoritmo. É válido ressaltar que cada grupo está bem segmentado, ou seja, grupos com intervalos inválidos que finalizam na primeira iteração estão no mesmo grupo e WIs que irão processar a curva estão no mesmo grupo. Logo, WIs que irão finalizar com poucas iterações não estão no mesmo grupo. Isto só foi possível pela divisão do intervalo [-4,8] entre os 1344 WIs, o que não ocorreu com o intervalo [-4,4].

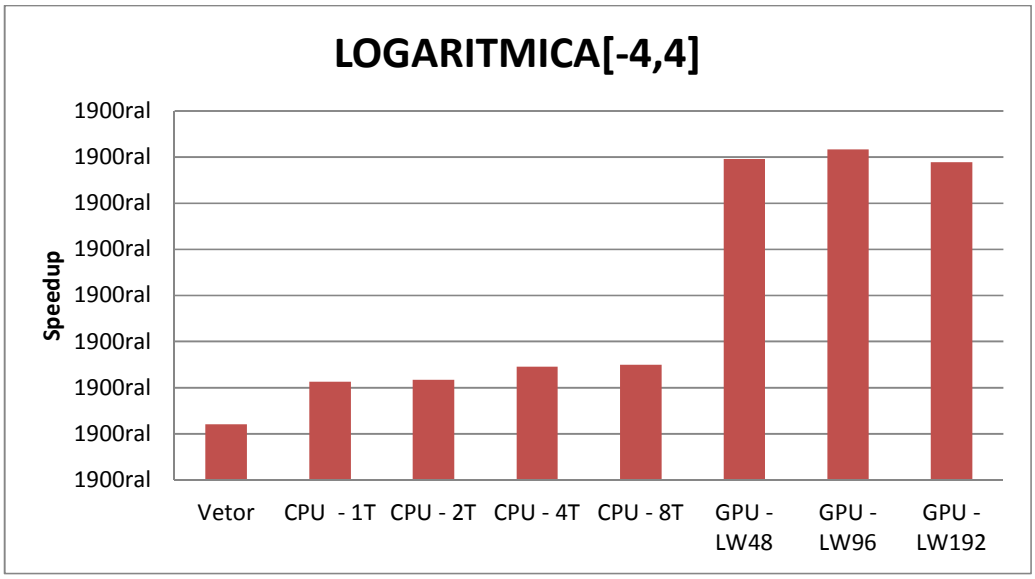
Considerando o intervalo [-4,4] e [-4,8], para funções com as características iguais da Exponencial utilizada, o ideal é utilizar um intervalo inicial de entrada que irá segmentar o espaço de busca de forma que os grupos fiquem divididos em quem possui a função para processar e quem não possui, conforme o intervalo [-4,8]. Em CPU, como o máximo de WIs que irão avaliar os intervalos onde está à função são 5, ocorre também esta divisão mencionada, onde, os WIs com índice 2 e 3 irão processar a curva dividindo o trabalho em uma frequência de 2,7 GHz.

Para o intervalo [0,4], o ideal é utilizar a GPU, uma vez que ao dividir para todos os WIs ocorre um maior balanceamento da carga, já que a curva é a parte de maior processamento. Neste caso, executar com um maior número de grupos eleva o desempenho, por evitar que WIs com poucas iterações estejam em grupos com WIs com mais iterações.

Apenas é possível concluir a influência do intervalo no tempo da aplicação, por a função exponencial demandar um nível de processamento que o tempo de execução do *Kernel* tenha maior peso no tempo final da aplicação, que o tempo de preparação do dispositivo.

Em sequência será apresentado o gráfico de ganho para a função logarítmica.

Figura 33. Resultados de *Speedup* para função logarítmica [-4,4].



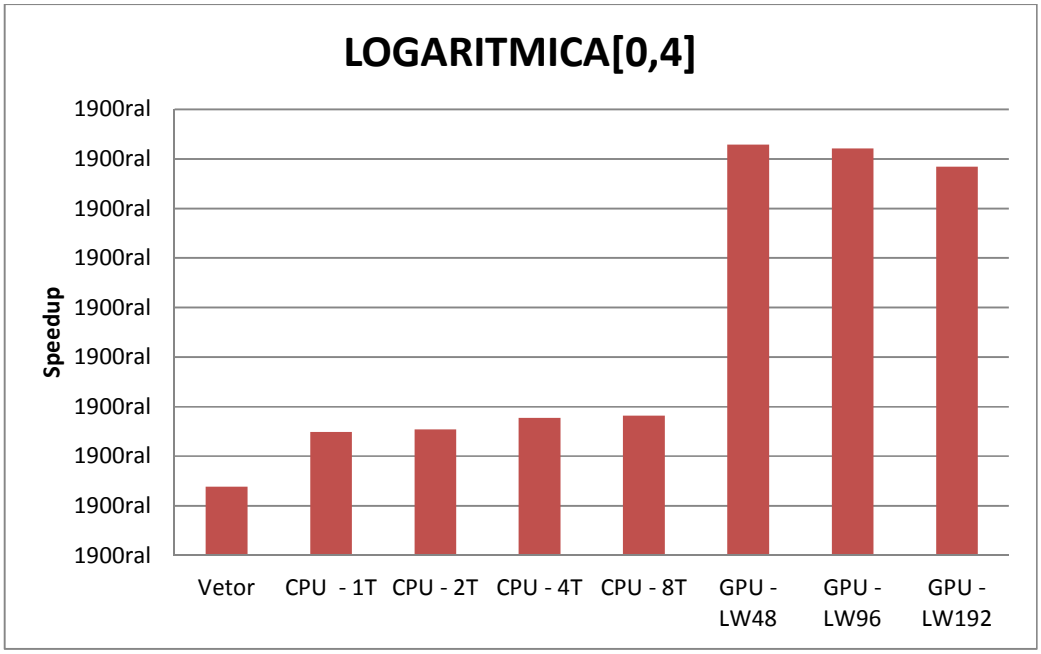
Na função logarítmica há ganho apenas na alteração da estrutura de dados, pois o ganho com a utilização do SIVIA sequencial com o vetor alocado é de 6 vezes. A utilização do PSIVIA-HP em CPU com apenas 1 WI levou a um ganho de 10 vezes, considerando que é uma execução sequencial e há o tempo do *Host*.

Para este intervalo e função, a melhora do tempo de *Kernel* em CPU não contribuiu para a melhora do tempo da aplicação. Isto é comprovado com o aumento do nível de paralelismo de 4 para 8 WIs, onde, o tempo de *Kernel* com 4 WIs foi de 0,694 segundos e o tempo com 8 WIs 0,439 segundos, sendo que os níveis de ganho da aplicação permaneceram os mesmos. Não houve mudança nos níveis de ganho, pois a diferença de 0,255 segundos foi suprimida pelo tempo do *Host* que é sequencial e sofre com escalonamento de processos do S.O.

Com a utilização da GPU o ganho máximo alcançado foi de 35 vezes com o LW igual a 96, superior ao da CPU que alcançou ganho máximo de 12 vezes. Este ganho obtido com a GPU superior ao da CPU é pela ocupação de todos os SMPs, pois dos 14 grupos, metade finaliza na primeira iteração e a outra metade processa a curva. Este nível de ganho com a GPU também só foi possível, pelo tempo de preparação do dispositivo ser inferior ao da CPU, como já mencionado.

. Em seguida serão apresentados os resultados de desempenho para o intervalo [0,4].

Figura 34. Resultados de *Speedup* para função logarítmica [0,4].

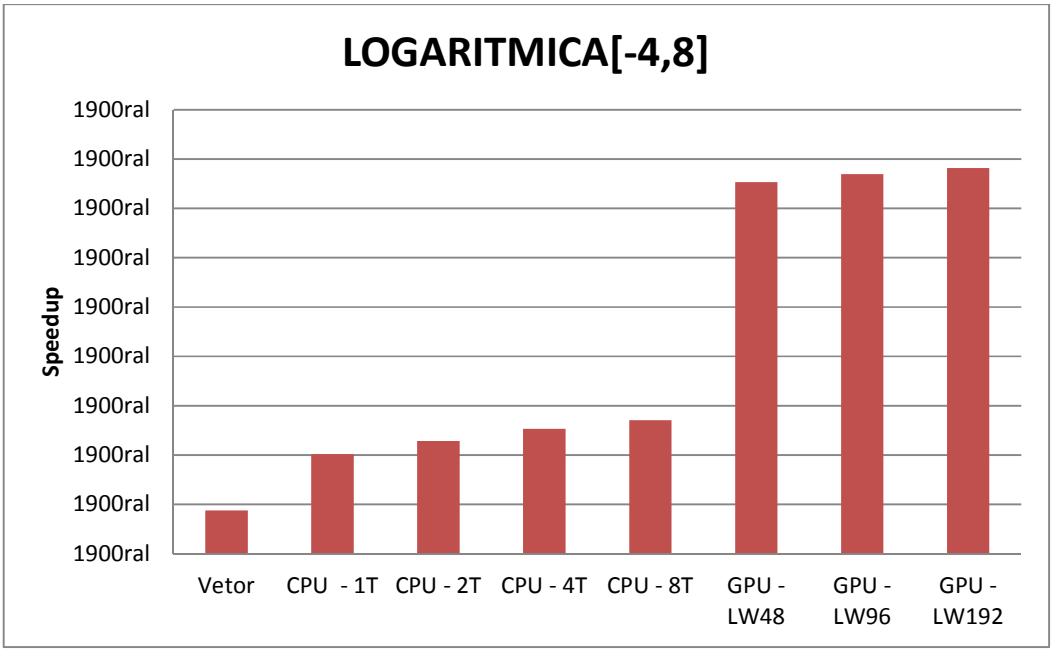


A primeira análise sobre o desempenho da função logarítmica para o intervalo [0,4] é que a aplicação não foi escalável em CPU, já que não houve alteração no desempenho com o aumento do nível de paralelismo de 1 WI para 2 WIs e de 4 WIs para 8 WIs. Conforme apresentado na seção 5.1.1, o aumento de WIs melhorou o tempo médio de execução do *Kernel*, porém a proporção média do tempo do *Kernel* em relação ao tempo da aplicação para esta função foi de 27,4% com 1 WI e 18,1% com 2 WIs. Sendo assim, a melhora do tempo do *Kernel* não contribuiu em ganho no tempo da aplicação. Esta análise também se mantém quando se eleva de 4 para 8 WIs. Porém, mesmo sem escalabilidade a execução em CPU obteve ganho máximo de 14 vezes em relação à lista encadeada.

O maior nível de ganho para este intervalo foi na execução em GPU com LW igual a 48 e 96, que alcançou ganho de 41 vezes. Novamente, apesar do tempo de *Kernel* em GPU ter sido superior ao da CPU, o tempo de preparação do dispositivo em GPU é menor.

Em sequência será apresentado o gráfico com o desempenho do intervalo [-4,8], que possui o maior custo de processamento para esta função, pois a curva é maior como se pode comprovar na figura 34.

Figura 35. Resultados de *Speedup* para função logarítmica [-4,8].



O PSIVIA-HP em CPU é escalável para este intervalo e função, uma vez que há ganho com o aumento de WIs. A aplicação apresentou o mesmo comportamento de escalabilidade na análise do tempo do *Kernel*, já que a proporção do tempo médio de execução do *Kernel* em CPU em relação ao tempo médio total da aplicação foi de 58,2% com 1 WI e 16,8% com 8 WIs. Logo, o tempo do *Kernel* exerceu maior influência no tempo final da aplicação.

Com a utilização do PSIVIA-HP, o ganho máximo obtido em CPU foi de 13 vezes e o ganho máximo da GPU foi de 39 vezes. Pois, com a maior demanda de processamento a proporção média do tempo do *Kernel* em relação à aplicação em GPU foi de 47,5%.

O maior ganho da GPU foi com LW igual a 192, porém o melhor tempo médio de execução do *Kernel*, conforme a figura 24 foi com LW igual a 96. Isto significa que o tempo do *Host* para as execuções com LW igual 96 foi maior do que para o LW igual a 192.

O nível de ganho da GPU superior ao da CPU, neste caso, é devido ao menor tempo médio de execução do *Kernel* e do menor tempo médio do *Host*, sendo que, apenas para este intervalo e função o tempo médio de execução do *Kernel* em GPU foi inferior ao da CPU.

Os resultados apresentados nesta seção demonstram que:

- Apenas modificar a estrutura de dados de lista encadeada para um vetor previamente alocado, já eleva o desempenho por diminuir a quantidade de acessos obrigatórios à memória que a lista encadeada impõe;

- A utilização do OpenCL sem nenhum nível de paralelismo também elevou o desempenho, sendo superior às outras abordagens sequenciais implementadas;

- Quando aplicado níveis de paralelismo no PSIVIA-HP em CPU, houve aumento de ganho em relação aos códigos sequencias e em média, aumento do ganho em relação aos níveis de paralelismo inferiores.

- Os ganhos máximos do PSIVIA-HP em GPU foram superiores a CPU na maioria das execuções. Exceto para a função exponencial no intervalo [-4,4] em que o desempenho foi igual, e para o intervalo [-4,8] também da função exponencial em que o desempenho foi inferior.

- O módulo de distribuição de carga foi projetado para distribuir o eixo x do espaço de busca de forma uniforme para todos os WIs. Porém, os resultados demonstraram que apenas esta distribuição não é a ideal, pois há WIs que nas primeiras iterações já finalizam sua execução por processarem porções de intervalos que já são classificados como sendo parte ou não da função, levando a ociosidade de EPs.

5.3 Análise de preparação do dispositivo

Como constatado na seção anterior, há uma proporção significativa no tempo do *Host* para preparar o dispositivo em relação ao tempo de execução do *Kernel*. Sendo assim, esta seção é destinada a identificar quais são as porções do código no *Host,* para preparar o dispositivo, que demandam maior quantidade de tempo, já que há diferença nos tempos de preparação entre GPU e CPU. Para realizar esta avaliação, foram selecionados os experimentos com melhor desempenho em cada dispositivo (em negrito na tabela 12) e a contraparte do melhor experimento no outro dispositivo.

Tabela 14. Escolha dos experimentos de verificação da preparação do dispositivo.

	Experimento 1		Experimento 2	
Função:	Polinomial		Exponencial	
Intervalo:	[0,4]		[-4,8]	
Dispositivo:	**GPU**	CPU	GPU	**CPU**
GW:	**1344**	8	1344	**8**
LW:	**48**	1	96	**1**
Tempo Total em (s):	**2,106**	4,401	4,873	**4,797**
Tempo *Kernel* em (s):	**0,942**	0,325	3,701	**0,844**

Com a seleção dos resultados apresentados na tabela 12, foram realizados experimentos para identificar as funções e conjunto de ações com maior custo para execução do PSIVIA-HP. Em seguida são descritas as rotinas e o conjunto de ações levantadas nos resultados dos experimentos:

- Obter Plataforma: Responsável por identificar a plataforma OpenCL do fabricante do dispositivo;
- Criação do Contexto: Criado por um ou mais dispositivos é utilizado em tempo de execução para gerenciar lista de comandos, memória e *Kernels* (KHRONOS, 2013);
- Construir Programa: Compila e liga o *Kernel* de um arquivo fonte ou binário (KHRONOS, 2013);
- Tempo de *Kernel*: Tempo de execução do *Kernel*;
- Mapeamento do Buffer de Resultado: Mapear o *buffer* no dispositivo que contém o conjunto de intervalos que pertencem à função encontrados na execução do *Kernel* do PSIVIA-HP.

A figura 35 apresenta os resultados de proporção de cada item descrito, em relação ao tempo total de execução em GPU e CPU, para a função polinomial e o intervalo [0,4].

Figura 36. Proporção de cada função em relação ao tempo total em GPU para a função polinomial e intervalo [0,4].

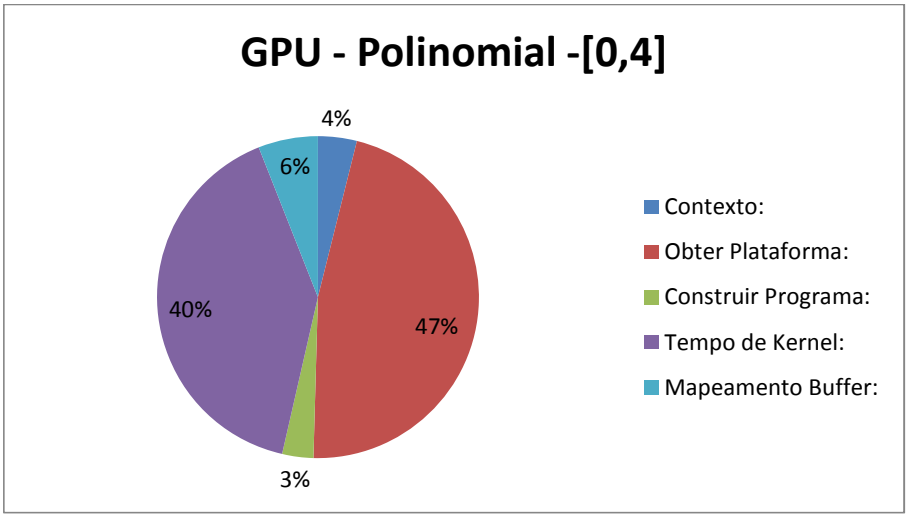
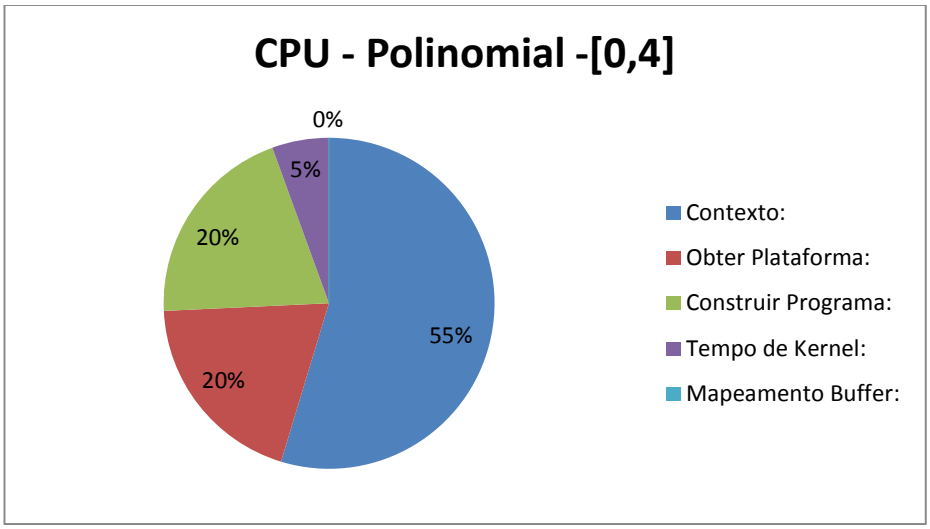


Figura 37. Proporção de cada função em relação ao tempo total em CPU para a função polinomial e intervalo [0,4].



Para ambos os dispositivos utilizados nesta dissertação o tempo para obter a plataforma é expressivo em relação ao tempo total da aplicação. Sendo com custo maior na GPU, uma vez que, para obter as informações necessárias, deve-se comunicar com a placa através do barramento PCI-EXPRESS, que não é o caso da CPU.

O tempo de mapeamento do *buffer* de resposta na GPU, apesar de ser apenas 6%, deve ser considerado como custo, já que o custo em CPU é zero. Este custo é gerado pela leitura da memória global da GPU e a latência do trafego através do barramento PCI-EXPRESS para a memória principal da CPU. Não há custo em CPU, pois é realizada apenas uma atribuição de ponteiros na memória primária.

Já a construção do programa em CPU obteve um custo maior, provavelmente devido ao fato que a CPU é um processador de propósito geral e a tradução para linguagem de máquina gere um maior número de instruções. Porém, é válido ressaltar que isto é uma hipótese, já que não há como investigar o desempenho e a arquitetura do compilador OpenCL da CPU e da GPU utilizada.

A proporção do tempo de execução do *Kernel* é maior em GPU pelas razões citadas na seção 4.2.1. e pelo fato da frequência de cada EP na GPU ser menor do que em CPU.

Houve maior custo na criação do contexto em CPU do que em GPU, já que em CPU o custo foi de 55% e em GPU apenas 4% do tempo total da aplicação. Não é possível determinar a razão desta diferença sem conhecimento da implementação do compilador, dos *drivers* utilizados e do SDK (*Software Development Kit*).

É válido ressaltar que excetuando o tempo do *Kernel*, a princípio todas as demais rotinas são executadas sequencialmente no *Host*. O termo a princípio é utilizado, pois, alguma

otimização paralela pode ter sido implementada nas rotinas, porém, não é possível ter acesso às implementações dessas funções.

Em seguida são apresentados os gráficos com os resultados do experimento 2.

Figura 38. Proporção de cada função em relação ao tempo total em GPU para a função exponencial e intervalo [-4,8].

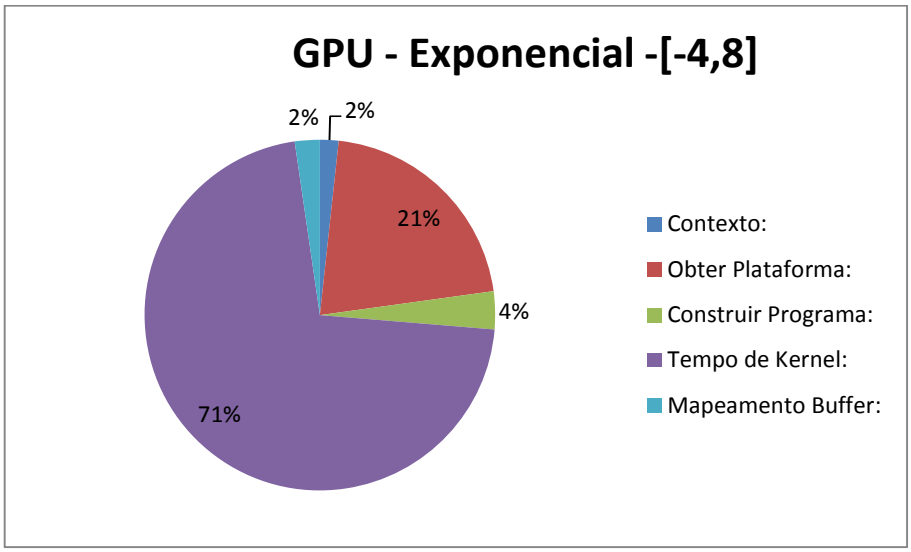
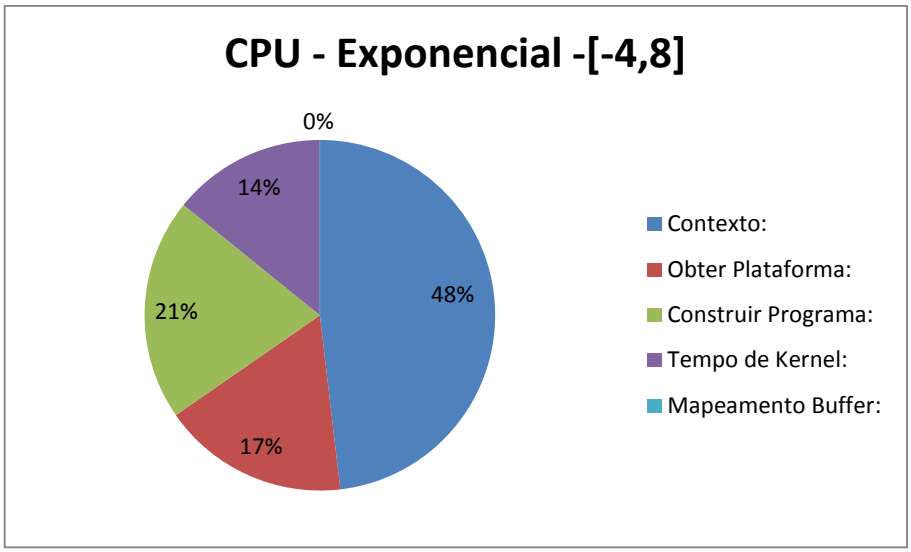


Figura 39. Proporção de cada função em relação ao tempo total em CPU para a função exponencial e intervalo [-4,8].



No experimento 2, a proporção do tempo de execução do *Kernel* foi superior para GPU e CPU, já que a função exponencial no intervalo [-4,8] demanda maior processamento. Vale mencionar novamente que, exceto o *Kernel* que é executado na GPU as demais rotinas são

executadas no *Host* (CPU), logo, quanto maior a demanda de processamento de uma função e intervalo, a proporção do tempo de *Kernel* em relação à aplicação irá crescer.

O tempo de mapeamento do *Buffer* de resultado continua sendo zero para CPU, pela mesma razão citada para a função polinomial. Não considerando o tempo de mapeamento do *Buffer* de resposta, a proporção do tempo de execução do *Kernel* foi menor do que as proporções de obter a plataforma, criar o contexto e construir o programa em CPU. O tempo de preparação médio foi o mesmo para as duas funções para ambas as plataformas, sendo 3,99 segundos em média para CPU e 1,45 segundos em média para GPU. Sendo assim, a utilização do OpenCL só é efetiva para cargas de trabalho que demandem processamento suficiente que valha o custo de preparação do dispositivo, frente ao tempo de execução de abordagens sequenciais.

6. CONCLUSÃO

Este capítulo é destinado a discutir os principais resultados, as principais contribuições e possíveis trabalhos futuros sobre esta dissertação.

6.1 Discussão dos Resultados

O escopo desta pesquisa foi definido em melhorar o desempenho do tempo de execução do algoritmo SIVIA para uma instância do problema. Para tal, foi utilizado o OpenCL que é um modelo de programação paralela que permite a execução de códigos em plataformas paralelas heterogêneas. Apesar de o OpenCL ser um modelo de programação paralela, a forma de paralelização do algoritmo é de responsabilidade do desenvolvedor. Sendo assim, para não ter que esperar um número de iterações que gere uma quantidade de intervalos que possam ser distribuídos para cada EP, foi determinada como hipótese a divisão uniforme do eixo do x do intervalo de entrada. Logo, na primeira iteração todos os EPs irão receber uma porção do intervalo inicial e podem iniciar o processamento.

Com a análise dos resultados, apresentada no capítulo 5, é possível concluir que o PSIVIA- HP possui um desempenho superior às implementações sequencias, independente do nível de paralelismo empregado, já que as execuções com apenas 1 WI (sequencial) foram superiores a versões sequenciais. O comportamento do *Kernel* em CPU, em relação ao aumento do nível de paralelismo mostrou que funções que possuem maior extensão no eixo x (como a exponencial) são mais escaláveis. Porém, as funções que não possuem essa característica são mais suscetíveis a ociosidades de EPs por WIs finalizarem na primeira iteração, não melhorando o desempenho após determinado nível de paralelismo.

O desempenho do *Kernel* em GPU, a diferença se apresentou maior em relação ao intervalo de entrada do que na variação do LW. Mas, o que é relativo à variação do LW, os grupos de tamanho igual à quantidade de EPs em um SMP (48 para GPU utilizada), foram que alcançaram melhor desempenho, excetuando apenas os intervalos [-4,4] da função exponencial e o intervalo [-4,8] da função logarítmica. Com o valor de LW menor, mais grupos são criados e assim há maior precisão na divisão de WIs com porções do intervalo que

não pertencem à função, dos WIs com porções de intervalos que pertencem à função. Essa precisão permite que, os grupos com WIs que finalizam nas primeiras iterações sejam escalonados, deixando todos os SMPs com grupos de WIs que tenham porções da função para processar.

Diante das análises do comportamento do *Kernel* em GPU e CPU é possível concluir que a hipótese apresentada da divisão uniforme do eixo *x* entre os WIs, não se mostrou a ideal por permitir que EPs fiquem ociosos, prejudicando o nível de escalabilidade e o desempenho máximo que poderia ser alcançado.

Na avaliação do tempo total da aplicação, o PSIVIA-HP alcançou desempenho superior em GPU em relação a CPU, exceto para a função exponencial nos intervalos [-4,8] e [-4,4]. Sendo que, na função polinomial e logarítmica o nível de ganho foi mais expressivo. Apesar do tempo do *Kernel* em GPU ter sido superior ao da CPU, em virtude do grau de ociosidade e frequência menor dos EPs da GPU, o custo de preparação do dispositivo em OpenCL em GPU é inferior ao da CPU, sendo esta a razão do desempenho superior em GPU. Este fato é comprovado pelos resultados apresentados e analisados na seção 5.3, onde para a função exponencial o tempo do *Kernel* na GPU foi de 71% do tempo total da aplicação e da função polinomial 40% do tempo total. Porém, na CPU para os dois experimentos, os tempos do *Kernel* foram apenas de 20% e 21% do tempo total da aplicação. Diante do exposto também é possível concluir que utilizar intervalos reduzidos, como o intervalo [0,4], com menor quantidade de espaços que não pertencem à função, impossibilita que WIs finalizem nas primeiras iterações, diminuindo o grau de ociosidade. Porém, a melhora de utilizar o intervalo [0,4], por exemplo, foi no máximo de 20%. Ou seja, para a exatidão de 0,00001 utilizada, a alteração do *Kernel*, a modificação da hipótese ou a utilização de um intervalo que irá produzir um grau de ociosidade 0, não irá produzir um ganho expressivo em razão do custo de preparação do dispositivo.

O objetivo geral do trabalho foi alcançado, uma vez que foi desenvolvido o PSIVIA-HP, onde obteve ganhos na execução em GPU e CPU em relação as implementações sequenciais, mesmo que a hipótese de balanceamento de carga não tenha sido a ideal. Os objetivos específicos também foram totalmente alcançados, já que a biblioteca intervalar em C99 foi implementada e validada, podendo ser utilizada para quaisquer aplicações de matemática intervalar em C e em OpenCL. A avaliação e análise também foram realizadas para compreensão do comportamento do PSIVIA-HP nos níveis do *Kernel* e da aplicação, sendo identificados os aspectos que propiciam ganho e os que impedem de alcançar níveis maiores de ganho.

6.2 Contribuições

Como contribuições desta dissertação ficam:

- O PSIVIA-HP que pode ser utilizado para solucionar qualquer problema intervalar N-DIMENSIONAL em qualquer plataforma suportada pelo OpenCL;
- A biblioteca intervalar em C99 voltada para OpenCL, podendo ser utilizada por qualquer desenvolvedor que queira trabalhar com os operadores intervalares em aplicações OpenCL ou apenas na linguagem C;
- Um algoritmo de divisão de carga para a utilização em aplicações que utilizem o mesmo modo atribuição de carga de trabalho através da divisão do eixo *x*;
- Uma análise de desempenho e escalabilidade da execução do *Kernel* em CPU e de adequação dos WIs para GPU;
- Análise de viabilidade da utilização do OpenCL, já que o tempo de preparação do dispositivo é fixo independente da carga de trabalho. É válido ressaltar que este tempo de preparação pode ser maior, pois só foram passados 21504 bytes (intervalo de *x* e *y* para os 1344 WIs) para a memória global da GPU. Caso fossem mais dados, como outras aplicações, o tempo de transferência da memória primária para a memória global da GPU iria aumentar o tempo final;
- Uma versão paralela do SIVIA proposto por Joulin (1992), em OpenCL, permitindo a execução em plataformas heterogêneas diferentemente da versão *multhreading* não descrita por Drevelle e Bonnifait (2013);
- Uma avaliação do desempenho global da aplicação incluindo o tempo de preparação de dispositivo, uma vez que apenas o tempo do *Kernel* foi avaliado por Fraire, Ferreyra e Marques (2013);
- Uma análise de desempenho da utilização do PSIVIA-HP em relação a métodos de programação tradicionais na área de computação, lista encadeada e vetor alocado;
- Como citado ficam também, o algoritmo SIVIA implementado utilizando a lista encadeada e o vetor alocado, baseados na biblioteca intervalar.

6.3 Trabalhos Futuros

Como trabalhos futuros desta dissertação ficam as possibilidades:

- Um novo método de divisão de carga de trabalho que propicie um melhor balanceamento, diminuindo o grau de ociosidade dos EPs;
- Um novo método de alocar os intervalos na memória *Global* para economia de recursos;
- Executar o PSIVIA-HP em CPUs e GPUs de diferentes modelos e fabricantes;
- Realizar experimentos com outros tipos de funções;
- Realizar experimentos quando há mais de uma instância do problema para ser solucionada;
- Executar os experimentos em CPU e em GPU simultaneamente;
- Executar o PSIVIA-HP em FPGAs;
- Realizar experimentos com funções multiobjetivos, uma vez que o PSIVIA-HP suporta esse tipo de execuções;
- Realizar experimentos para problemas que utilizem o espaço de busca tridimensional;
- A elaboração de um método que diminua o grau de redundância dos intervalos encontrados;
- A codificação do PSIVIA-HP para trabalhar com MPI.
- Utilizar o PSIVIA-HP para solucionar problemas reais de engenharia.

REFERÊNCIAS

Advanced Micro Devices, "**Programming Guide – ATI Stream Computing OpenCL**", AMD, Sunnyvalle, 2010.

CHEND, D., e SINGH, D., **Fractal Video Compression in OpenCL: An Evaluation of CPUs, GPUs, and FPGAs as Acceleration Platforms**, 18th Design Automation Conference (ASP-DAC), Asia and South Pacific, 2013.

DREVELLE, V., e BONNIFAIT, P., **Localization Confidence Domains via Set Inversion on Short-Term Trajectory**, IEEE Transactions on Robotics, 2013.

EL-REWINI, H., ABD-EL-BARR, M., **Advanced Computer Architecture and Parallel Processing**, Wiley-Interscience, e-book, pp2-15, 2005.

FLYNN, M, **Very High-Speed Computing Systems**, Proceedings of the IEEE, Vol.54, No. 12, pp. 1901-1909, Dezembro, 1966.

FRAIRE, J., FERREYRA, P., MARQUES, C., **OpenCL Overview, Implementation, and Performance Comparison,** IEEE LATIN AMERICA TRANSACTIONS, Vol. 11, No. 1, 2013.

HELD, J., BAUTISTA, J. e KOEHL, S., **From a Few Cores to Many: A Tera-scale Computing Research Overview,** White Paper Research Intel, pp. 2-5, 2006.

HENESSY, J. e PATTERSON, D., **Organização e Projeto de Computadores – Interface Hardware/Software**, Elsevier, Vol. 3, 2005.

JOULIN, L., WALTER, E., **Set Inversion via Interval Analysis for Nonlinear Bounded-error Estimation,** Automatica, Vol. 29, No. 4, pp. 1053-1064, 1993.

INTEL, **GPGPU Computing Horizons: Developing and Deploying for Microsoft Windows**, Microsoft HPC Whitepaper, pp. 4-7, 2010.

JAULIN, L., **Interval Contractors and Their Applications**, Université Polytechnique de Catalogne, 2004.

Khronos Group, **OpenCL**, disponívem em: http://www.khronos.org/, acessado: 22 de julho de 2013.

MAZEIKA, A., JAULIN, L., e OSSWALD, C., **A new approach for computing with fuzzy sets using interval analysis**, 10Th International Conference Information Fusion, 9-12 July 2007.

MOORE, G., **Cramming more components onto integrated circuits**, Eletronics, Vol. 38, no. 8, 1965.

MPI FORUM, **MPI: A message-passing interface standard**, The International Journal of Supercomputing Applications and High Performance Computing 8, p.159–416, 1994.

Nvidia, **CUDA**, disponível em: http://www.nvidia.com.br/object/cuda_home_new.html, acessado em: 22 de julho de 2013.

OPENMP, disponível em www.openmp.org, acessado dia 22 de julho de 2013.

ROCHA, H., SILVEIRA, L., MARTINS, C., **Avaliação de desempenho em GP-GPU utilizando a arquitetura CUDA,** Concurso de Trabalhos de Iniciação Científica, *Workshop* de Sistemas Computacionais de Alto Desempenho, Campo Grande, Brasil, 2008.

TANENBAUM, A., **Sistemas Operacionais Modernos**, Prentice Hall, Vol. 3, 2010.

TANENBAUM, A., **Organização Estruturada de Computadores**, Pearson Prentice Hall, Vol.5, São Paulo, 2007.