Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

# Técnicas de Evolução Gramatical Massivamente Paralela com OpenCL adotando Interpretação ou Compilação dos Modelos Candidatos

**Igor Lucas de Souza Russo**

JUIZ DE FORA
DEZEMBRO, 2014

# Técnicas de Evolução Gramatical Massivamente Paralela com OpenCL adotando Interpretação ou Compilação dos Modelos Candidatos

Igor Lucas de Souza Russo

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Heder Soares Bernardino

JUIZ DE FORA

DEZEMBRO, 2014

# Técnicas de Evolução Gramatical Massivamente Paralela com OpenCL adotando Interpretação ou Compilação dos Modelos Candidatos

Igor Lucas de Souza Russo

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

---

Heder Soares Bernardino
Prof. Doutor em Modelagem Computacional

---

Leonardo Goliatt da Fonseca
Prof. Doutor em Modelagem Computacional

---

Bernardo Martins Rocha
Prof. Doutor em Modelagem Computacional

---

Luciana Conceição Dias Campos
Prof[a]. Doutora em Engenharia Elétrica

JUIZ DE FORA

02 DE DEZEMBRO, 2014

*Este trabalho é dedicado aos meus pais, Aristea e Elcyr.*

# Resumo

A Evolução Gramatical (EG) é uma metaheurística bioinspirada capaz de evoluir programas em linguagem arbitrária através de uma gramática formal. Dentre suas principais aplicações, pode-se destacar a inferência automática de modelos a partir de dados. Assim como outras técnicas de programação genética, a EG possui alto custo computacional. Entretanto, o algoritmo possui etapas que podem ser computadas independentemente, possibilitando o uso de computação paralela para redução do tempo de execução, viabilizando sua aplicação a problemas maiores e mais complexos. Neste trabalho são estudados e propostos modelos de computação massivamente paralela para a EG utilizando OpenCL, um framework para a criação de algoritmos paralelos em ambientes heterogêneos. Além disso, na EG os programas candidatos podem ser avaliados por interpretação, ou compilação e execução. As duas abordagens foram comparadas aqui em implementações massivamente paralelas , e, para problemas com grandes conjuntos de treinamento, observou-se que a abordagem de compilação é vantajosa em relação à de interpretação. Em problemas de regressão simbólica, experimentos computacionais foram realizados para analisar o desempenho da técnica utilizando GPUs (Unidades de Processamento Gráfico), em relação à execução sequencial e ganhos de desempenho de até 63.3× foram observados, ao executar em paralelo todos os passos da técnica.

**Palavras-chave:** Evolução gramatical, programação genética, paralelismo, OpenCL

# Abstract

Grammatical Evolution (GE) is a bioinspired metaheuristic capable of evolving programs in an arbitrary language through a formal grammar, typically context-free. Among the major applications of the technique, can be highlighted the automatic inference of models from data. As well as other genetic programming techniques, GE has a high computational cost. However, the algorithm has steps that can be computed independently, enabling the use of parallel computing to reduce the execution time, making possible its application to larger and more complex problems. In this work models of massively parallel computation for GE are studied and proposed using OpenCL, a framework for the creation of parallel algorithms in heterogeneous computing environments. Moreover, in GE candidate programs can be evaluated by interpretation or compilation and execution. Both are compared here in massively parallel implementations, and for problems involving large training sets, it was observed that the compilation approach is advantageous compared to the use of an interpreter. In symbolic regression problems, computational experiments were performed to analyze the performance technique using GPUs (Graphics Processing Units), in relation to the sequential execution in CPUs (Central Processing Units), and performance gains of up to 63.3× were observed, when all steps were performed in parallel.
**Keywords:** Grammatical evolution, genetic programming, parallel computing, OpenCL

# Agradecimentos

A Deus por ter me dado forças para enfrentar os desafios encontrados.

A minha amada Graziela, pelo carinho e compreensão, mesmo nos momentos mais difíceis.

Ao meu orientador, professor Heder Bernardino, pela oportunidade, amizade e apoio na elaboração deste trabalho.

Aos meus pais, irmãos, tio e avós pelo encorajamento e apoio constantes.

Aos companheiros e amigos de jornada: Bruno, Thiago, Marcelo e Daniel, pessoas com que tive o prazer de conviver durante os últimos quatro anos.

Aos professores do Departamento de Ciência da Computação pelos seus ensinamentos e a todos aqueles que, durante esses anos, contribuíram de algum modo para meu enriquecimento pessoal e profissional.

# Sumário

**Lista de Figuras** **6**

**Lista de Tabelas** **7**

**1 Introdução** **8**
1.1 Contextualização . . . . . 8
1.2 Objetivos . . . . . 11
1.3 Trabalhos relacionados . . . . . 11

**2 Evolução gramatical** **13**
2.1 Introdução . . . . . 13
2.2 Algoritmos Genéticos . . . . . 14
2.3 Programação genética . . . . . 16
2.4 Gramáticas formais . . . . . 18
2.5 Evolução gramatical . . . . . 20
2.5.1 Avaliação das soluções candidatas . . . . . 22

**3 Paralelismo** **26**
3.1 Introdução . . . . . 26
3.2 GPGPU . . . . . 29
3.3 OpenCL . . . . . 31

**4 Evolução Gramatical massivamente paralela** **35**
4.1 Paralelismo da etapa de avaliação . . . . . 35
4.1.1 Avaliação via interpretação dos modelos candidatos . . . . . 35
4.1.2 Avaliação via compilação dos modelos candidatos . . . . . 41
4.2 Paralelismo de todo o processo de busca . . . . . 44
4.2.1 Melhorias no algoritmo . . . . . 51

**5 Experimentos computacionais** **58**
5.1 Introdução . . . . . 58
5.2 Experimento I . . . . . 59
5.3 Experimento II . . . . . 63
5.4 Experimento III . . . . . 65
5.5 Experimento IV . . . . . 67
5.6 Discussões complementares . . . . . 69

**6 Conclusões e trabalhos futuros** **73**

**Referências Bibliográficas** **75**

# Lista de Figuras


1.1 Regressão simbólica da função $f(x) = 2x^3$ a partir de um conjunto de pontos. 8

2.1 Árvore para o programa que calcula o valor de $X^2 + Y$. . . . . 17
2.2 Exemplo de produção da expressão $X * Y$. . . . . 20
2.3 Algoritmo Genético. . . . . 21
2.4 Exemplo da primeira etapa do mapeamento, utilizando 8 bits para cada inteiro. . . . . 21
2.5 Exemplo contendo as duas etapas do mapeamento de uma solução candidata. . . . . 23

3.1 Taxonomia de Flynn. . . . . 28
3.2 Modelo de plataforma, adaptado de Khronos (2011). . . . . 32
3.3 Modelo de memória do OpenCL. . . . . 34

4.1 Paralelismo da etapa de avaliação, adaptado de Augusto e Barbosa (2013). 38
4.2 Exemplo de soma paralela, adaptado de (Pacheco, 2011). . . . . 40
4.3 Fluxograma de um AG com indicação da ocorrência de sincronismos. . . . 45

5.1 Tempo total de execução em função do número de registros (Russo et al., 2014b). . . . . 61
5.2 Tempo de execução da avaliação em função do número de registros (Russo et al., 2014b). . . . . 62
5.3 Tempo total de execução em função do número de registros (escala logarítmica). . . . . 62
5.4 Tempo de execução da avaliação em função do número de registros (escala logarítmica). . . . . 63
5.5 Gráfico do tempo de execução em função do número de variáveis para os casos de compilação e interpretação de modelos candidatos (Russo et al., 2014b). . . . . 65
5.6 Speedup em função do número de registros (Russo et al., 2014a). . . . . 66
5.7 Tempo total de execução em função do número de registros. . . . . 67
5.8 Speedup de acordo com a variação do número de pontos e do tamanho da população, considerando o tempo total de execução da GPU. . . . . 68
5.9 Speedup de acordo com a variação do número de pontos e do tamanho da população, considerando somente o tempo de processamento da GPU (sem *overhead*). . . . . 69

# Lista de Tabelas

2.1 Exemplo de mapeamento de vetor de inteiros em um programa. . . . . 22
2.2 Exemplos de expressões em notação pós-fixada. . . . . 24

5.1 Parâmetros para o Experimento I. . . . . 60
5.2 Parâmetros para o Experimento II. . . . . 64
5.3 Parâmetros para o Experimento IV. . . . . 67

# 1 Introdução

## 1.1 Contextualização

A construção de modelos matemáticos a partir de dados é um problema tratado pela área de identificação de sistemas e consiste na inferência de estruturas capazes de mapear entradas em saídas, de modo a representar o comportamento do fenômeno observado. É importante ressaltar que a inferência de modelos em forma simbólica é uma ferramenta que pode auxiliar especialistas na descoberta de conhecimento a partir de dados.

Uma das abordagens utilizadas para este problema é a identificação paramétrica, na qual a estrutura do modelo é inicialmente postulada e seus coeficientes numéricos são ajustados por alguma técnica (como o método dos mínimos quadrados, por exemplo). Nota-se que neste caso é necessária a atuação de um especialista para identificação da estrutura a ser inferida.

Pode-se pensar em automatizar a etapa criativa deste processo, inferindo não somente parâmetros numéricos, mas também a própria estrutura do modelo. Surge assim a **regressão simbólica**, cujo objetivo é a obtenção de um modelo simbólico que explique o comportamento de um fenômeno a partir de um conjunto de amostras. Na Figura 1.1 é ilustrado um exemplo de aplicação da regressão simbólica, no qual deve ser inferida a função $f(x) = 2x^3$.

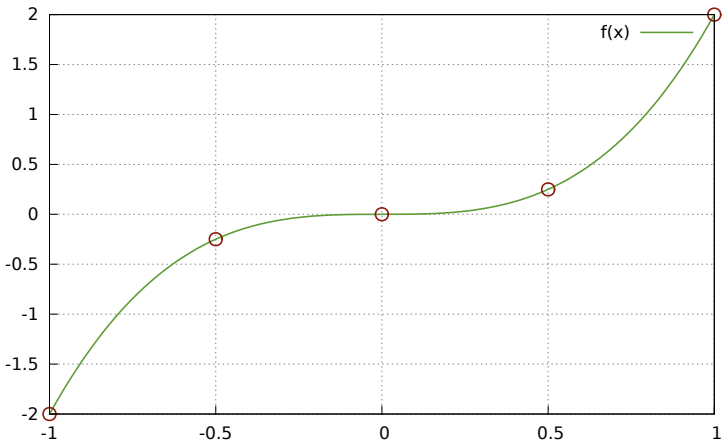



Figura 1.1: Regressão simbólica da função $f(x) = 2x^3$ a partir de um conjunto de pontos.

Segundo Langley (2002), alguns dos principais desafios da automação da descoberta científica são:

- As técnicas de descobrimento devem produzir saídas que sejam facilmente comunicáveis aos especialistas;
- Sistemas devem possibilitar a inserção de conhecimento prévio no processo de descoberta;
- As técnicas devem ser capazes de realizar a inferência a partir de pequenos conjuntos de dados;
- Sistemas de descobrimento devem propiciar interação com os especialistas.

Neste contexto, técnicas de Programação Genética (PG) têm sido amplamente utilizadas, dadas suas virtudes como robustez, exigência de pouco conhecimento prévio do domínio e, sobretudo, fornecimento de **soluções simbólicas**. A PG é uma metaheurística inspirada na teoria da evolução das espécies, que possui o ambicioso objetivo de evoluir automaticamente programas de computador (Koza, 1992). Na literatura pode-se encontrar exemplos de sucesso da PG na evolução de topologias em circuitos eletrônicos, controladores, antenas de transmissão, entre outros (Glover e Kochenberger, 2003).

Como indicado anteriormente, a possibilidade de introdução de conhecimento prévio no processo de descoberta é uma característica desejável. Considerando que na PG as soluções candidatas são programas de computador, pode-se pensar na utilização de **gramáticas formais** para restringir o espaço de busca.

Neste trabalho é utilizada a Evolução Gramatical (EG), um tipo de PG que é capaz de evoluir programas em linguagem arbitrária através de uma gramática formal (O'Neill and Ryan, 2001; O'Neill e Ryan, 2003). Além da vantagem evidente de introdução de conhecimento prévio, adequando-se o espaço de busca ao interesse do especialista, a utilização de uma gramática impede a geração de soluções candidatas inválidas.

Apesar de sua flexibilidade, a EG, assim como outras técnicas de PG, apresenta custo computacional elevado, limitando sua aplicação a problemas de grande porte. Este custo computacional concentra-se na etapa de avaliação de qualidade das soluções candi-

datas e é diretamente proporcional ao tamanho do conjunto de treinamento utilizado e à complexidade dos modelos de interesse.

Por outro lado, as técnicas de PG são naturalmente paralelas, uma vez que existem etapas do algoritmo que podem ser computadas de forma independente. Assim, a adoção de computação paralela pode ser uma estratégia para redução do tempo de execução destas técnicas, viabilizando sua aplicação a problemas de grande porte.

Nos últimos anos tem se destacado o GPGPU (do inglês *General-purpose computing on graphics processing units*), que consiste na utilização de Unidades de Processamento Gráfico (GPUs) para implementação de algoritmos de propósito geral, isto é, não necessariamente relacionados à computação gráfica. Um fator de incentivo à utilização das GPUs em pesquisas é sua boa relação de custo/benefício, uma vez que estes dispositivos geralmente possuem centenas ou milhares de elementos de processamento.

São propostas nesse trabalho diferentes formas de paralelizar a EG utilizando a arquitetura GPGPU. Para exploração deste poder computacional, utiliza-se aqui o OpenCL (*Open Computing Language*), um *framework* para criação de sistemas paralelos em ambientes heterogêneos (que podem ser compostos por CPUs e GPUs).

Usualmente, as implementações paralelas de PG realizam a avaliação dos modelos candidatos utilizando um interpretador. Considerando que os indivíduos aqui são programas de computador, gerados através de uma gramática, pode-se também avaliá-los através de sua compilação e execução. Enquanto a interpretação possui maior custo computacional, a compilação necessita de um passo adicional para transformação do código-fonte em código executável.

São propostos neste trabalho modelos de paralelismo da Evolução Gramatical utilizando tanto interpretação quanto compilação das soluções candidatas.

Experimentos computacionais foram realizados para (i) analisar os ganhos de desempenho obtidos pelas técnicas propostas com a utilização de GPUs, em relação à execução sequencial em CPU; e (ii) comparar o desempenho das estratégias de interpretação e compilação, em implementações massivamente paralelas.

## 1.2 Objetivos

A principal contribuição deste trabalho é a proposta e análise de formas de paralelismo da Evolução Gramatical utilizando computação massivamente paralela, de forma a viabilizar sua aplicação em problemas de grande porte. Inicialmente são estudados modelos em que somente a etapa de avaliação é realizada em paralelo. Por fim, são propostas estratégias para paralelização de todo o processo da EG. Experimentos são realizados para comparação dos desempenhos computacionais das técnicas propostas utilizando GPUs, em relação à execução sequencial em CPU.

Além disso, duas abordagens são consideradas para a avaliação dos modelos candidatos: uma que utiliza um interpretador e outra que compila o programa candidato para ser executado no dispositivo de processamento. Experimentos foram realizados para determinar se para grandes conjuntos de dados a estratégia de compilação dos indivíduos candidatos é vantajosa, em relação à utilização de um interpretador.

## 1.3 Trabalhos relacionados

O assunto tratado aqui ainda tem sido pouco explorado na literatura. Em pesquisas realizadas, Pospichal et al. (2011) foi o primeiro trabalho neste sentido, no qual foi proposta uma paralelização da EG utilizando CUDA, uma plataforma de computação paralela desenvolvida pela Nvidia que permite explorar o GPGPU em suas placas gráficas. No trabalho citado, os autores relataram um ganho de desempenho de até 39× em problemas de regressão simbólica.

Diferentemente do que foi feito por Pospichal et al. (2011), neste trabalho é utilizado OpenCL, um padrão aberto para criação de sistemas paralelos em ambientes heterogêneos. Apesar de haver poucos artigos que apresentam técnicas de PG utilizando OpenCL, este *framework* merece destaque por não ser proprietário e permitir a exploração de todo o poder computacional disponível no sistema, utilizando, por exemplo, GPUs e CPUs em conjunto.

Em (Augusto e Barbosa, 2013) foi proposta a paralelização da etapa de avaliação da PG tradicional utilizando OpenCL. Em tal trabalho, experimentos computacionais

mostraram um ganho de desempenho expressivo, de até 126× em problemas de regressão simbólica.

Quanto ao modelo de paralelismo, segundo Robilliard et al. (2008) existem duas abordagens de paralelismo da PG sob a arquitetura GPGPU:

1. de dados, onde um modelo é avaliado por vez, mas as avaliações são realizadas concomitantemente; e

2. de programas, em que vários modelos são avaliados paralelamente.

É adotado aqui um modelo híbrido baseado no trabalho de Augusto e Barbosa (2013), no qual as duas abordagens são utilizadas em conjunto.

Em relação à estratégia de avaliação das soluções candidatas, as propostas de PG paralela usualmente utilizam interpretadores (Juille e Pollack, 1996; Harding e Banzhaf, 2007; Langdon e Banzhaf, 2008; Augusto e Barbosa, 2013). A avaliação via interpretação mostrou-se eficiente do ponto de vista do custo computacional, mesmo considerando-se pequenos conjuntos de treinamento (Langdon e Banzhaf, 2008).

Por outro lado, Chitty (2007) propôs um modelo de paralelismo da PG no qual a avaliação dos indivíduos é realizada através de sua compilação. Naquele trabalho, que utilizou o *framework* OpenGL[1], foi adotado o paralelismo de dados. Em experimentos realizados utilizando problemas de regressão simbólica com 400 registros de treinamento, foi obtido um ganho médio de desempenho de aproximadamente 9×, em relação à execução sequencial da PG.

[1] http://www.opengl.org/

# 2 Evolução gramatical

## 2.1 Introdução

As metaheurísticas, em sua definição original, são metodologias que combinam procedimentos de busca local com estratégias de mais alto nível, de modo a criar um processo capaz de escapar de ótimos locais e realizar uma busca robusta no espaço de soluções de um problema (Glover e Kochenberger, 2003). Com o passar do tempo, esta definição passou a englobar todas as técnicas que utilizam estratégias para escapar de ótimos locais em espaços de busca complexos (Glover e Kochenberger, 2003).

Apesar de não haver garantias da obtenção da solução ótima para um problema, as metaheurísticas têm sido utilizadas com sucesso em diversas áreas, como engenharia e pesquisa operacional. De acordo com Talbi (2009), a utilização de metaheurísticas é indicada nos seguintes casos:

- Problemas de baixa complexidade (para os quais são conhecidos algoritmos de tempo polinomial), mas com instâncias muito grandes, tornando o tempo de resposta elevado.
- Problemas de baixa complexidade, porém com restrições de tempo para encontrar uma boa solução.
- Problemas NP-difíceis, para os quais não são conhecidos algoritmos polinomiais.
- Problemas em que o cálculo da medida de qualidade das soluções apresenta alto custo computacional.
- Problemas de otimização em que a medida de qualidade não é diretamente conhecida, mas sim calculada utilizando uma solução "caixa preta".

Uma outra vantagem das metaheurísticas sobre outros métodos é que elas geralmente requerem pouco conhecimento sobre o problema.

Grande parte das metaheurísticas é de inspiração natural, ou seja, seu funcionamento baseia-se em algum comportamento observado na natureza. Os algoritmos evolutivos pertencem a esta categoria e consistem em técnicas baseadas na teoria da evolução das espécies de Darwin. Na literatura pode-se encontrar vários outros exemplos de metaheurísticas bioinspiradas, como Otimização por Colônias de Formigas (Colorni et al., 1991), Otimização por Enxame de Partículas (Kennedy e Eberhart, 1995), Sistemas Imunológicos Artificiais (Farmer et al., 1986), entre outros.

No que se tem conhecimento, Fogel (1966) foi a primeira proposta de um algoritmo evolutivo presente na literatura. Naquele trabalho, uma técnica denominada programação evolucionista (do inglês *evolutionary programmning*) foi utilizada para desenvolver máquinas de estados finitos através de um processo simulado de mutação. Na mesma época, Ingo Rechenberg e Hans-Paul Schwefel aplicaram estratégias evolutivas (do inglês *evolutionary strategies*) para a solução de problemas com funções objetivo de valores reais (Rechenberg, 1973). Pouco tempo depois, os Algoritmos Genéticos (AGs) foram propostos por Holland (1975), introduzindo o cruzamento de indivíduos como parte do processo evolutivo.

## 2.2 Algoritmos Genéticos

Os AGs são métodos de busca estocásticos inspirados pela teoria da evolução das espécies e pela genética (Glover e Kochenberger, 2003). As soluções candidatas são representadas pelos indivíduos de uma população. Através de um processo iterativo de evolução simulada, os melhores indivíduos da população (mais aptos) são selecionados e passam por um processo de cruzamento (ou recombinação) e mutação, dando origem a uma nova população de soluções. O algoritmo é executado até que o critério de parada seja satisfeito. O Algoritmo 1 ilustra este processo.

**Algoritmo 1:** Algoritmo Genético

```
1  início
2      populacao_corrente ← InicializaPopulacao();
3      Avaliacao(populacao_corrente);
4      enquanto Critério de parada não for atingido faça
5          individuos_selecionados ← Selecao(populacao_corrente);
6          populacao_filhos ← Recombinacao(individuos_selecionados);
7          Mutacao(populacao_filhos);
8          Avaliacao(populacao_filhos);
9          populacao_corrente ← Substituicao(populacao_corrente, populacao_filhos);
10     fim enquanto
11 fim
```

Cada indivíduo possui um genótipo, que é a codificação de uma solução candidata em uma estrutura de dados. Na proposta original, a solução é codificada em um vetor de valores binários.

Além do genótipo, os indivíduos possuem também um fenótipo, que de maneira análoga ao que ocorre nos sistemas biológicos, representa as características do indivíduo.

Apesar da representação binária ser de fácil utilização e ter sido largamente empregada, outras foram propostas e recomendadas para determinadas situações, como a representação real, recomendada para problemas com parâmetros contínuos (Michalewicz, 1996) e a representação inteira, mais indicada para problemas que envolvam números inteiros (Soares, 1997).

A codificação/decodificação das soluções é dependente do problema e de suma importância para o sucesso do algoritmo, uma vez que define o espaço de busca. Além da codificação/decodificação, outra etapa dependente do problema é a função de avaliação, da qual a aptidão (do inglês *fitness*) de um indivíduo é derivada.

Após a avaliação dos indivíduos, é empregado um processo de seleção, no qual indivíduos com maior aptidão têm mais chances de serem escolhidos. A abordagem de seleção adotada aqui foi o **torneio**, no qual para cada indivíduo selecionado é promovida uma competição entre um número determinado de indivíduos tomados aleatoriamente, sendo escolhido aquele que possuir a maior aptidão.

Aos indivíduos selecionados são aplicados os operadores genéticos de recombinação e mutação, cada um com probabilidade definida por parâmetros do algoritmo.

O operador de recombinação realiza o cruzamento entre os genótipos dos indivíduos selecionados na etapa anterior. Geralmente são selecionados 2 indivíduos e é aplicado o operador de recombinação, dando origem a 2 novos indivíduos. No **cruzamento de um ponto**, operador de recombinação adotado aqui, o genótipo é dividido em uma posição selecionada aleatoriamente. O primeiro filho recebe a primeira parte do genótipo do primeiro pai, e o restante é herdado do segundo. Com o segundo filho, ocorre o oposto. O processo se repete até que toda a nova população seja gerada (esquema geracional). Ao longo das iterações do algoritmo, a recombinação tem a tendência de disseminar as características dos indivíduos mais aptos.

O operador de mutação é aplicado sobre os indivíduos gerados pelo cruzamento, e consiste em perturbar o genótipo. O operador adotado aqui para mutação consiste em percorrer todas as posições do vetor do genótipo e realizar a inversão de seu valor de acordo com a probabilidade definida por um parâmetro do algoritmo chamado **taxa de mutação**. Ao longo das gerações, este operador permite introduzir novas características nos indivíduos, aumentando a diversidade da população.

Após a criação de uma nova população, esta tem sua aptidão calculada e ocorre a substituição da população anterior. A política de substituição empregada aqui é a **geracional**, na qual a nova população substitui a geração anterior, passando a ser a população corrente. Além disso, geralmente utiliza-se o **elitismo**, que consiste em manter na população as melhores soluções candidatas geradas na última iteração.

Diferentes critérios de parada podem ser utilizados, como número total de gerações ou um valor de aptidão a ser atingido. Enquanto este critério não for satisfeito, novas gerações são operadas a partir da seleção de indivíduos mais aptos, recombinação e mutação.

## 2.3 Programação genética

A programação genética (PG) é uma metaheurística evolucionista capaz de evoluir programas automaticamente, sem a necessidade de que o usuário conheça ou especifique a

estrutura da solução desejada (Poli et al., 2008). Segundo Glover e Kochenberger (2003), a PG é uma extensão dos AGs, na qual a população sendo evoluída é formada por programas. Na literatura pode-se encontrar exemplos de sucesso da PG na evolução de topologias em circuitos eletrônicos, controladores, antenas de transmissão, entre outros (Glover e Kochenberger, 2003).

A PG é uma técnica robusta que pode ser aplicada a vários tipos de problemas, como por exemplo regressão simbólica e classificação de dados. Para que a técnica possa ser aplicada a um problema, basta que este (i) possa ser expresso por um programa de computador; e (ii) possua uma função capaz de calcular a medida de qualidade de um indivíduo. A função de avaliação é dependente do problema e no caso da regressão simbólica consiste na medida da discrepância entre os valores das saídas dos registros de treinamento e aqueles obtidos a partir da execução dos modelos candidatos para as respectivas entradas.

As soluções são usualmente representadas por estruturas de árvore, nas quais as folhas contém os **terminais**, que são as as constantes e variáveis do programa. Os outros nós da árvore contém as **funções**, como operações lógicas e aritméticas, comandos condicionais, ou até mesmo sub-rotinas. A Figura 2.1 ilustra a representação em árvore para o programa que calcula o valor de $X^2 + Y$.

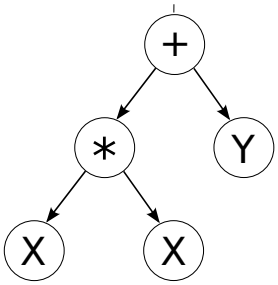



Figura 2.1: Árvore para o programa que calcula o valor de $X^2 + Y$.

Assim como nos AGs, na PG a exploração do espaço de busca se dá através da aplicação de operadores genéticos a indivíduos escolhidos mediante um processo inspirado na seleção natural, na qual os indivíduos mais aptos têm mais chances de sobrevivência.

Os operadores geralmente utilizados são o cruzamento e a mutação. No cruzamento, dois indivíduos selecionados (chamados pais) permutam sub-árvores para geração de dois novos indivíduos. A mutação de um indivíduo possui duas etapas: inicialmente

uma sub-árvore aleatoriamente escolhida é removida da árvore, e uma nova sub-árvore é gerada aleatoriamente e introduzida neste local.

Para que a PG seja utilizada de forma eficaz, os conjuntos de funções selecionados devem obedecer a uma propriedade conhecida por **fechamento**, sendo sua principal característica a consistência entre tipos de dados. A consistência entre tipos de dados é necessária porque os operadores genéticos da PG podem inserir/unir nós arbitrariamente. Desta forma, é preciso que todas as sub-árvores possam ser seguramente utilizadas como argumentos de qualquer função do conjunto de funções. Esta propriedade pode ser garantida fazendo com que todas as funções tenham o mesmo tipo de dados em suas entradas e saídas. Assim, um dado gerado por uma sub-árvore sempre poderá ser utilizado como parâmetro de entrada para o nó que a precede. Por exemplo, considerando um conjunto de funções aritméticas, como $(+, -, *, /)$, poderia ser definido que todas recebem como parâmetros dois valores reais e retornam um valor real.

Para evolução de programas mais complexos, pode-se perceber que a propriedade de fechamento pode ser um fator limitante. Neste contexto, foi proposta a programação genética fortemente tipada (Montana, 1995), na qual cada função tem seus tipos de entradas e saída, e todas as operações realizadas sobre as soluções candidatas (inicialização, cruzamento, mutação, etc.) são construídas de forma a respeitar as relações entre os tipos de dados.

## 2.4 Gramáticas formais

Uma característica desejável nas técnicas de programação genética é a possibilidade de se introduzir conhecimento prévio no processo evolutivo, restringindo o espaço de busca de acordo com o interesse do especialista. Na PG tradicional, entretanto, não é possível alcançar este objetivo.

Como os indivíduos gerados são programas de computador, pode-se pensar na utilização **gramáticas formais** para restringir o espaço de busca.

Gramática é um formalismo que permite a produção de cadeias de símbolos de uma linguagem (Chomsky, 2002). Neste contexto, pode ser utilizada aqui como uma técnica de restrição do espaço de busca, permitindo somente a geração de programas que

pertençam a uma linguagem definida.

Uma gramática livre de contexto G pode ser definida por uma tupla $(N, \Sigma, R, S)$, onde $N$ é um conjunto finito de não-terminais; $\Sigma$ é um conjunto finito de terminais da linguagem; $R$ é um conjunto de regras de produção; e $S$ é o símbolo inicial ($S \in N$).

A forma mais usual para representação das regras de produção em gramáticas livres de contexto é a Forma de Backus-Naur (BNF, do inglês *Backus-Naur Form*) (Knuth, 1964). Utilizando esta notação, cada regra de produção de $R$ é expressa como $<$ *não_terminal* $>::=$ *expressão*, onde <*não_terminal*> é um símbolo não-terminal e *expressão* pode ser tanto um não-terminal quanto um terminal. Se houver mais de uma possibilidade de derivação para um não-terminal, estas são separadas pelo símbolo "|".

Segue um Exemplo de gramática em BNF para expressões aritméticas:

$N =$ {<expr>, <op>, <op_u>, <var>, <const>}

$\Sigma =$ {+, −, ∗, /, 0, 1, 3.14, $X$, $Y$}

$S =$ <expr>

$R =$ <expr> ::= (<expr> <op> <expr>) | <op_u> (<expr>) | <var> | <const>

<op> ::= + | − | ∗ | / |

<op_u> ::= $\sqrt{}$ | *sen* | *cos*

<var> ::= $X$ | $Y$

<const> ::= 0 | 1 | 3.14

A produção de uma cadeia parte do símbolo inicial da gramática ($S$). Este não-terminal é então substituído arbitrariamente por uma de suas produções, definidas no conjunto $R$. O processo é repetido para cada não-terminal ($\in N$) presente na cadeia, até que existam somente símbolos terminais ($\in \Sigma$). A Figura 2.2 mostra um exemplo de produção utilizando a gramática exemplificada. Neste exemplo, partindo do não-terminal inicial, <expr>, foi selecionada a primeira produção, <expr><op><expr>, substituindo o símbolo inicial. No passo seguinte, os dois símbolos <expr> foram substituídos pelo não-terminal <var>, enquanto o símbolo <op> derivou a operação ∗. Finalmente, os símbolos não-terminais restantes foram substituídos pelas variáveis X e Y, respectivamente.

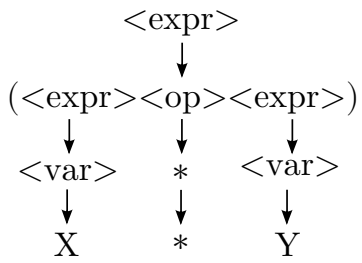



Figura 2.2: Exemplo de produção da expressão $X * Y$.

A utilização de gramáticas é uma forma eficiente de introduzir viés no processo de busca na PG, limitar o espaço de soluções, além de impedir a geração de programas inválidos (uma vez que estes devem obedecer à definição de uma linguagem formal).

Neste contexto, foi proposta a programação genética gramatical (PGG) (Whigham, 1995), variante da PG na qual é utilizada uma gramática formal para definir o espaço de possíveis programas. Na técnica proposta, diferentemente da PG tradicional, os nós da árvore contém os símbolos não-terminais e terminais de uma gramática. Esta abordagem permite maior flexibilidade na definição de restrições no espaço de busca. Além disso, como os programas obedecem à estrutura de uma gramática, é possível garantir a propriedade de fechamento. Entretanto, como desvantagem pode-se citar o aumento da complexidade de implementação e uso.

## 2.5 Evolução gramatical

Como discutido, a utilização de gramáticas é uma forma flexível para restringir o espaço de soluções na PG. Entretanto, ao invés de se introduzir a gramática diretamente na representação das soluções candidatas, como ocorre na PGG, pode-se pensar em separar a codificação dos programas da aplicação das restrições impostas pela gramática.

Neste contexto foi proposta a evolução gramatical (EG) (O'Neill and Ryan, 2001; **?**), um tipo de PG que utiliza uma gramática formal para a geração dos programas. Diferentemente das técnicas de PG tradicionais, em que o processo de busca se dá sobre os próprios modelos candidatos, na EG o mecanismo de busca atua no genótipo, representado por uma cadeia binária. Assim, é possível evoluir os programas utilizando um AG binário clássico, ao invés de realizar operações mais complexas em estruturas de árvore (como ocorre na PGG).

Neste trabalho, como originalmente proposto na EG, o algoritmo de busca utilizado foi um AG, mostrado no Algoritmo 1 . O fluxograma do AG utilizado é ilustrado na Figura 2.3. Entretanto, é importante ressaltar que a separação entre a representação dos indivíduos e os programas possibilita a utilização de outras metaheurísticas além dos AGs, como Otimização por Exame de Partículas (PSO, do inglês *Particle Swarm Optimization*) (O'Neill e Brabazon, 2006) e Sistemas Imunológicos Artificiais (Bernardino e Barbosa, 2011).

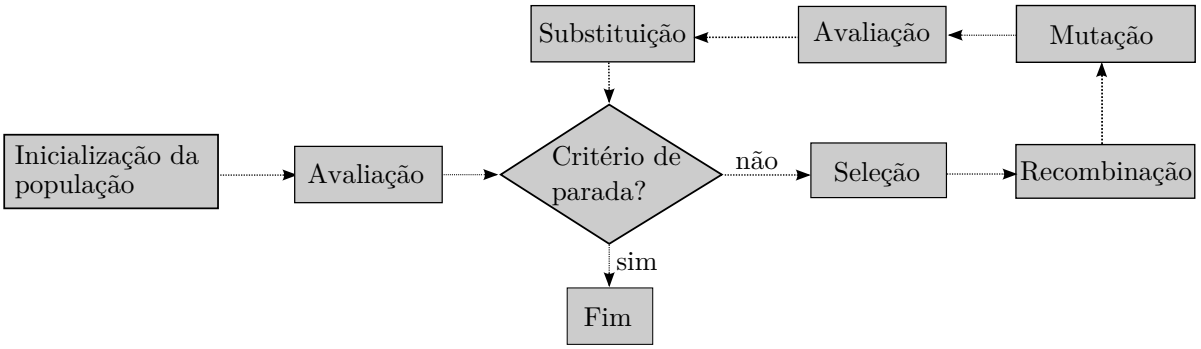



Figura 2.3: Algoritmo Genético.

A geração dos programas na EG é feita através de um processo de mapeamento que inicialmente converte o genótipo binário em um vetor de inteiros e então utiliza as regras de produção de uma gramática formal para a geração de um modelo candidato.

A primeira etapa consiste na conversão do genótipo binário em um vetor de inteiros, usualmente utilizando codificação de 8 bits (O'Neill e Ryan, 2003). Esta etapa é ilustrada na Figura 2.4.

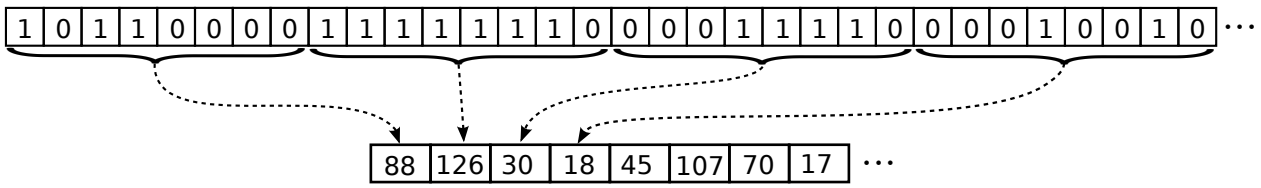



Figura 2.4: Exemplo da primeira etapa do mapeamento, utilizando 8 bits para cada inteiro.

A segunda etapa consiste na utilização do vetor de inteiros para seleção de regras de produção da gramática. O processo parte do símbolo inicial $S$ e prossegue até que existam apenas símbolos terminais ($\in \Sigma$). Quando houver mais de um não-terminal ($\in N$) para ser derivado, uma estratégia comum é escolher sempre aquele mais à esquerda

da cadeia. Dado um não-terminal ($\in N$), a escolha de uma regra de produção é realizada através da Equação:

$$regra = a \, mod \, (nr), \tag{2.1}$$

onde $a$ é o próximo inteiro na sequência e $nr$ é o número de regras de produção do não-terminal atual. A cada passo, procura-se o não-terminal mais à esquerda e obtém-se o próximo número inteiro. O número da regra de produção é calculado utilizando a Equação 2.1. Finalmente, o não-terminal atual é substituído pela regra de produção selecionada.

Um exemplo de mapeamento de uma solução candidata é ilustrado na Tabela 2.1, na qual é utilizada a gramática especificada na Seção anterior e os números inteiros utilizados são os obtidos na Figura 2.4. Cada linha da tabela representa um passo da produção, correspondendo à substituição de um símbolo não-terminal (em negrito) por uma de suas produções.

Tabela 2.1: Exemplo de mapeamento de vetor de inteiros em um programa.

| passo | entrada | valor | resultado |
|---|---|---|---|
| 1 | **<expr>** | (88) mod (4) = 0 | (<expr><op><expr>) |
| 2 | (**<expr>**<op><expr>) | (126) mod (4) = 2 | (<var><op><expr>) |
| 3 | (**<var>**<op><expr>) | (30) mod (2) = 0 | (X <op><expr>) |
| 4 | (X **<op>**<expr>) | (18) mod (4) = 2 | (X * <expr>) |
| 5 | (X * **<expr>**) | (45) mod (4) = 1 | (X * (<op_u> <expr>)) |
| 6 | (X * (**<op_u>** <expr>)) | (107) mod (3) = 2 | (X * (cos <expr>)) |
| 7 | (X * (cos **<expr>**)) | (70) mod (4) = 2 | (X * (cos <var>)) |
| 8 | (X * (cos **<var>**)) | (17) mod (2) = 1 | X * cos(Y) |

Na Figura 2.5 é mostrado um exemplo envolvendo as etapas do mapeamento, partindo da representação binária do genótipo até a geração de um programa candidato.

### 2.5.1 Avaliação das soluções candidatas

A avaliação dos indivíduos é outra etapa dependente do problema. No caso da regressão simbólica, a avaliação consiste na execução dos programas candidatos sobre um conjunto de treinamento. A aptidão de um indivíduo é calculada aqui como a soma dos quadrados das diferenças entre os valores esperados, e aqueles obtidos pela execução do programa candidato, como mostrado na Equação 2.2.

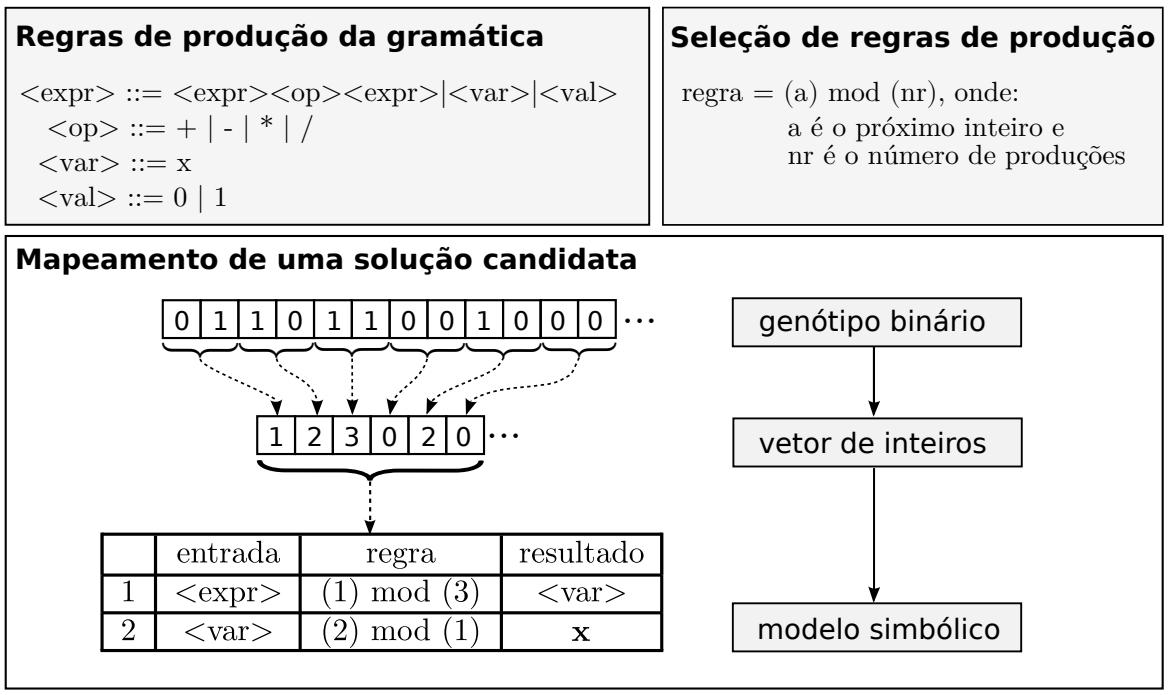



Figura 2.5: Exemplo contendo as duas etapas do mapeamento de uma solução candidata.

$$\text{aptidão}(p_i) = \sum_{j=1}^{N} [y_j - p_i(x_j)]^2, \tag{2.2}$$

onde $p_i$ é o i-ésimo programa, $N$ é o tamanho do conjunto de treinamento e $y_j$ é o valor esperado para a entrada $x_j$.

Apesar de sua robustez, as técnicas de PG possuem alta demanda computacional, sobretudo na etapa de avaliação dos modelos candidatos. Este custo é diretamente proporcional à complexidade dos modelos, ao tamanho do conjunto de treinamento, além do tamanho da população e número de gerações. Por outro lado, estas técnicas são consideradas naturalmente paralelas, pois há vários componentes que podem ser computados de forma independente sobretudo na etapa de avaliação dos modelos candidatos. Portanto, a adoção de computação paralela é uma alternativa promissora para acelerar a execução do algoritmo e, assim, possibilitar a resolução de problemas mais complexos e/ou reduzir seu tempo de execução. Na próxima Seção serão discutidos conceitos importantes para a compreensão dos modelos de paralelismo propostos.

Além disso, geralmente as implementações da EG realizam a avaliação dos indivíduos utilizando um **interpretador**. Após o mapeamento, os programas gerados são mantidos em estruturas de dados e executados passo a passo por um procedimento de

interpretação. Para simplificar o processo de interpretação, as gramáticas aqui utilizadas geram expressões aritméticas em **notação pós-fixada**, ou Notação Polonesa Inversa, que dispensa a utilização de parênteses para representação da prioridade das operações de uma expressão. Na Tabela 2.5.1 são mostrados alguns exemplos de expressões em notação convencional e a expressão correspondente utilizando notação pós-fixada.

Tabela 2.2: Exemplos de expressões em notação pós-fixada.

| Notação convencional | Notação pós-fixa |
|---|---|
| $a + b$ | $a\ b\ +$ |
| $(a + b) + c$ | $a\ b\ +\ c\ +$ |
| $(a + b)/c$ | $a\ b\ +\ c\ /$ |
| $x * cos(y)$ | $x\ y\ cos\ *$ |

A estrutura de um interpretador é mostrada no Algoritmo 2 (Augusto e Barbosa, 2013). O interpretador recebe como entradas o programa, representado por um vetor de inteiros em notação pós-fixada e um registro de treinamento para avaliação. O algoritmo utiliza uma estrutura de pilha, na qual cada operação executada tem seu resultado colocado sobre o topo e posteriormente desempilhado quando necessário. No algoritmo, o comando $TIPO$ extrai o tipo de instrução; $VALOR$ extrai o valor numérico de uma constante ou índice de variável; $PUSH$ acrescenta um valor ao topo da pilha e $POP$ extrai o valor presente no topo da pilha.

Como os modelos são programas válidos segundo uma gramática, também é possível realizar a avaliação através da **compilação e execução** dos programas sobre os dados de treinamento. Apesar da interpretação possuir maior custo computacional que a execução direta do código, esta técnica não adiciona nenhum custo adicional à execução, enquanto na compilação é necessária uma etapa de transformação dos códigos-fonte em código de máquina. Neste trabalho as duas abordagens são analisadas em implementações massivamente paralelas.

**Algoritmo 2:** Interpretador para uma solução candidata identificada por *programa* ao avaliar o dado $X$.

```
1  início
2      para i ← 0  até  tamanho_programa − 1 faça
3          selecione TIPO( programa[i] ) faça
4              caso Constante
5                  PUSH( VALOR(programa[i]) );
6              fim caso
7              caso Variável
8                  PUSH( X[VALOR( programa[i])] );
9              fim caso
10             caso Soma
11                 PUSH( POP() + POP() );
12             fim caso
13             caso Subtração
14                 PUSH( POP() - POP() );
15             fim caso
16             caso Multiplicação
17                 PUSH( POP() * POP() );
18             fim caso
19             caso Divisão
20                 PUSH( POP() / POP() );
21             fim caso
22             ⋮
23         fim selec
24     fim para
25 fim
```

# 3 Paralelismo

## 3.1 Introdução

A Lei de Moore, conjectura feita na década de 1960, previa que o poder computacional das unidades centrais de processamento (CPUs) seria multiplicado por 2 ao final de cada biênio. Tal previsão se baseava na possibilidade de se aumentar progressivamente a densidade de transístores nos *chips* dos processadores (Kish, 2002). Neste cenário, melhorias de desempenho eram obtidas com o surgimento de processadores cada vez mais rápidos, sem a necessidade de modificação de *software*.

Entretanto, a possibilidade de se aumentar a quantidade de transístores por *chip* chegou a limites físicos, em razão por exemplo do aumento de calor gerado por esses dispositivos (Kish, 2002). A partir de então, a indústria tem adotado outras estratégias para o aumento da capacidade de computação, como a criação dos processadores *multicore*: ao invés de aumentar-se a densidade de transístores por processador, os circuitos integrados passaram a conter vários processadores completos (Pacheco, 2011).

O desenvolvimento dos processadores *multicore* foi capaz de melhorar a vazão dos sistemas, pois vários processos podem ser executados simultaneamente pelo sistema operacional. Todavia, isto não reduz o tempo necessário para execução de uma tarefa, uma vez que os programas seriais não exploram a existência de vários núcleos de processamento, comportando-se como se houvesse apenas um (Pacheco, 2011). Para melhorar o desempenho de um programa nesta arquitetura, são necessárias técnicas que possibilitem a utilização de vários processadores, como o uso de *threads* ou de múltiplos processos.

Segundo Tanenbaum (1995), todos os *softwares* que podem executar em um computador, inclusive o próprio sistema operacional, são organizados em vários processos [2]. O sistema operacional, através de um mecanismo denominado multiprogramação, é capaz de alternar entre a execução de diferentes processos, executando cada processo por

[2]Um processo é um programa em execução acompanhado dos valores atuais dos contadores de programa, dos registradores, das variáveis e de outros recursos, como por exemplo dados sobre os arquivos abertos pelo processo (Tanenbaum, 1995).

um pequeno intervalo de tempo. É dessa forma que os sistemas operacionais modernos são capazes de executar várias tarefas simultaneamente, mesmo que o computador tenha somente uma unidade de processamento.

Outro conceito apresentado pelo processo é a *thread* (fluxo) de execução, que mantém controle sobre as instruções que serão executadas, além de registradores contendo o valor das variáveis utilizadas. Tipicamente um processo possui apenas uma *thread*. Entretanto, não raramente um processo pode conter vários *threads*, que compartilham os mesmos recursos e espaço de endereçamento. A utilização de várias *threads* num processo é vantajosa, quando comparada à utilização de múltiplos processos pois, como as *threads* compartilham os mesmos recursos, a alternância entre sua execução é menos custosa ao sistema operacional. Por essa característica, as *threads* são por vezes chamadas de processos leves (Tanenbaum, 1995).

Muitos avanços de *hardware* foram promovidos com o intuito de aumentar o desempenho das CPUs, como *pipelining* e *caching*. O *pipelining* é um paralelismo a nível de instruções, enquanto *caching* consiste na utilização de uma memória de acesso mais rápido pela CPU, explorando as localidades espaciais e temporais de acesso (Pacheco, 2011).

Aqui serão considerados sistemas paralelos como aqueles em que o programador pode criar soluções que explorem explicitamente a existência de *hardware* paralelo. A classificação destes sistemas geralmente utiliza a taxonomia apresentada por Flynn (1966), que considera a capacidade de execução de fluxos de instruções e de dados. Esta taxonomia define os seguintes conceitos (Pacheco, 2011):

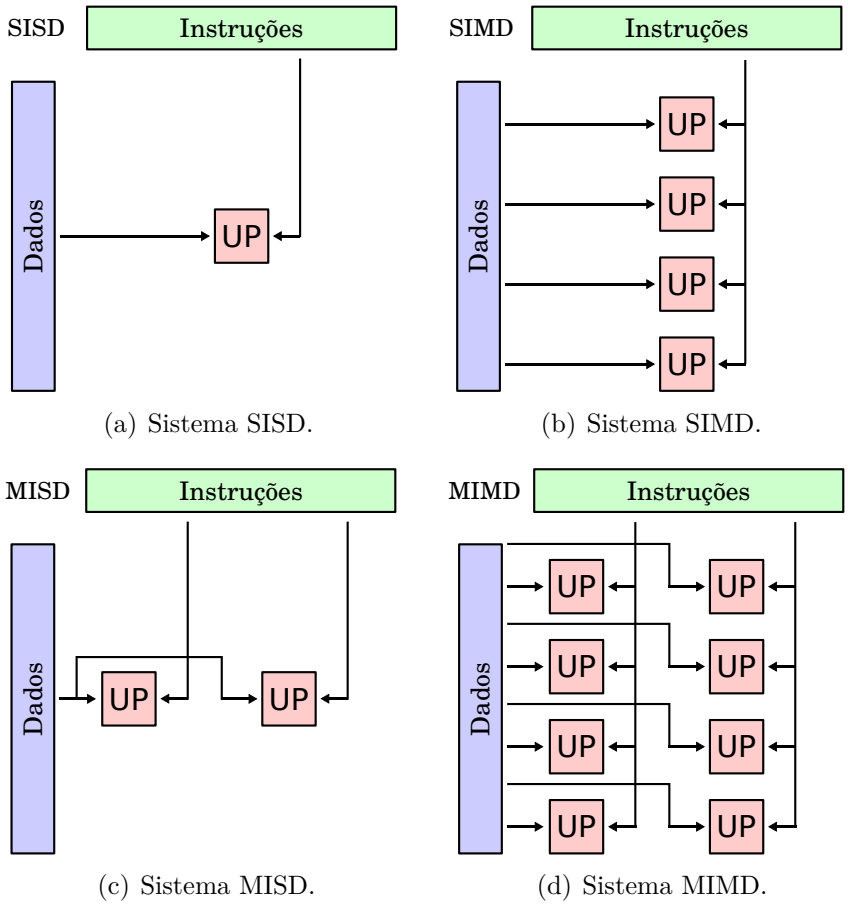
- *Single Instruction Stream - Single Data Stream* (SISD): Compreende sistemas capazes de executar uma única instrução sobre um fluxo de dados a cada instante. Processadores com um único núcleo pertencem a esta categoria (Figura 3.1(a)).

- *Single Instruction Stream - Multiple Data Stream* (SIMD): Sistemas em que cada unidade de processamento executa a mesma instrução em um instante, mas com fluxos de dados distintos (Figura 3.1(b)). Processadores vetoriais e GPUs constituem exemplos desta categoria.

- *Multiple Instruction Stream - Single Data Stream* (MISD): Sistemas em que as unidades de processamento são capazes de executar diferentes instruções a cada instante, porém sobre um mesmo fluxo de dados (Figura 3.1(c)).

- *Multiple Instruction Stream - Multiple Data Stream* (MIMD): Sistemas que suportam a execução de múltiplas instruções sobre vários fluxos de dados (Figura 3.1(d)). Tipicamente, consistem em várias unidades de processamento independentes, cada um com suas próprias unidade de controle e unidade lógica e aritmética. Sistemas de memória compartilhada e distribuída são exemplos desta categoria.

Estas categorias são ilustradas na Figura 3.1, na qual "UP" representa uma unidade de processamento do sistema.




(a) Sistema SISD. (b) Sistema SIMD.

(c) Sistema MISD. (d) Sistema MIMD.

Figura 3.1: Taxonomia de Flynn.

Um modelo adicional muito utilizado é o *Single Program - Multiple Data* (SPMD) (Darema et al., 1988), no qual um mesmo programa é executado em paralelo por todos os elementos do sistema. Ao invés de executar programas diferentes em cada unidade,

programas SPMD consistem de um único executável, que é capaz de simular a execução de vários programas com o uso de estruturas condicionais (Pacheco, 2011). Os Algoritmos 3 e 4 mostram dois programas genéricos que utilizam este modelo. No primeiro caso, ocorre o paralelismo de tarefas, no qual o problema é decomposto em N sub-tarefas que devem ser executadas paralelamente; no segundo, há paralelismo de dados, em que uma mesma tarefa é executada em paralelo, utilizando diferentes conjuntos de dados. Nos dois algoritmos, $thread_{id}$ representa a identificação do *thread* que está sendo executado.

**Algoritmo 3:** SPMD com paralelismo de tarefas

```
1  início
2      se thread_id = 1 então
3          Executa tarefa 1;
4      fim se
5      se thread_id = 2 então
6          Executa tarefa 2;
7      fim se
       ⋮
8      se thread_id = N então
9          Executa tarefa N;
10     fim se
11 fim
```

**Algoritmo 4:** SPMD com paralelismo de dados

**Entrada**: $V$: vetor de dados

```
1  início
2      x ← V[thread_id];
3      Executa tarefa (x);
4  fim
```

## 3.2 GPGPU

Uma arquitetura SPMD amplamente utilizada nos últimos anos é a **GPGPU** (*General-purpose computing on graphics processing units*), na qual unidades de processamento gráfico (GPUs) são utilizadas para computação de propósito geral (Owens et al., 2008).

Assim, operações convencionais, não necessariamente dedicadas ao processamento gráfico, podem ser executadas. Um dos fatores que incentiva a utilização desta arquitetura é a boa relação custo/benefício das GPUs, que geralmente possuem centenas ou milhares de unidades de processamento.

As GPUs sempre foram dispositivos com grandes recursos computacionais (Owens et al., 2008), mas apenas recentemente seu poder computacional passou a ser explorado em computação de propósito geral. Impulsionados pela indústria de jogos, avanços tanto em *hardware* como em *software* foram desenvolvidos com o intuito de suportar modelos de programação mais genéricos.

O desenvolvimento de aplicações GPGPU, também chamado de *GPU computing*, era inicialmente uma tarefa árdua, pois exigia o mapeamento do problema em primitivas de um *pipeline* gráfico (Owens et al., 2008). Cada etapa deste *pipeline* era desempenhada por uma unidade de *hardware* de função fixa. Com o passar do tempo, o *hardware* das GPUs tornou-se mais genérico, substituindo-se algumas unidades de função fixa por unidades programáveis, possibilitando o desenvolvimento de um modelo de programação mais simples. Desta forma, os programadores podem utilizar diretamente estas unidades programáveis ao invés de dividir a computação a ser realizada ao longo do *pipeline* gráfico (Owens et al., 2008). A maior flexibilidade do *hardware* possibilitou o surgimento de ferramentas mais robustas para criação de aplicações GPGPU, como os *frameworks* CUDA[3] e OpenCL.

CUDA (*Compute Unified Device Architecture*) é uma plataforma de computação paralela de propósito geral criada pela Nvidia. Através dela, o desenvolvedor pode executar operações de propósito geral em dispositivos gráficos deste fabricante.

Neste trabalho foi utilizado o OpenCL, uma especificação não proprietária para criação de sistemas paralelos em ambientes heterogêneos (Gaster et al., 2011), que será detalhada na próxima Seção. O OpenCL foi adotado aqui pois, diferentemente do CUDA, pode ser utilizado em dispositivos de diferentes fabricantes, com pouca ou sem nenhuma alteração no código (Gothandaraman et al., 2011). Além disso, o *framework* possibilita a exploração do poder computacional de outros dispositivos, além das GPUs, implicando

[3]http://developer.nvidia.com/cuda-zone

em maior flexibilidade das soluções desenvolvidas.

Existem alguns fatores que podem influenciar o desempenho de sistemas paralelos executados em GPUs e que devem, portanto, ser considerados no projeto de algoritmos. Segundo Owens et al. (2008), os principais são:

- **Ênfase em paralelismo:** Como as GPUs são dispositivos naturalmente paralelos, quanto maior o número de tarefas desempenhadas em paralelo, maior o ganho de desempenho;
- **Minimização de divergências de instrução:** Como discutido anteriormente, o GPGPU é um modelo de paralelismo SPMD. Dessa forma, elementos de processamento que pertencem ao mesmo bloco devem apresentar poucas divergências de instrução, para que as tarefas sejam executadas realmente em paralelo; e
- **Maximização de operações aritméticas:** As transferências de dados entre a memória principal e a GPU são custosas em relação ao tempo de processamento. Dessa forma, é importante reduzir as transferências de dados e aumentar a quantidade de operações aritméticas executadas por cada elemento de processamento.

## 3.3 OpenCL

**OpenCL** (*Open Computing Language*) é um *framework* para a construção de sistemas paralelos em ambientes heterogêneos. Sua especificação é aberta e mantida pelo grupo Khronos[4]. O *framework* suporta diferentes níveis de paralelismo e pode ser utilizado eficientemente em ambientes computacionais homogêneos ou heterogêneos (Gaster et al., 2011). Estes ambientes podem ser formados por diferentes tipos de dispositivos, como CPUs, GPUs e unidades de processamento acelerado (APUs, do inglês *Accelerated Processing Units*).

O OpenCL define abstrações de *hardware* e *software*, oferecendo ao programador um modelo independente da estrutura real do *hardware*. Esta característica possibilita a criação de sistemas portáveis e de alto desempenho. É importante ressaltar, entretanto,

[4] `http://www.khronos.org`

que para obter o desempenho máximo pode ser necessário utilizar otimizações específicas para cada tipo de dispositivo (Gaster et al., 2011).

Para o desenvolvimento dos programas, o *framework* oferece uma linguagem de programação denominada OpenCL C, que utiliza um subconjunto das instruções da especificação ISO C99, acrescentando extensões para paralelismo. Um programa OpenCL é formado por um ou vários *kernels*, que são as funções executadas nos dispositivos. Além disso, o OpenCL fornece uma API para a coordenação da execução dos *kernels*. A arquitetura do OpenCL pode ser dividida em 4 modelos: Modelo de plataforma, Modelo de Execução, Modelo de Memória e Modelo de Programação (Khronos, 2011).

O **Modelo de plataforma** define um modelo abstrato de *hardware*, que consiste em um hospedeiro conectado a um ou mais dispositivos de computação compatíveis com OpenCL. Cada dispositivo é dividido em uma ou mais unidades computacionais, que por sua vez possuem um ou vários elementos de processamento, nos quais ocorre o processamento. O modelo de plataforma está ilustrado na Figura 3.3.

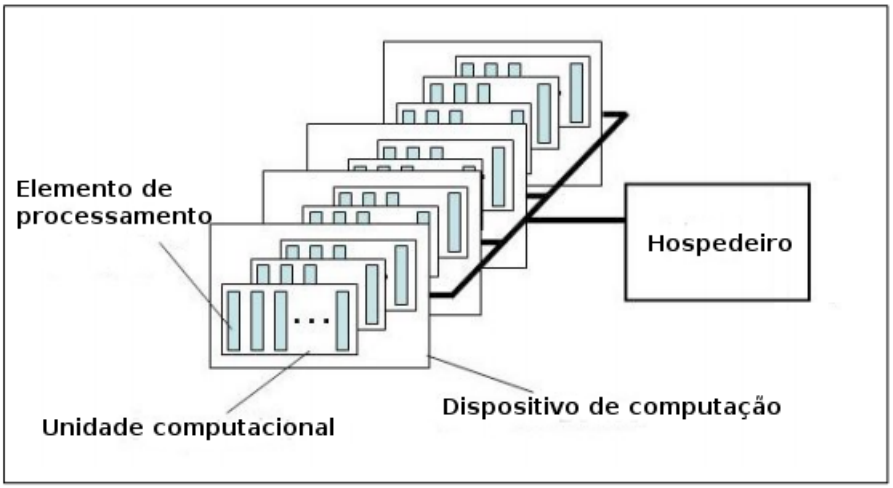



Figura 3.2: Modelo de plataforma, adaptado de Khronos (2011).

No **Modelo de execução** do OpenCL, existe um programa hospedeiro que gerencia a execução de *kernels* em um ou mais dispositivos, através de um contexto de execução. Este contexto inclui os dispositivos utilizados, um ou mais *kernels* e objetos para manipulação de memória. A comunicação entre o hospedeiro e os dispositivos se dá utilizando uma fila de comandos, na qual são submetidas as execuções de *kernels* e operações de leitura e escrita em memória.

Para a execução de um *kernel*, o programador define um espaço de execução, que pode ter uma, duas ou três dimensões. Para cada ponto deste espaço é executada uma instância do kernel, chamada de *work-item*. *Work-items* são organizados em grupos chamados *work-groups*, sendo cada item de um *work-group* executado de forma concorrente através da mesma unidade computacional. O número de *work-items* em cada *work-group* é chamado *local size*, enquanto *global size* é o número total de *work-items*.

O **Modelo de memória** define a hierarquia de memória acessível pelos *kernels*, independente da arquitetura física do dispositivo. Esta hierarquia compreende os espaços de memória global, constante, local e privado, definidos a seguir:

- **Memória global**: Esta região de memória é visível por todos os *work-items* de todos os grupos, permitindo acessos de leitura e escrita.
- **Memória constante**: Região de memória acessível por todos os *work-items*, assim como a memória global. Entretanto, seu conteúdo não pode ser alterado, permitindo somente acessos de leitura. Dependendo do dispositivo, esta memória pode ter latência de acesso menor que a memória global, como é o caso das GPUs.
- **Memória local**: Região de memória local compartilhada pelos itens de um *work-group*.
- **Memória privada**: Espaço de memória visível apenas para o *work-item*. Variáveis declaradas no espaço privado de um *work-item* não serão acessíveis por nenhum outro.

A memória global é acessível ao hospedeiro através de estruturas de dados denominadas *buffers*, que são gerenciadas pelo *framework*. Para transferir dados para o dispositivo, o hospedeiro utiliza a fila de comandos para solicitar operações de escrita. Para copiar dados da memória do dispositivo, o hospedeiro solicita operações de leitura nos *buffers*.

Em relação à consistência da memória, o OpenCL adota um modelo "relaxado": O estado da memória visível por todos os *work-items* não é necessariamente o mesmo. Em outras palavras, as alterações realizadas por um *work-item* podem não ser imediatamente visíveis aos outros. No caso da memória local, é possível introduzir barreiras de

sincronismo no programa, a partir das quais o estado da memória local é garantidamente consistente ao longo de todos os *work-items* do grupo. Em relação à memória global, não há como garantir a consistência entre *work-items* de grupos diferentes. A Figura 3.3 ilustra a hierarquia de memória do OpenCL.

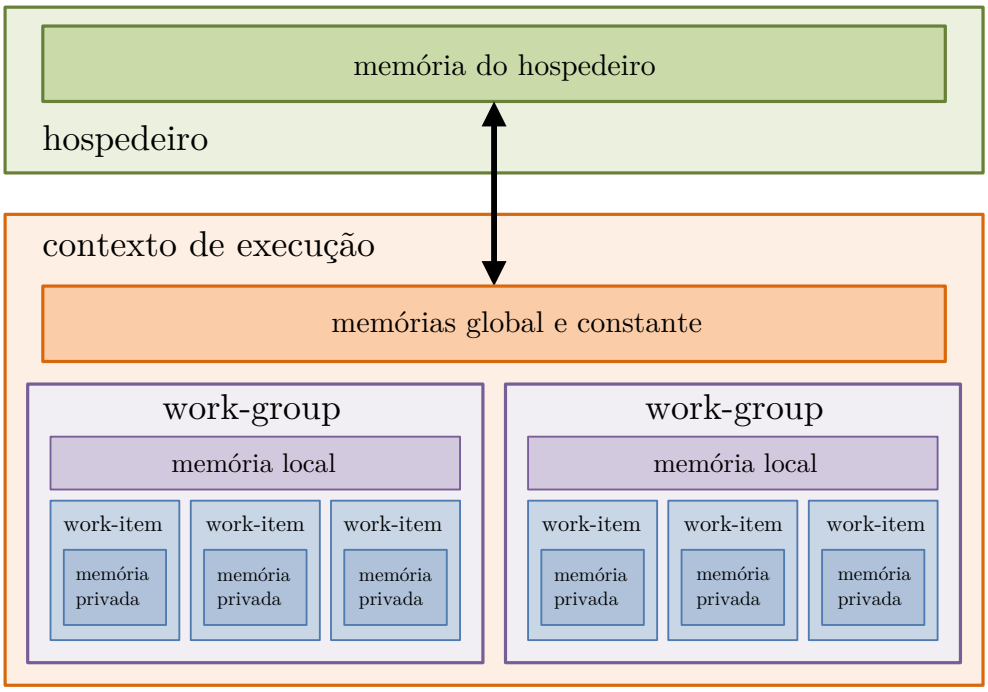



Figura 3.3: Modelo de memória do OpenCL.

**Modelo de programação:** Os modelos de programação suportados são o paralelismo de tarefas e de dados, sendo este último o foco da especificação.

# 4 Evolução Gramatical massivamente paralela

Neste capítulo são apresentadas as propostas de paralelização da Evolução Gramatical na arquitetura GPGPU utilizando OpenCL. Inicialmente são apresentados dois modelos de paralelismo da etapa de avaliação: o primeiro utilizando interpretação dos modelos e, o segundo, compilação. Finalmente, é apresentado um modelo em que todo o processo de busca é realizado em paralelo.

## 4.1 Paralelismo da etapa de avaliação

Na Evolução Gramatical, assim como em outras metaheurísticas, a etapa de avaliação é a que tem maior custo computacional. Este custo é diretamente proporcional ao tamanho das bases de dados e da complexidade dos modelos. Considerando que esta etapa concentra grande parte do custo de processamento, poderia se pensar em utilizar computação paralela para reduzir seu tempo de execução, viabilizando o uso da técnica em problemas de grande porte. Assim, nesta Seção são propostos dois modelos de paralelismo da etapa de avaliação (linha 8 do Algoritmo 1) utilizando a arquitetura GPGPU.

### 4.1.1 Avaliação via interpretação dos modelos candidatos

A estratégia de paralelismo apresentada aqui pode ser classificada como do tipo mestre-escravo[5], pois todo o processo de busca da EG ocorre no hospedeiro, com exceção da etapa de avaliação da população. Neste passo, o hospedeiro realiza a cópia da população corrente para o dispositivo OpenCL e adiciona a execução do *kernel* de avaliação à fila de comandos. O *kernel* de avaliação é responsável por calcular a aptidão de cada indivíduo e disponibilizar estes dados em um *buffer* da memória global do dispositivo. Ao final da

[5]Paradigma de programação paralela que possui duas entidades: um mestre é responsável pela decomposição e distribuição das tarefas para um ou vários escravos, que executam as tarefas recebidas e enviam seu resultado de volta ao mestre (Buyya, 1999).

execução deste *kernel*, o hospedeiro solicita a cópia das aptidões da memória global do dispositivo para sua estrutura de dados da população.

O Algoritmo 5 exibe um pseudo-código para a etapa de avaliação, do ponto de vista do hospedeiro. Inicialmente é utilizada a função $clEnqueueWriteBuffer$, que realiza a cópia dos dados da memória do hospedeiro (*populacao*) para a memória global do dispositivo ($bufferPopulacao$). Depois é executada a função $clEnqueueNDRangeKernel$, encarregada de enfileirar o *kernel* de avaliação ($kernelAvaliacao$) para execução. Na linha 4, é utilizada uma função de sincronismo, $clFinish$, que aguarda até o término da execução de todos os comandos presentes na fila de comandos. Na linha 5 é utilizada a função $clEnqueueReadBuffer$, que solicita a cópia das aptidões calculadas pelo *kernel*, armazenadas em $bufferAptidao$, para a variável *aptidao* do hospedeiro. Finalmente, um laço é utilizado para atribuir as aptidões a cada um dos indivíduos.

**Algoritmo 5:** Algoritmo do hospedeiro para execução do *kernel* de avaliação.

```
1 início
2     clEnqueueWriteBuffer(queue, bufferPopulacao, populacao);
3     clEnqueueNDRangeKernel(queue, kernelAvaliacao, local_size, global_size);
4     clFinish(queue);
5     clEnqueueReadBuffer(queue, bufferAptidao, aptidao);
6     para i ← 0 até tamanho_populacao − 1 faça
7         populacao[i].aptidao ← aptidao[i];
8     fim para
9 fim
```

Existem na literatura trabalhos que propõem estratégias de paralelismo da PG para a arquitetura GPGPU (Harding e Banzhaf, 2007; Ando e Nagao, 2007; Chitty, 2007; Robilliard et al., 2008; Pospichal et al., 2011; Augusto e Barbosa, 2013). Tais trabalhos utilizam paralelismo de dados, de programas ou uma combinação das duas abordagens.

No paralelismo de dados, um indivíduo é avaliado por vez, sendo os registros de treinamento avaliadas em paralelo. Segundo Augusto e Barbosa (2013), a desvantagem desta abordagem é a necessidade de um conjunto de treinamento com muitos registros para explorar o poder de processamento das GPUs. Uma outra possibilidade é o paralelismo de programas, no qual os indivíduos são avaliados paralelamente, enquanto os registros

de treinamento são avaliados de forma sequencial. Apesar de proporcionar maior taxa de utilização do processamento, mesmo com poucos registros, esta abordagem tende a causar muitas divergências de instrução, levando a um baixo desempenho computacional (Augusto e Barbosa, 2013).

O modelo de paralelismo da avaliação adotado aqui é baseado em Augusto e Barbosa (2013) e consiste em utilizar as duas abordagens em conjunto: alguns indivíduos são avaliados simultaneamente, sendo os registros de treinamento computados concomitantemente. Neste modelo, cada indivíduo é avaliado em uma unidade computacional e cada elemento de processamento executa um ou mais registros de treinamento. O processo de avaliação é ilustrado na Figura 4.1. Nesta estratégia, os programas gerados são transferidos para estruturas de dados na memória global do dispositivo, assim como os dados de treinamento. A avaliação dos indivíduos é dividida entre as unidades computacionais. Estas, por sua vez, dividem o processamento entre seus elementos de processamento, de forma que cada um possua aproximadamente o mesmo volume de dados.

Como comentado na Seção anterior, o modelo de programação utilizado nas GPUs modernas é o SPMD, que é suportado pelo OpenCL. Desta forma, um mesmo programa deve ser executado por todas as unidades computacionais do dispositivo. Considerando que a população é formada de programas distintos, isto poderia representar um obstáculo para a implementação de um procedimento de avaliação. No entanto, com a utilização de um interpretador é possível eliminar esta dificuldade, pois um mesmo *kernel* é capaz de executar diferentes programas e, assim, avaliar toda a população.

No OpenCL, o espaço de execução de um *kernel* é definido pelos parâmetros *global size* e *local size*. O primeiro indica o número total de work-items necessários e o último representa o número de work-itens em cada work-group. Os valores de *local size* e *global size* aqui utilizados são dados pelas Equações 4.1 e 4.2, respectivamente. O parâmetro $local\_size_{max}$ depende do dispositivo utilizado e indica o número máximo de *work-items* permitidos em um *work-group*, enquanto $registros_{tamanho}$ representa o tamanho do conjunto de treinamento.

O *local size* utilizado propicia um balanceamento de carga entre *work-items* de um mesmo *work-group*, de forma que cada um processe aproximadamente o mesmo número

de registros de treinamento. O valor do *global size*, que corresponde ao número total de *work-items* necessários, é igual ao número de *work-items* em cada grupo multiplicado pelo tamanho da população.

$$local_{size} = min\{local\_size_{max}, registros_{tamanho}\} \tag{4.1}$$

$$global_{size} = população_{tamanho} \times local_{size} \tag{4.2}$$

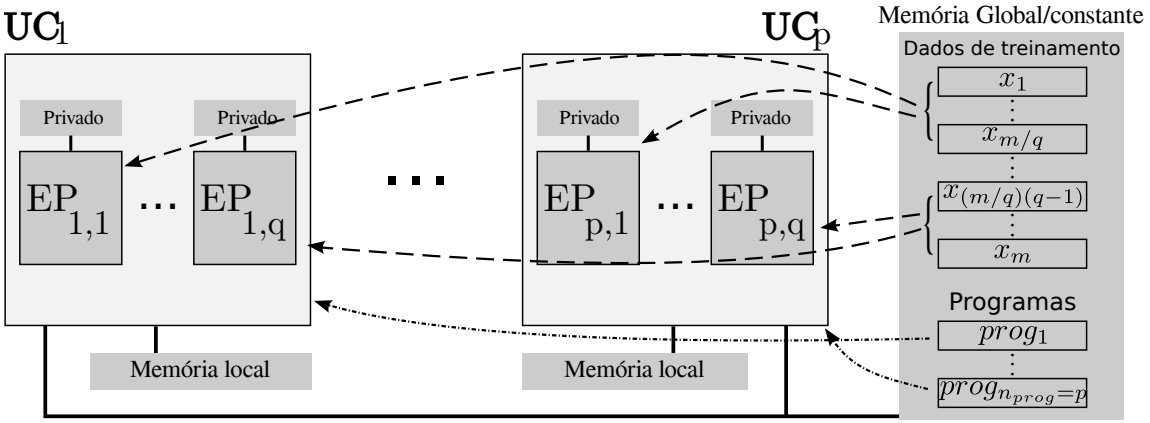



Figura 4.1: Paralelismo da etapa de avaliação, adaptado de Augusto e Barbosa (2013).

No problema aqui tratado (regressão simbólica), a aptidão de uma solução candidata é calculada executando-se o programa correspondente sobre um conjunto de dados e comparando os resultados obtidos com os valores esperados. No Algoritmo 6 é ilustrado o *kernel* de avaliação utilizando um interpretador (Augusto e Barbosa, 2013). Neste Algoritmo, *get_local_id*, *get_group_id* e *get_local_size* são funções disponíveis no OpenCL que retornam, respectivamente, o identificador do elemento de processamento em seu *work-group*, o identificador do *work-group* do qual faz parte o elemento de processamento e o número de *work-items* em cada *work-group*.

Os parâmetros do *kernel* são o conjunto de programas gerados na etapa de mapeamento (*Programas*) e o conjunto de treinamento utilizado ($X$ e $Y$). O parâmetro *Programas* é uma lista de programas, cada um composto por um vetor de números inteiros, representando um programa/expressão em notação pós-fixada. A entrada $X$ é um vetor de $registro_{tamanho} \times dimensões_{tamanho}$ elementos que contém os dados de treinamento. O parâmetro $Y$ representa os valores esperados para os dados de treinamento.

O algoritmo opera como a seguir. Nas linhas 2 e 3 são obtidos o identificador local e o do *work-group*, respectivamente. Na linha 4 é obtido o tamanho do *work-group* (número de *work-items*). Como cada *work-group* é responsável pela avaliação de um indivíduo, o identificador do grupo ($group_{id}$) é utilizado para indexar o programa que deve ser avaliado. Na linha 5, a variável local $p$ recebe uma cópia do programa a ser avaliado. O laço entre as linhas 6 e 11 percorre os registros que devem ser avaliados pelo *work-item*. A cada passo, cada *work-item* avalia um registro; como há $local_{size}$ *work-items*, $local_{size}$ registros são avaliados simultaneamente, por cada *work-group*. Na linha 7 a variável $id$ recebe o índice do registro a ser avaliado no passo corrente, calculado como $i \times local_{size} + local_{id}$. Desta forma os elementos de processamento acessam regiões adjacentes de memória. A comparação realizada na linha 8 garante que não serão realizados acessos além do tamanho do vetor de dados. Na linha 9 o erro correspondente à execução do registro $X$ é calculado utilizando um interpretador. O erro é dado pelo quadrado da diferença entre o valor esperado, $Y[id]$, e aquele gerado pela execução do programa $p$, para o registro $X[id]$. Finalmente, na linha 12, é realizado um procedimento de redução paralela, que agrega os valores dos erros locais de todos os *work-items* e atribui à variável $Erro_{total}$. Um exemplo de soma paralela é ilustrado na Figura 4.2.

**Algoritmo 6:** *Kernel* de avaliação de uma solução candidata utilizando interpretação.

```
1  início
2      local_id ← get_local_id(0);
3      group_id ← get_group_id(0);
4      local_size ← get_local_size(0);
5      p ← CopiaPrograma(Programas[group_id]);
6      para i ← 0 até ⌈registros_tamanho/local_size⌉ faça
7          id ← i × local_size + local_id;
8          se id < registros_tamanho então
9              Erro[local_id] ← Erro[local_id] + |Interpretador(p, X[id]) − Y[id]|²;
10         fim se
11     fim para
12     Erro_total[group_id] ← ReduçãoParalela(0, ..., local_size − 1);
13 fim
```

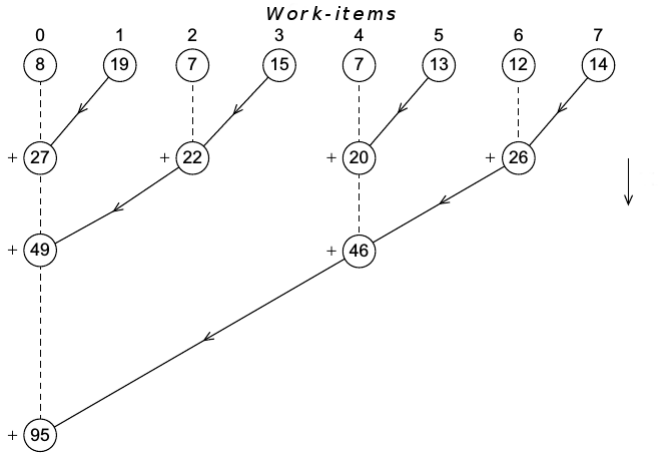



Figura 4.2: Exemplo de soma paralela, adaptado de (Pacheco, 2011).

O procedimento de redução paralela é mostrado no Algoritmo 7. O funcionamento do algoritmo considera que o $local_{size}$ seja uma potência de 2. Assim, na linha 2 é computado o valor da próxima potência de 2, a partir do $local_{size}$ (inclusive). A cada iteração do laço, cada *work item* realiza a soma de 2 elementos do vetor, somando o valor da posição $local_{id}$ à posição $local_{id} + i$ e atribuindo o resultado à posição $local_{id}$. O sincronismo_local é necessário para que as operações realizadas sejam visíveis a todos os *work-items* do *work-group*. É importante ressaltar que, como cada indivíduo é avaliado

por um *work group*, os erros parciais são computados por elementos de processamento da mesma unidade computacional e, assim, não são necessários sincronismos globais. Ao final da iteração, o valor de $i$ é dividido pela metade, assim como o número de elementos do vetor a serem somados na próxima iteração. O processo se repete até que toda a soma seja computada e seu valor seja atribuído ao primeiro elemento do vetor.

**Algoritmo 7:** Procedimento de redução paralela.

**Entrada**: $local_{id}$, $local_{size}$: inteiros, *erros*: vetor de $float$ em memória local

```
1  início
2      proxima_potencia_de_2 = pow(2, ceil ( log2( local_size)));
       i ← proxima_potencia_de_2/2;
3      enquanto i > 0 faça
4          sincronismo_local();
5          se (local_id < i) & (local_id + i < local_size) então
6              erros[local_id] ← erros[local_id] + erros[local_id + i];
7          fim se
8          i ← i/2;
9      fim enquanto
10 fim
```

### 4.1.2 Avaliação via compilação dos modelos candidatos

Como os programas são gerados por meio de uma gramática, existe também a possibilidade de realizar a avaliação através da compilação e execução das soluções candidatas. Assim como na estratégia anterior, nos modelos propostos utilizando compilação dos modelos candidatos, somente a etapa de avaliação é paralelizada. Além disso, o modelo de paralelismo adotado também é o de programas e dados.

Inicialmente, poderia se pensar em compilar e executar cada programa individualmente, como feito em Chitty (2007). Desta forma, em cada etapa do processo evolutivo são realizadas $tamanho_{populacao}$ compilações. Considerando a utilização de GPUs para acelerar esta etapa do processo de busca, esta abordagem seria conceitualmente correta, uma vez que o modelo de programação utilizado nestes dispositivos é o SPMD, no qual somente um programa pode ser executado a cada instante. Entretanto, foram realizados experimentos preliminares utilizando OpenCL, e concluiu-se que apesar da execução da

avaliação ser mais rápida do que na interpretação, o custo adicional de compilação de cada indivíduo separadamente torna inviável esta estratégia.

Uma outra possibilidade, proposta neste trabalho e publicada em (Russo et al., 2014b), é compilar todos os indivíduos em um mesmo *kernel*, introduzindo uma estrutura de controle que permita avaliar cada um deles independentemente em paralelo, por unidades computacionais distintas. Após as etapas de seleção, recombinação e mutação, todos os indivíduos são mapeados em programas e concatenados no mesmo código fonte, que é então compilado e executado. Dessa forma, a cada iteração do algoritmo é realizada uma única compilação englobando toda a população.

Ao invés de compilar toda a população em um único programa, pode-se pensar também em dividir esta etapa, realizando a compilação de blocos de indivíduos. Portanto, a cada iteração do algoritmo evolutivo são realizadas $\lceil popula\text{ção}_{tamanho}/bloco_{tamanho} \rceil$ compilações.

A vantagem desta compilação em blocos é que enquanto um bloco é executado, o próximo pode ser compilado, ocorrendo portanto uma sobreposição entre as tarefas executadas no hospedeiro e no dispositivo OpenCL, contribuindo para a redução do tempo total de execução da técnica. Entretanto, foi identificado que a escolha do tamanho ideal para cada bloco não é trivial: para que exista ganho de desempenho, o tempo de compilação de cada bloco deve ser idealmente menor ou igual ao tempo de avaliação, pois, desta forma, sempre haverá um bloco sendo avaliado enquanto outro é compilado (com exceção do primeiro bloco compilado). Caso contrário, as compilações adicionais contribuem para o aumento do tempo total de execução.

Identificou-se que esta metodologia de compilação por blocos pode ser vantajosa quando utilizada com grandes populações e/ou grandes conjuntos de treinamento. Foram realizados experimentos preliminares utilizando uma população de tamanho igual a 1000, $10^7$ registros de treinamento e 7 tamanhos de bloco: 100, 128, 200, 256, 300, 500 e 1000. O melhor desempenho foi obtido ao utilizar blocos de tamanho 200, observando-se uma redução de $33,72\%$ no tempo de execução, em relação ao obtido através da compilação de todos os indivíduos. Entretanto, estudos mais detalhados devem ser realizados para encontrar a melhor relação de sobreposição entre compilação e avaliação dos modelos.

Considerando que o custo computacional das avaliações ao longo da execução do algoritmo não é constante, e sim dependente das características dos modelos presentes na população, pode-se pensar também em uma estratégia adaptativa, que ajuste o tamanho dos blocos de acordo com o tempo de execução das últimas avaliações.

Outra abordagem que pode ser utilizada é criar um *kernel* distinto para avaliação de cada indivíduo. Neste caso, a cada iteração do algoritmo compila-se um programa contendo vários *kernels*, que são posteriormente adicionados à fila de execução do dispositivo. Apesar de ser uma estratégia compatível com os conceitos do OpenCL, experimentos mostraram que seu custo de compilação é mais alto que o observado na estratégia de compilação aqui proposta. Isto ocorre porque é necessário replicar o algoritmo de avaliação, exibido no Algoritmo 6, para cada *kernel* criado, tornando mais extenso o código-fonte a ser compilado e, portanto, aumentando consideravelmente o tempo total de execução.

Na estratégia proposta nesse trabalho (Russo et al., 2014b), assim como na abordagem que utiliza interpretação das soluções candidatas, cada programa é avaliado por uma unidade computacional, de forma a minimizar a ocorrência de divergências de instrução (Augusto e Barbosa, 2013). É criado, portanto, um *work-group* para avaliação de cada indivíduo.

Todos os indivíduos são mapeados e compilados em um único código-fonte. Como cada *work-group* é responsável por avaliar um indivíduo, deve ser introduzido no *kernel* de avaliação uma estrutura de controle que permita selecionar o programa a ser avaliado, de forma semelhante ao que ocorre no Algoritmo 6, no caso da interpretação de modelos. A estrutura de controle utilizada permite a execução de um determinado bloco de código de acordo com o valor de uma variável. O valor utilizado é o índice do *work-group* ao qual pertence o elemento de processamento, que é utilizado para identificar qual solução candidata deve ser avaliada.

O *kernel* de avaliação utilizado na metodologia de compilação dos modelos candidatos é semelhante àquele utilizado na interpretação (Algoritmo 6). A diferença se encontra na linha 9, onde ao invés de executar um interpretador, é utilizado um procedimento para selecionar entre os programas compilados, como mostrado no Algoritmo 8. Neste algoritmo $Programa_k$ corresponde ao programa gerado pelo mapeamento do indi-

víduo $k$ e o parâmetro $X$ é um registro de treinamento. Se o programa não tiver sido mapeado com sucesso, não haverá uma cláusula *Caso* para ele e, portanto, será selecionado o caso padrão. Neste caso, a aptidão é dada pelo valor constante MAX, que representa a pior aptidão possível. Este procedimento é repetido para cada registro de treinamento e a aptidão é dada pela Equação 2.2.

**Algoritmo 8:** Processo de avaliação de uma solução candidata identificada por seu *work-group* para um dado $X$.

```
1  início
2      programa_id ← get_group_id(0);
3      selecione programa_id faça
4          caso 0
5              retorna Programa_0(X);
6          fim caso
7          caso 1
8              retorna Programa_1(X);
9          fim caso
10         ⋮
11         caso n-1
12             retorna Programa_{n-1}(X);
13         fim caso
14         caso padrão
15             retorna MAX;
16         fim caso
17     fim selec
18 fim
```

## 4.2 Paralelismo de todo o processo de busca

Ao invés de paralelizar somente a etapa de avaliação, é possível também realizar todas as etapas do algoritmo de busca de forma paralela, como proposto por Pospichal et al. (2011). Esta abordagem tem a vantagem de não necessitar de transferências de memória entre o hospedeiro e o dispositivo OpenCL a cada iteração do algoritmo, uma vez que

todo o processo ocorre nos *kernels*. Na etapa de avaliação foi utilizada a interpretação dos modelos candidatos, como estratégia descrita anteriormente.

A única alteração neste passo do processo de otimização é que o *kernel* de avaliação, além da execução dos modelos candidatos, agora engloba também o procedimento de mapeamento. A paralelização das outras etapas é descrita a seguir.

A Figura 4.3 ilustra o fluxograma do algoritmo inicialmente proposto, no qual são exibidos os pontos em que ocorrem sincronismos globais. Os sincronismos são necessários para que os dados gerados em uma etapa sejam acessíveis por todos os *work-items* no passo seguinte.

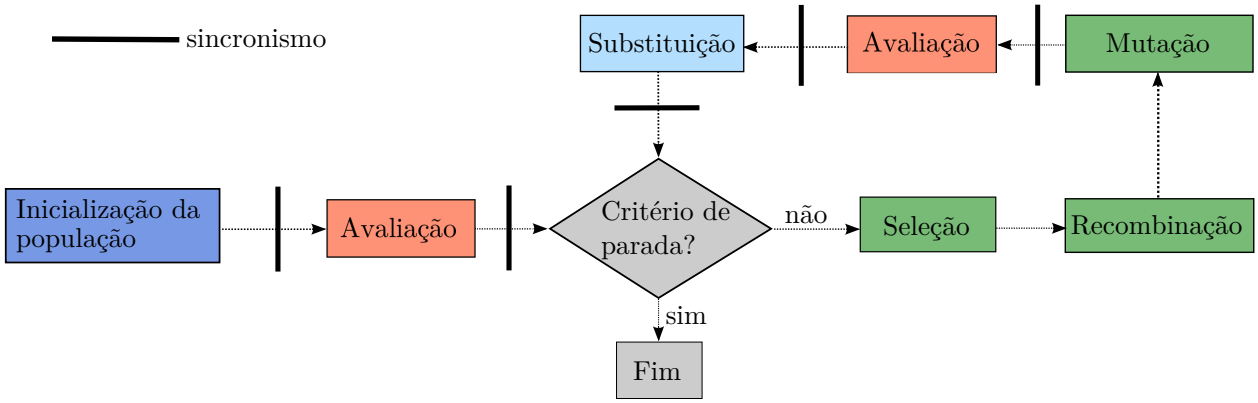



Figura 4.3: Fluxograma de um AG com indicação da ocorrência de sincronismos.

Inicialmente, foram criados quatro *kernels*: um para inicialização da população, o segundo englobando as etapas de seleção, mutação e recombinação, o terceiro responsável pela avaliação e o quarto para realizar a substituição da população.

A geração de números aleatórios, crucial para o sucesso de um algoritmo estocástico, foi realizada utilizando a biblioteca Random123[6], cujo funcionamento está descrito em Salmon et al. (2011). Cada *work-item* possui uma semente de geração de números aleatórios distinta, que é acumulada e persiste às execuções dos *kernels*.

A primeira etapa desenvolvida foi a **inicialização da população**, que é naturalmente paralela. Neste caso, é instanciado um *work-item* para a inicialização de cada indivíduo. O identificador global do *work-item*, *tid*, é utilizado para indexar o indivíduo na população. É executado um laço que passa por todas as posições do genótipo do indivíduo, sendo cada valor gerado aleatoriamente como 0 ou 1. O Algoritmo 9 mostra o

[6]http://www.thesalmons.org/john/random123/releases/1.06/docs/index.html

pseudo-código para este *kernel*. A função *rand*() retorna um número inteiro aleatório.

**Algoritmo 9:** *Kernel* de inicialização da população.

```
1 início
2     tid ← get_global_id(0);
3     lid ← get_local_id(0);
4     para i ← 0  até  tamanho_individuo faça
5         pop[tid].genotipo[i] ← rand() % 2;
6     fim para
7 fim
```

Para o segundo *kernel*, responsável pelas etapas de **seleção**, **recombinação** e **mutação**, foram consideradas três possibilidades, descritas a seguir. No primeiro caso, foram criados $tamanho_{populacao}/2$ *work-items*, sendo cada um responsável por selecionar dois pais e gerar dois filhos, realizando *crossover* e mutação. A segunda possibilidade é utilizar um *work-item* para cada indivíduo da população e 2 itens por unidade de computação. O primeiro *work-item* de trabalho é responsável por selecionar os 2 pais; logo em seguida há um sincronismo, para que o segundo *work-item* tenha acesso (via memória local) aos indivíduos selecionados. Em seguida cada *work-item* realiza sua parte da recombinação e a mutação do indivíduo resultante. A última alternativa considerada é semelhante à segunda, porém utilizando $N$ *work-items* por unidade de computação. Neste caso, os *work-items* pares realizam a seleção dos dois pais e também sorteiam se haverá ocorrência de recombinação. É feito então um sincronismo para que todos os *work-items* tenham acesso (via memória local) aos indivíduos selecionados.

Experimentos preliminares mostraram que a terceira abordagem é a que apresenta melhor desempenho computacional, uma vez que permite utilizar melhor a distribuição de elementos de processamento típica do dispositivo utilizado. Entretanto, esta implementação exige o armazenamento de $N$ indivíduos em memória local e, para os experimentos realizados neste trabalho, a quantidade de memória do dispositivo utilizado foi insuficiente para atender a este requisito. Desta forma, foi adotada a segunda abordagem, que apresentou o segundo melhor desempenho computacional.

O pseudo-código para este *kernel* é mostrado no Algoritmo 10. Para geração da nova população, são executadas $tamanho_{populacao}$ seleções, cada uma desempenhada por

um par de *work-items* trabalhando em conjunto. O primeiro *work-item* de cada par realiza a seleção dos pais e decide aleatoriamente se o operador de recombinação será aplicado. Caso decida-se aplicar a recombinação, é sorteado também o ponto de corte do *crossover*. Depois é executado um sincronismo para que o segundo *work-item* tenha acesso aos pais selecionados, bem como aos dados da recombinação. Após este passo, os dois *work-items* cooperam na execução da recombinação e mutação. Finalmente, os indivíduos gerados são copiados para a estrutura de dados da nova população.

**Algoritmo 10:** *Kernel* para seleção, recombinação e mutação.

```
1  início
2      tid ← get_global_id(0);
3      lid ← get_local_id(0);
4      se lid = 0 então
5          pais[0] ← pop[torneio(pop, tid)];
6          pais[1] ← pop[torneio(pop, tid + 1)];
7          aleatorio ← u_rand();
8          se aleatorio < TAXA_DE_RECOMBINACAO então
9              recombinar ← VERDADEIRO;
               pontoCrossOver ← rand() % tamanho_individuo;
10         fim se
11     fim se
12     sincronismo_local();
13     recombinacao(pais, filhos, recombinar, lid, pontoCrossOver);
14     mutacao(filhos[lid], TAXA_DE_MUTACAO);
15     novaPopulacao[tid] ← filhos[lid];
16 fim
```

A abordagem de seleção utilizada foi o **torneio**, no qual para cada indivíduo selecionado é promovida uma competição entre $tamanho_{torneio}$ indivíduos tomados aleatoriamente, sendo escolhido aquele que possuir a maior aptidão. Ao invés de selecionar todos os oponentes aleatoriamente, o torneio é aqui iniciado com o indivíduo correspondente ao identificador global do *work-item*. Desta forma, garante-se que todos os indivíduos da população tenham chance de participar do processo seletivo. O restante dos indivíduos é selecionado aleatoriamente. O pseudo-código para o torneio é mostrado no Algoritmo 11.

**Algoritmo 11:** Procedimento de seleção via torneio.

**Entrada**: populacao:vetor de individuo, participante:inteiro

**Saída**: vencedor:inteiro

```
1  início
2      vencedor ← participante;
3      para i ← 0  até  (tamanho_torneio − 1) faça
4          aleatorio ← rand() % tamanho_populacao;
5          se populacao[aleatorio].aptidao > populacao[vencedor].aptidao então
6              vencedor ← aleatorio;
7          fim se
8      fim para
9      retorne vencedor;
10 fim
```

Após a seleção dos dois indivíduos pais, são aplicados os operadores genéticos de **recombinação** e **mutação**. A recombinação cruza a informação dos genótipos dos dois pais, enquanto a mutação realiza alterações bit-a-bit aleatórias ao longo do genótipo de cada indivíduo. Os pseudo-códigos para os procedimentos de recombinação e mutação, acessados pelo *kernel* de seleção, são mostrados nos Algoritmos 12 e 13, respectivamente.

**Algoritmo 12:** Procedimento de recombinação.

**Entrada**: recombinar: lógico; lid, pontoCrossover: inteiro

```
1  início
2      se recombinar = VERDADEIRO então
3          para i ← 0  até  pontoCrossover − 1 faça
4              filhos[lid].genotipo[i] ← pais[lid].genotipo[i];
5          fim para
6          para i ← pontoCrossover  até  TAMANHO_INDIVIDUO − 1 faça
7              filhos[lid].genotipo[i] ← pais[1 − lid].genotipo[i];
8          fim para
9      fim se
10     senão
11         filhos[lid] ← pais[lid];
12     fim se
13 fim
```

**Algoritmo 13:** Procedimento de mutação.

```
1  início
2      para i ← 0  até  tamanho_individuo faça
3          aleatorio ← u_rand();
4          se aleatorio <= TAXA_DE_MUTACAO então
5              individuo.genotipo[i] ← (1 − individuo.genotipo[i]);
6          fim se
7      fim para
8  fim
```

Após a execução do *kernel* de seleção, recombinação e mutação, é realizada a avaliação dos novos modelos candidatos. O último passo da iteração do algoritmo é a substituição da população corrente pelos novos indivíduos gerados. O tipo de substituição empregada aqui é a **geracional**, na qual a nova população substitui a geração anterior, passando a ser a população corrente. Entretanto, foi adotado o **elitismo**, abordagem que consiste em manter na população as melhores soluções candidatas da última iteração. O número de soluções mantidas é definida pelo parâmetro *elite*. Dessa forma, as duas populações (a anterior e a nova) são ordenadas por aptidão dos indivíduos e são mantidos

na população corrente somente os melhores modelos, substituindo-se os demais pelos melhores indivíduos da nova geração. Foi criado um *kernel* para realizar a substituição da população, que é enfileirado para execução após o *kernel* de avaliação. Este *kernel* realiza a ordenação do vetor da nova população considerando seu valor de aptidão. Assim, os indivíduos mais aptos ocupam as primeiras posições. A ordenação desenvolvida baseia-se no algoritmo *rank-sort* e o pseudo-código deste *kernel* é mostrado no Algoritmo 14. Para execução deste *kernel* são criados $tamanho_{populacao}$ $workgroups$, cada um contendo somente um *work-item*, responsável por encontrar a posição dos elementos $geracaoAtual[group_{id}]$ e $novaGeracao[group_{id}]$ no vetor ordenado da nova população. Dessa forma, cada *work-item* percorre os vetores das duas populações e contabiliza o número de elementos maiores que aqueles correspondentes ao seu grupo. Finalmente, a nova população ($saida$) é formada pelos $elite$ melhores elementos da geração anterior, preenchendo-se o restante do vetor com os $(tamanho_{populacao} - elite)$ melhores elementos da nova geração.

**Algoritmo 14:** Pseudo-código do *Kernel* de substituição.

**Entrada**: geracaoAtual, novaGeracao, saida: Vetor de indivíduos

```
1  início
2      local_id ← get_local_id(0);
3      group_id ← get_group_id(0);
4      local_size ← get_local_size(0);
5      itemGeracaoAtual ← geracaoAtual[group_id];
6      itemNovaGeracao ← novaGeracao[group_id];
7      pos1 ← 0, pos2 ← 0;
8      para i ← 0  até  tamanho_populacao faça
9          //Conta elementos menores que itemGeracaoAtual
10         se geracaoAtual[j].aptidao > itemGeracaoAtual.aptidao então
11             pos1 ← pos1 + 1;
12         fim se
13         //Conta elementos menores que itemNovaGeracao
14         se novaGeracao[j].aptidao > itemNovaGeracao.aptidao então
15             pos2 ← pos2 + 1;
16         fim se
17     fim para
18     se pos1 < elite então
19         saida[pos1] = geracaoAtual[group_id];
20     fim se
21     se pos2 < tamanho_populacao − elite então
22         saida[pos2 + ELITE] = novaGeracao[group_id];
23     fim se
24 fim
```

### 4.2.1 Melhorias no algoritmo

Com o intuito de identificar no algoritmo elementos que poderiam ser aprimorados para aumentar o desempenho computacional, foi realizada uma pesquisa na literatura, envolvendo propostas de paralelismo de Algoritmos Genéticos, algoritmo de busca da Evolução Gramatical adotada aqui. Considerando que a etapa de avaliação, individualmente, já

apresentava um bom desempenho computacional, buscou-se inspiração para o aperfeiçoamento das demais etapas da busca.

Inicialmente foi estudado o trabalho de Pospíchal et al. (2009), que consistiu na paralelização de um AG de cadeia binária na arquitetura GPGPU, utilizando CUDA. Apesar de ter alcançado um ganho de desempenho expressivo (da ordem de 7000×, considerando o tempo de execução da GPU em relação à execução sequencial em CPU), naquele trabalho foi utilizado um AG baseado em ilhas, no qual a população é segmentada em diferentes partes e permutam indivíduos (migração) em algumas iterações. A abordagem de ilhas não é considerada no presente trabalho, mas é indicada como trabalho futuro.

As alterações realizadas foram baseadas no trabalho de Arora et al. (2010), cujo objetivo foi paralelizar todas etapas de Algoritmos Genéticos binários e reais. No trabalho citado, foi alcançado um *speedup* de até 267×, e suas ideias permitiram o aprimoramento da técnica proposta aqui.

De forma a permitir a comparação dos resultados obtidos aqui com o trabalho de Pospichal et al. (2011), a representação dos indivíduos foi alterada para utilizar diretamente a codificação inteira, ao invés da binária. Desta forma, durante o mapeamento, eliminou-se a conversão do genótipo binário em um vetor de inteiros.

A principal alteração no algoritmo foi a utilização de **granularidade** mais fina nos *kernels*. Nesse contexto, o conceito de granularidade está associado à distribuição de carga de trabalho entre os elementos de processamento. Utilizando granularidade mais fina, as tarefas são divididas em muitas sub-tarefas paralelas. A vantagem observada ao utilizar este nível de paralelismo foi a redução do custo de acesso à memória da GPU, considerando que cada *work-item* deve acessar um espaço de memória reduzido. As mudanças em cada *kernel* são discutidas a seguir.

A geração de números aleatórios também foi alterada, seguindo a mesma abordagem de Arora et al. (2010), que consiste em utilizar vários geradores Park-Miller (Park e Miller, 1988). Este gerador funciona como descrito na Equação 4.3, na qual $r_{k+1}$ é o próximo número aleatório e $r_k$ é o último gerado. Para um dado problema, é possível saber o número máximo de *work-items* que irão gerar números aleatórios (digamos, $M$). De posse dessa informação, é criado um vetor com $M$ posições, contendo sementes gera-

das pelo hospedeiro no início do processo. Para a geração dos números, cada *work-item* acessa uma posição específica deste vetor, indexada pelo seu identificador. O *work-item* realiza a leitura do valor corrente desta posição, utiliza-o para a geração do próximo número aleatório e finalmente atualiza o vetor com o novo valor gerado. Desta forma, em duas iterações consecutivas os número aleatórios gerados por cada *work-item* tendem a ser distintos.

$$r_{k+1} = (r_k * 16807) mod (2^{31} - 1) \tag{4.3}$$

Mesmo no *kernel* de inicialização da população que, como visto, é bastante simples, foi possível melhorar seu desempenho em cerca de 10×. Ao invés de dividir a população entre *work-items*, foi utilizado um *work-group* para cada indivíduo da população, sendo seus *work-items* responsáveis pela geração de um único elemento do genótipo. Assim, são criados $tamanho_{populacao}$ *work-groups*, cada um com $tamanho_{individuo}$ *work-items*. O Algoritmo 15 mostra esta alteração. É importante ressaltar que cada posição do genótipo agora pode assumir valores entre 0 e 255, uma vez que foi adotada a representação inteira. O procedimento $rand()$ recebe o valor corrente da semente de geração de números aleatórios e calcula o próximo número utilizando a Equação 4.3. Esta função também atualiza o valor da variável *semente*.

**Algoritmo 15:** *Kernel* de inicialização da população.

```
1 início
2     gid ← get_group_id(0);
3     lid ← get_local_id(0);
4     semente ← D_sementes[tid];
5     pop[gid].genotipo[lid] ← rand(semente) % 256;
6     D_sementes[tid] ← semente;
7 fim
```

O *kernel* de seleção, recombinação e mutação foi separado em dois outros: um responsável pela seleção; e outro para recombinação e mutação. Isto se deu pela diferença de granularidade entre as etapas. Na seleção são utilizados $tamanho_{populacao}$ *work-groups*, cada um contendo $tamanho_{individuo}$ *work-items*. No *kernel* de recombinação e mutação são utilizados $(tamanho_{populacao}/2)$ grupos, contendo $tamanho_{individuo}$ *work-items* em cada um

deles.

Na proposta inicial para a seleção, um único *work-item* era responsável pela seleção de dois indivíduos, incluindo a realização do torneio e a cópia dos indivíduos gerados para a nova população. Observou-se que a cópia dos indivíduos realizada desta forma contribui para o aumento do tempo de execução, pois cada elemento de processamento realizava muitos acessos à memória, enquanto havia outros elementos de processamento ociosos. Assim foi utilizado um *work-group* para a seleção de cada indivíduo, contendo um *work-item* para cada posição do genótipo. O pseudo-código para este *kernel* é mostrado no Algoritmo 16.

Neste algoritmo, os *work-items* de 0 até $tamanho_{torneio}$, geram números aleatórios que são utilizados para selecionar os participantes do torneio, os quais são armazenados em memória local. Então, um procedimento de sincronismo local é executado para que os participantes sejam acessíveis por todo o grupo. Depois, o item de índice 0 percorre o vetor de participantes e seleciona aquele que possui a maior aptidão. Finalmente, cada *work-item* copia uma posição do genótipo para o indivíduo correspondente do vetor da nova população.

**Algoritmo 16:** *Kernel* para a etapa de seleção.

```
1  início
2      tid ← get_global_id(0);
3      lid ← get_local_id(0);
4      semente ← D_sementes[tid];
5      se lid < tamanho_torneio então
6          aleatorio ←rand(semente) % tamanho_populacao;
7          individuos_local[lid] ← populacao[aleatorio];
8      fim se
9      sincronismo_local();
10     se lid = 0 então
11         melhor_local ←ObtemMelhor(individuos_local);
12     fim se
13     sincronismo_local();
14     novaPopulacao[gid].genotipo[lid] ← melhor_local.genotipo[lid];
15     D_sementes[tid] ← seed;
16 fim
```

As etapas de recombinação e mutação são desempenhadas pelo mesmo *kernel*. A granularidade dessas etapas foi também modificada, de acordo com Arora et al. (2010). São criados $tamanho_{populacao}$ *work-groups*, cada um responsável pela recombinação e mutação de um par de indivíduos. Cada grupo possui $tamanho_{individuo}$ *work-items*, de modo que cada um é responsável pelo processamento correspondente a uma posição do genótipo. Um grupo com identificador *gid* realiza a recombinação e mutação dos indivíduos de índice $(2 \times gid)$ e $(2 \times gid + 1)$, selecionados pelo *kernel* anterior. O pseudo-código deste *kernel* é mostrado no Algoritmo 17.

O item de índice 0 decide aleatoriamente se será aplicada a recombinação e, em caso positivo, sorteia também o ponto de corte no genótipo. Essas duas informações são armazenadas em memória local e, após esse passo, ocorre um sincronismo local. A recombinação e a mutação ocorrem como descritas anteriormente, com a diferença de que cada *work-item* é agora responsável por uma única posição do genótipo.

**Algoritmo 17:** *Kernel* para recombinação e mutação.

```
1  início
2      se lid = 0 então
3          se u_rand(semente) <= TAXA_DE_RECOMBINACAO então
4              recombinar ← VERDADEIRO;
5              pontoCrossOver ← rand() % tamanho_individuo;
6          fim se
7      fim se
8      sincronismo_local();
9      se recombinar então
10         se lid > ponto então
11             bitFilho1 ← novaPopulacao[2 × gid + 1].genotipo[lid];
12             bitFilho2 ← novaPopulacao[2 × gid].genotipo[lid];
13         fim se
14         senão
15             bitFilho1 ← novaPopulacao[2 × gid].genotipo[lid];
16             bitFilho2 ← novaPopulacao[2 × gid + 1].genotipo[lid];
17         fim se
18     fim se
19     senão
20         bitFilho1 ← newPop[2 × gid].genotipo[lid];
21         bitFilho2 ← newPop[2 × gid + 1].genotipo[lid];
22     fim se
23     se u_rand(semente) <= TAXA_DE_MUTACAO então
24         bitFilho1 ← rand(semente)%256;
25     fim se
26     se u_rand(semente) <= TAXA_DE_MUTACAO então
27         bitFilho2 ← rand(semente)%256;
28     fim se
29     novaPopulacao[2 × gid].genotipo[lid] ← bitFilho1 ;
30     novaPopulacao[2 × gid + 1].genotipo[lid] ← bitFilho2;
31 fim
```

Finalmente, o *kernel* de substituição teve sua granularidade também alterada, para que cada posição do genótipo fosse manipulada por um *work-item*. A posição relativa de um indivíduo na população continua a ser computada por somente um *work-item*, entretanto, a cópia dos indivíduos ocorre utilizando um *work-item* para cada posição do genótipo.

# 5 Experimentos computacionais

## 5.1 Introdução

Para análise do desempenho computacional da EG foram realizados quatro experimentos computacionais, nos quais a medida de desempenho considerada foi o tempo de execução. Em todos os casos, o tempo de execução foi medido utilizando o "tempo de relógio de parede" (do inglês *wall-clock time*), que consiste no tempo total decorrido entre o início e término do algoritmo. Desta forma, considera-se não apenas o tempo de processamento, mas também os custos adicionais como transferências de dados e compilação dos programas OpenCL. Nos casos em que uma GPU foi utilizada, foi medido também o tempo de execução dos *kernels* separadamente do tempo total de execução, utilizando funções de perfilação fornecidas pelo OpenCL.

Em todos os experimentos, a aptidão dos indivíduos foi calculada utilizando ponto flutuante de precisão simples (variáveis do tipo *float*), pois em experimentos preliminares observou-se que esta precisão seria suficiente, considerando os modelos utilizados para regressão. Em trabalhos futuros poderá ser considerada a utilização de ponto flutuante de precisão dupla (variáveis do tipo *double*), sobretudo para análise do impacto desta escolha no desempenho computacional de diferentes arquiteturas.

Os dois primeiros experimentos realizados permitiram comparar o desempenho da EG considerando a utilização de interpretação ou compilação dos modelos candidatos. Nestes casos, foram utilizadas as metodologias em que somente a etapa de avaliação é realizada em paralelo, respectivamente utilizando interpretação ou compilação dos indivíduos.

Os dois últimos experimentos tiveram como objetivo medir a aceleração (*speedup*) ao utilizar GPUs, em relação à execução sequencial da técnica em CPU. Como a implementação com OpenCL permite a execução do código também em CPU, foram realizados testes utilizando mais de um núcleo de processamento, além do caso sequencial. Nestes experimentos, foi utilizada a metodologia em que todos os passos da EG são realizados

em paralelo.

Em todos os experimentos foram utilizados problemas de regressão simbólica e o tempo de execução foi dado pela média entre 30 execuções independentes.

O ambiente computacional foi composto por um PC (Computador Pessoal, do inglês *Personal Computer*) contendo um processador Intel Core I3 3240 (3.4GHz) e 1 GPU Nvidia GTX-650 TI (768 núcleos de processamento de 928MHz e 2GB de memória global) e 4GB de memória principal. Foi utilizado também um notebook com um processador Intel Core I5 3210M (2.5GHz) e 6GB de memória principal (sem GPU). Ambas as máquinas utilizaram o sistema operacional Ubuntu 13.04 versão 64 bits.

## 5.2 Experimento I

O objetivo dos Experimentos I e II foi comparar o desempenho da EG utilizando interpretação ou compilação dos modelos candidatos. No primeiro experimento, o número de variáveis do modelo foi mantido constante e variou-se o número de registros de treinamento.

O experimento consistiu na regressão simbólica da Equação 5.1, a partir de dados igualmente espaçados no intervalo $[-1, 1]$. Foram realizadas 30 execuções independentes para cada conjunto de treinamento, cujos tamanhos foram variados entre 10 e $10^7$ registros. O limite para o número máximo de registros foi definido pela quantidade de memória global disponível no dispositivo utilizado. Os parâmetros do algoritmo são mostrados na Tabela 5.1, adotados de acordo com (Russo et al., 2014b).

$$f_1(x) = x^4 + x^3 + x^2 + x \tag{5.1}$$

Tabela 5.1: Parâmetros para o Experimento I.

| **Parâmetro** | **Valor** |
|---|---|
| Gerações | 50 |
| Tamanho da população | 500 |
| Tamanho do cromossomo | 128 bits |
| Taxa de mutação | 1% |
| Tipo de mutação | Bit-a-bit |
| Taxa de *crossover* | 70% |
| Seleção | Torneio |
| Tamanho do torneio | 3 |
| Elitismo | 2 indivíduos |

A gramática utilizada foi a seguinte:

$$N = \{\text{<expr>}, \text{<op>}, \text{<val>}, \text{<var>}\}$$

$$\Sigma = \{+, -, *, /, 1, x\}$$

$$S = \text{<expr>}$$

$$R = \text{<expr> ::= <expr> <expr> <op> } \mid \text{<var> } \mid \text{<val>}$$

$$\text{<op> ::= } + \mid - \mid * \mid /$$

$$\text{<var> ::= } x$$

$$\text{<val> ::= } 0 \mid 1$$

Neste experimento, somente a GPU foi utilizada para a execução dos *kernels* OpenCL, uma vez que os custos adicionais de compilação dos indivíduos tornam inviável a execução da técnica em CPU, considerando os tamanhos de conjuntos de treinamento utilizados aqui.

As medidas de tempo do experimento são ilustrados na Figura 5.1. O gráfico apresenta no eixo horizontal a variação do tamanho das entradas e o respectivo tempo de execução no eixo vertical. Para conjuntos de tamanho até $4 \times 10^6$, o desempenho computacional da estratégia de interpretação das soluções candidatas é superior. Quando o número de registros aumenta, então a compilação passa a ser vantajosa, pois o custo adicional de compilação é compensado pela execução mais rápida das avaliações. Para $10^7$ registros, a estratégia de compilação dos programas candidatos é $1,58\times$ mais rápida do que avaliar os modelos utilizando um interpretador.

Este fato é confirmado pelo gráfico mostrado na Figura 5.2, que ilustra o tempo de execução somente da etapa de avaliação, para as duas metodologias testadas. Este tempo foi medido utilizando funções disponibilizadas pelo OpenCL, que permitem obter o tempo de execução de cada *kernel* separadamente. Neste gráfico, observa-se que o tempo de execução da etapa de avaliação é diretamente proporcional ao número de registros de treinamento, nas duas abordagens. Além disso, a taxa de crescimento do tempo de execução na abordagem que utiliza um interpretador é superior e, dessa forma, para problemas com grandes quantidades de dados, a adoção dessa estratégia de paralelismo torna-se desvantajosa quando comparada à metodologia de compilação.

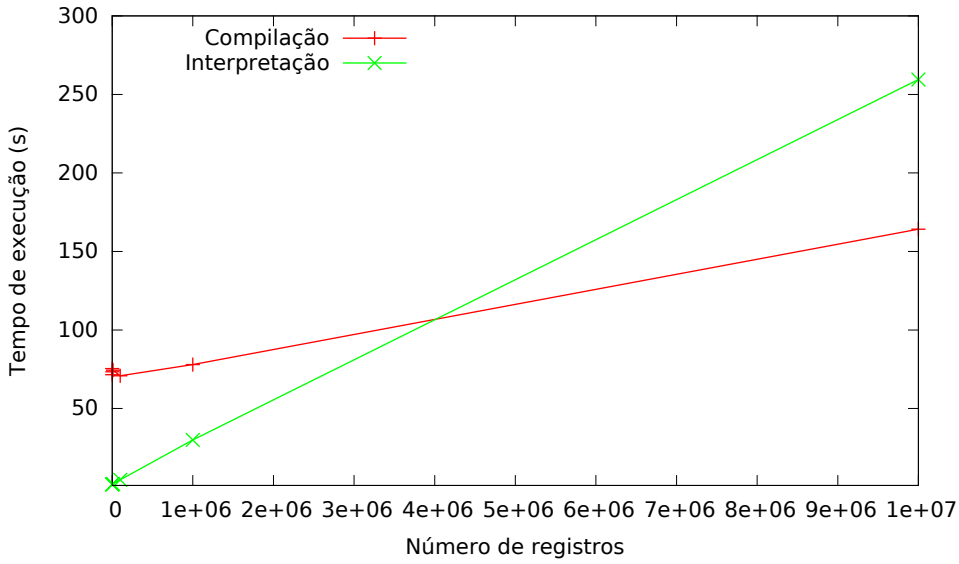



Figura 5.1: Tempo total de execução em função do número de registros (Russo et al., 2014b).

Nas Figuras 5.3 e 5.4 os dois gráficos são exibidos em escala logarítmica, para destacar o comportamento das duas abordagens para pequenos conjuntos de registros. Nestes gráficos podemos confirmar o fato de que para pequenos conjuntos de treinamento, a estratégia de interpretação é mais vantajosa que a de compilação, pois seu tempo de execução é inferior para a maioria dos conjuntos de treinamento testados, tornando-se superior apenas no último. No gráfico da Figura 5.4 podemos observar que os tempos de avaliação sofreram pequenas variações para os conjuntos de treinamento iniciais, até $10^5$ registros. A partir do teste com $10^6$ registros, entretanto, nota-se que o tempo de avaliação aumenta consideravelmente, com maior taxa de crescimento no caso da abordagem de interpretação, justificando o aumento do tempo total de execução observado no gráfico

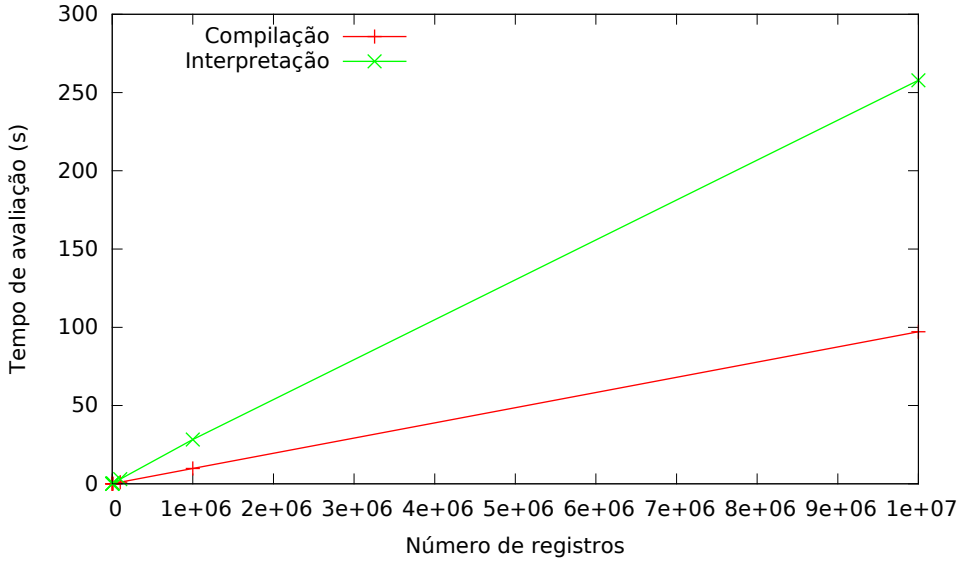



Figura 5.2: Tempo de execução da avaliação em função do número de registros (Russo et al., 2014b).

da Figura 5.3.

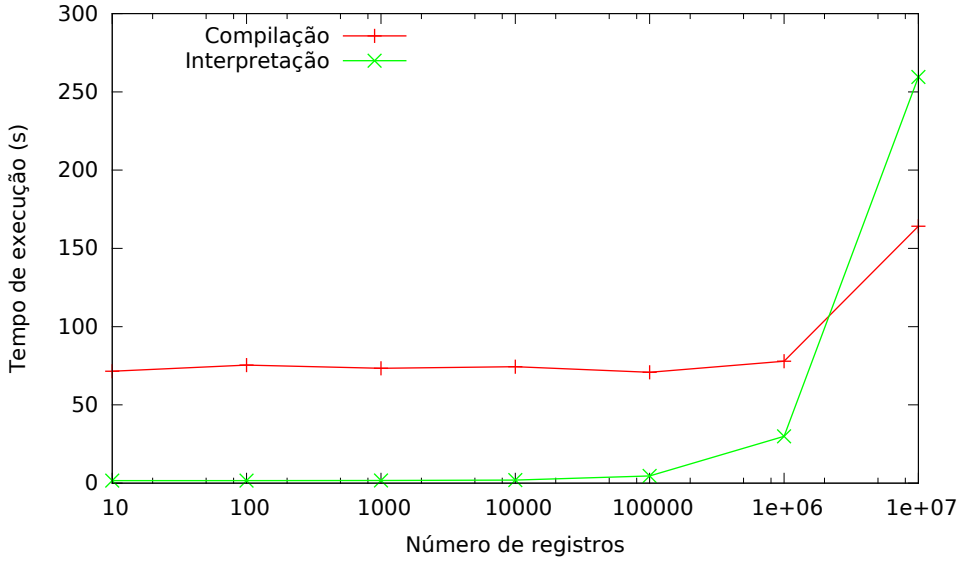



Figura 5.3: Tempo total de execução em função do número de registros (escala logarítmica).

Neste experimento foi possível inferir corretamente a função dada pela Equação 5.1 em todos os casos testados.

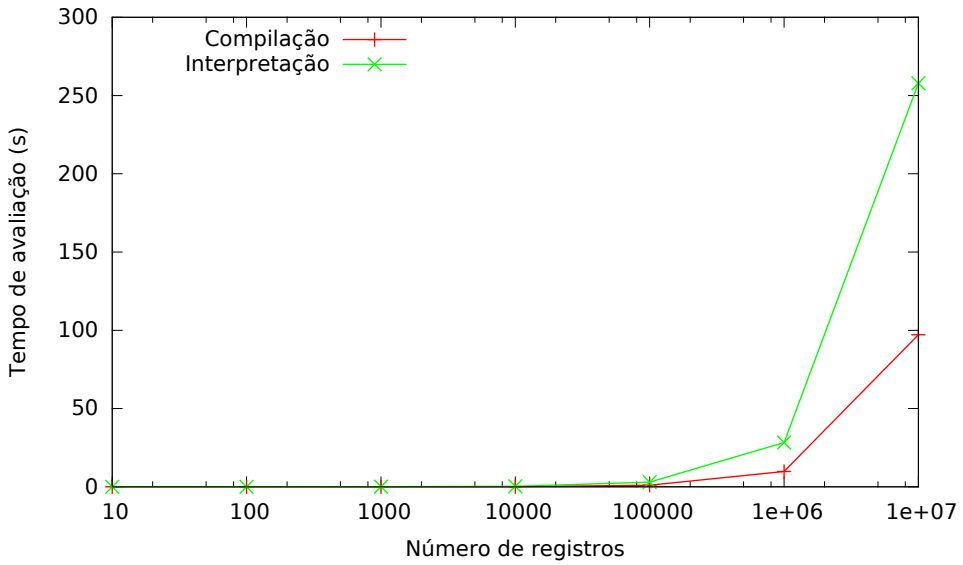



Figura 5.4: Tempo de execução da avaliação em função do número de registros (escala logarítmica).

## 5.3 Experimento II

Este experimento também teve como objetivo a comparação de desempenho entre as abordagens de interpretação e compilação de modelos candidatos na EG. Diferentemente do experimento anterior, no qual foi variado o tamanho das bases de dados, aqui a dimensão do modelo foi variada entre 1 e 15. Foram utilizadas $10^6$ registros de treinamento com valores igualmente espaçados no intervalo $[-1, 1]$ e 30 execuções independentes foram realizadas para cada número de dimensões. O limite para o número de dimensões do modelo foi dado pela quantidade de memória global disponível no dispositivo utilizado. O modelo a ser obtido é o apresentado na Equação 5.2.

$$f_2(x_1, x_2, ..., x_n) = \left( \sum_{k=1}^{n} x_k \right)^2 \tag{5.2}$$

Os parâmetros do experimento são mostrados na Tabela 5.2, adotados de acordo com (Russo et al., 2014b).

A gramática utilizada neste experimento varia de acordo com o número de dimensões utilizadas, limitando, desta forma, o espaço de busca às variáveis presentes no modelo. Para o caso de um conjunto de treinamento com todas as 15 dimensões, a seguinte gramática foi utilizada:

Tabela 5.2: Parâmetros para o Experimento II.

| **Parâmetro** | **Valor** |
|---|---|
| Gerações | 50 |
| Tamanho da população | 500 |
| Tamanho do cromossomo | 128 bits |
| Taxa de mutação | 1% |
| Tipo de mutação | Bit-a-bit |
| Taxa de *crossover* | 70% |
| Seleção | Torneio |
| Tamanho do torneio | 3 |
| Elitismo | 2 indivíduos |

$N$ =<expr>, <op>, <val>, <var>

$\Sigma = \{+, -, *, /, 1, x\}$

$S$ =<expr>

$R$ =<expr> ::= <expr> <expr> <op> | <var> | <val>

<op> ::= + | − | ∗ | /

<var> ::= $x1$ | $x2$ | $x3$ | $x4$ | $x5$ | $x6$ | $x7$ | $x8$ |

$x9$ | $x10$ | $x11$ | $x12$ | $x13$ | $x14$ | $x15$

<val> ::= 1.0

Os resultados do experimento são ilustrados na Figura 5.5. O gráfico apresenta o número de dimensões no eixo horizontal, o respectivo tempo de execução no eixo vertical, e duas curvas; indicando comportamentos para os casos de compilação e interpretação.

Pode-se observar neste gráfico que o desempenho computacional da estratégia de interpretação foi superior em todos os casos testados. Portanto sua adoção é vantajosa para problemas com várias dimensões, considerando o número de registros testado ($10^6$). Para determinar a melhor estratégia para problemas com várias dimensões e conjuntos de treinamento maiores do que os considerados aqui, outros testes devem ser realizados.

Neste experimento, com os parâmetros utilizados, não foi possível obter um modelo acurado para a Equação 5.2.

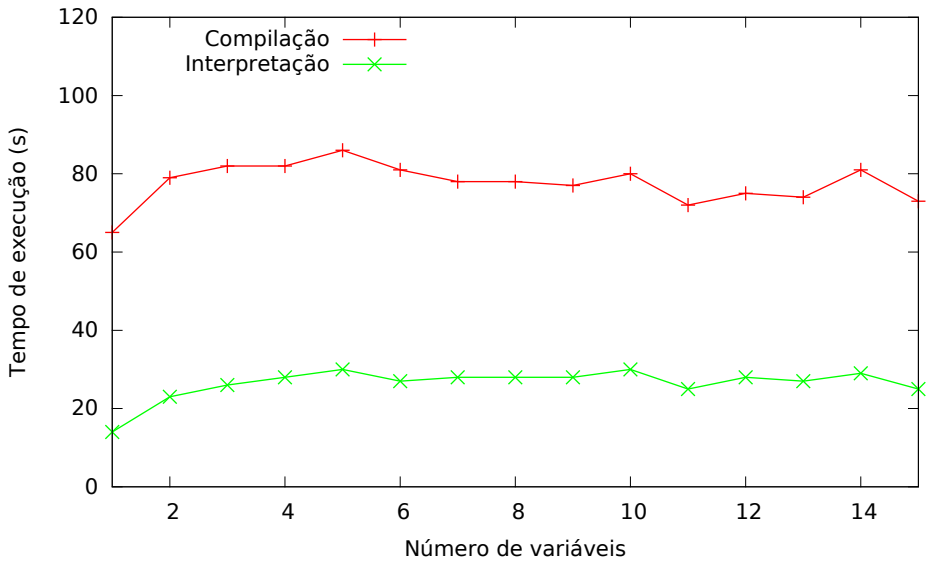



Figura 5.5: Gráfico do tempo de execução em função do número de variáveis para os casos de compilação e interpretação de modelos candidatos (Russo et al., 2014b).

## 5.4 Experimento III

O objetivo dos Experimentos III e IV foi analisar o desempenho da metodologia de paralelismo de todas as etapas da EG. No Experimento III, a implementação utilizada é a descrita na Seção 4.2, em que um genótipo binário é adotado.

O experimento consistiu na regressão simbólica da Equação 5.1, a partir de dados igualmente espaçados no intervalo $[-1, 1]$. Foram realizadas 30 execuções independentes para cada conjunto de treinamento, cujos tamanhos foram variados entre 10 e $10^6$ registros. Para execução da técnica, foram utilizados os processadores Core I3 e Core I5, além da GPU. No caso da execução em CPUs, foram utilizados as 4 *threads* de processamento disponíveis.

Uma população de tamanho 500 foi evoluída por 50 gerações, utilizando *crossover* com probabilidade de 70%, mutação bit-a-bit com probabilidade de 1%, seleção por torneio (entre 5 indivíduos) e elitismo de 2 indivíduos. Estes parâmetros foram adotados de acordo com (Russo et al., 2014a). A gramática utilizada foi a seguinte:

$$N = \{\text{<expr>}, \text{<op>}, \text{<val>}, \text{<var>}\}$$

$$\Sigma = \{+, -, *, /, 1, x\}$$

$$S = \text{<expr>}$$

$$R = \text{<expr>} ::= \text{<expr>}\ \text{<expr>}\ \text{<op>} \mid \text{<var>} \mid \text{<val>}$$

$$\text{<op>} ::= + \mid - \mid * \mid /$$

$$\text{<var>} ::= x$$

$$\text{<val>} ::= 0 \mid 1$$

As medidas de *speedup* deste experimento são apresentados na Figura 5.6. Em cada caso, foi calculado o *speedup* em relação à execução da EG utilizando um núcleo do processador Intel Core I5 3210M. O maior *speedup* alcançado foi de 66.3×, com a execução em GPU utilizando $10^6$ registros. Na Figura 5.7 é mostrado o gráfico do tempo de execução em relação ao número de registros de treinamento, no qual podemos observar que a adoção da GPU foi vantajosa, em relação à execução em CPU, sobretudo no caso em que somente um núcleo de processamento foi utilizado.

Neste experimento foi possível inferir corretamente a função dada pela Equação 5.1, em todos os casos testados.

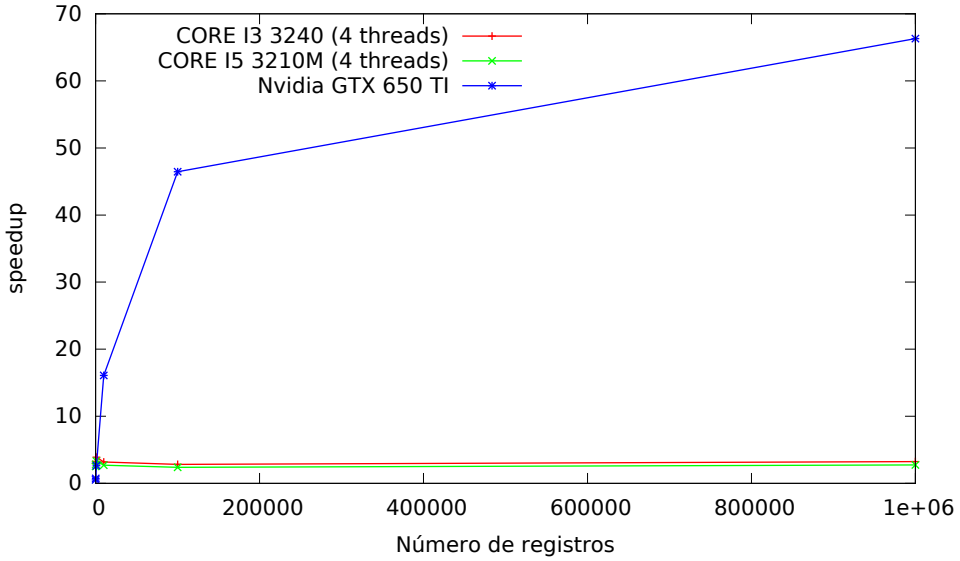



Figura 5.6: Speedup em função do número de registros (Russo et al., 2014a).

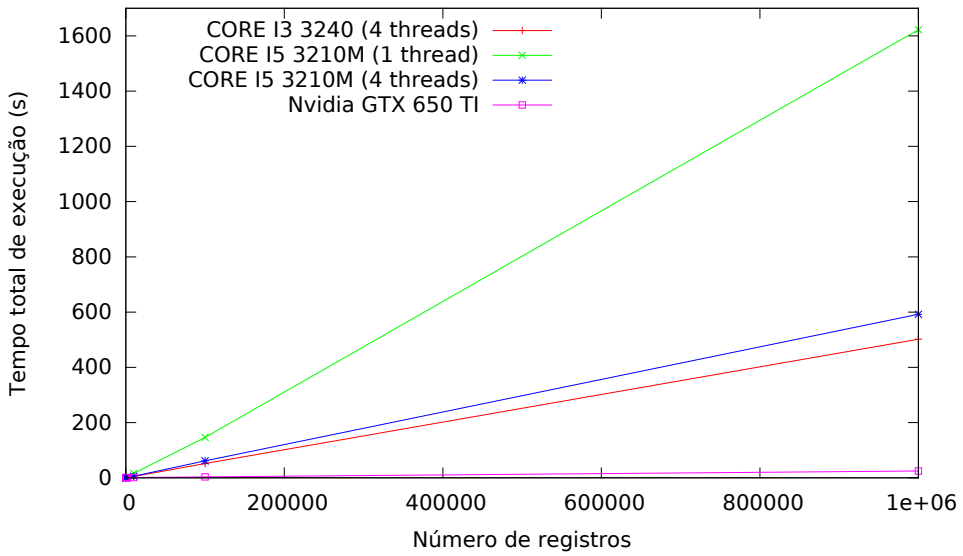



Figura 5.7: Tempo total de execução em função do número de registros.

## 5.5 Experimento IV

O último experimento realizado foi baseado em Pospichal et al. (2011), e consistiu também na regressão simbólica da função $f_1$ (Equação 5.1) a partir de dados igualmente espaçados no intervalo $[-1, 1]$. Neste caso, entretanto, além da variação do tamanho do conjunto de treinamento, testes foram realizados variando-se o tamanho da população. Foram realizadas 30 execuções independentes para cada conjunto de treinamento e tamanho de população. Os parâmetros utilizados são enumerados na Tabela 5.3. Neste experimento utilizou-se a mesma gramática do experimento anterior.

Tabela 5.3: Parâmetros para o Experimento IV.

| **Parâmetro** | **Valor** |
|---|---|
| Gerações | 50 |
| Tamanhos da população | 4, 8, 16, 32, 64 |
| Tamanhos do conjunto de treinamento | 128, 256, 1280, 2560 |
| Tamanho do cromossomo | 128 inteiros |
| Taxa de mutação | 1% |
| Tipo de mutação | Sorteio de inteiro |
| Taxa de *crossover* | 90% |
| Seleção | Torneio |
| Tamanho do torneio | 3 |
| Elitismo | 2 indivíduos |

A metodologia utilizada foi o paralelismo de todas as etapas da EG, com as alterações indicadas na Sub-seção 4.2.1. Foi calculado o *speedup* da execução da técnica

utilizando a GPU em relação à execução sequencial do algoritmo no processador Intel Core I5. No caso da execução em GPU, foram realizadas duas medidas de tempo: uma considerando o tempo total de execução, e uma outra contabilizando somente o tempo de processamento dos *kernels*, desconsiderando o tempo de inicialização da GPU e de compilação e criação dos *kernels*. O resultado dos dois testes são ilustrados nas Figuras 5.8 e 5.9, respectivamente. Os ganhos máximos de desempenho foram de 40×, no primeiro caso, e de 50×, no segundo.

Os resultados alcançados aqui são superiores aos apresentados por Pospichal et al. (2011), onde foram obtidos *speedups* de 25.9×, no caso do tempo total de execução, e de 39.0×, quando foi considerado somente o tempo de execução dos *kernels*. Em relação ao tempo total de execução da técnica, os resultados aqui obtidos superam aqueles apresentados em Pospichal et al. (2011), tanto na execução em GPU, na qual observou-se uma redução de tempo de cerca de 3×, quanto no caso sequencial, que foi executado em aproximadamente metade do tempo indicado naquele trabalho. O ambiente computacional utilizado por Pospichal et al. (2011) nos experimentos foi composto por um PC contendo um processador Intel Core I7 (3.3GHz) e uma GPU Nvidia GeForce GTX 480, executando o sistema operacional Ubuntu 10.04 versão 64 bits. Para a execução sequencial, somente um núcleo de processamento da CPU foi utilizado. Além disso, a versão sequencial do algoritmo foi implementada em linguagem C.

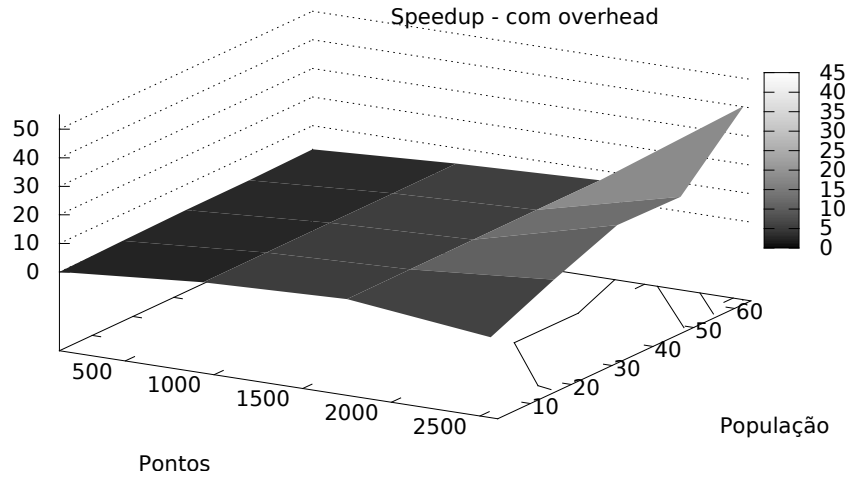



Figura 5.8: Speedup de acordo com a variação do número de pontos e do tamanho da população, considerando o tempo total de execução da GPU.

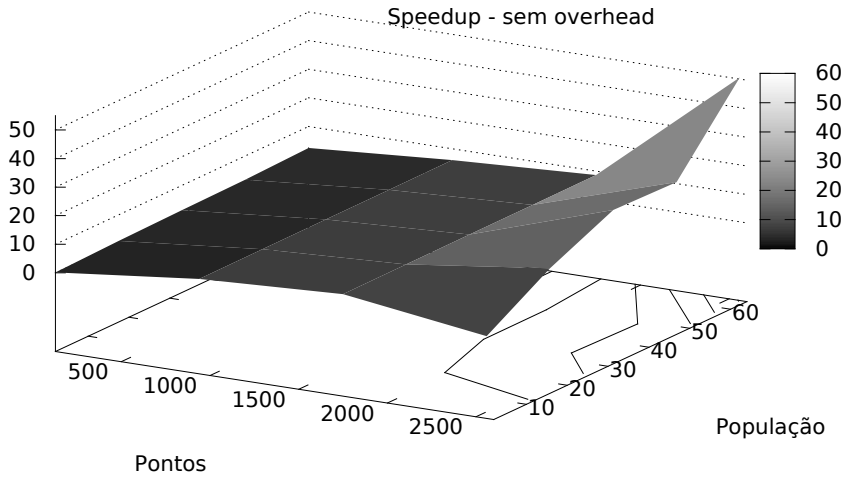



Figura 5.9: Speedup de acordo com a variação do número de pontos e do tamanho da população, considerando somente o tempo de processamento da GPU (sem *overhead*).

Os valores dos parâmetros utilizados foram os mesmos adotados em Pospichal et al. (2011). É importante ressaltar que no trabalho citado, havia uma limitação do número de registros de treinamento, pelo fato de ser utilizada a memória local para seu armazenamento. Neste trabalho, entretanto, poderiam ser utilizados conjuntos consideravelmente maiores, como aqueles adotados nos outros experimentos desenvolvidos aqui. Da mesma forma, os tamanhos de população utilizados foram limitados aos utilizados por Pospichal et al. (2011).

Neste experimento foi possível inferir corretamente a função dada pela Equação 5.1 na maioria dos casos testados.

## 5.6 Discussões complementares

Nos dois primeiros experimentos (Experimento I e Experimento II), o paralelismo adotado é do tipo mestre-escravo, no qual somente a etapa de avaliação é realizada em paralelo na GPU, sendo os demais passos executados sequencialmente em CPU. Além disso, com exceção da etapa de avaliação, as metodologias propostas aqui para interpretação e compilação dos modelos candidatos são idênticas. Em outras palavras, todas as etapas do processo de busca atuam da mesma forma nas duas abordagens (interpretação e compilação).

Isto posto, espera-se que as buscas das duas metodologias apresentem o mesmo

comportamento, isto é, que os modelos gerados ao longo das iterações em cada abordagem sejam os mesmos. Entretanto, após algumas execuções das duas implementações observou-se que a convergência dos algoritmos era diferente e, portanto, os modelos gerados apresentavam discrepâncias. Testes mais detalhados revelaram que soluções candidatas iguais recebiam valores de aptidão diferentes nas duas abordagens e, assim, o processo evolutivo acabava sendo distinto. Finalmente, foi identificado que a diferença no cálculo da aptidão se dava em razão das otimizações realizadas pelo compilador do OpenCL. As otimizações realizadas aumentam a precisão de algumas operações aritméticas, implicando em diferentes valores de aptidão dos indivíduos. Valores distintos de aptidão para a mesma solução candidata podem alterar a etapa de seleção dos indivíduos e, eventualmente, levar à geração de modelos distintos entre as duas abordagens.

A especificação do OpenCL suporta a operação de ponto flutuante FMA(a,b,c) [7] (do inglês, *fused multiply-add*), que realiza a soma do parâmetro $c$ com o produto entre $a$ e $b$ em um único passo, sem arredondar o resultado da operação de multiplicação. A disponibilidade desta função em cada implementação do OpenCL está atrelada à existência de recursos de *hardware* que possibilitem sua execução.

Experimentos mostraram que o compilador do OpenCL substitui a expressão $(a * b) + c$ pela função FMA sempre que possível. Isto ocorre inclusive se a ordem dos operandos estiver invertida, como em $c + (a * b)$. Nestes casos, pode-se observar que a abordagem que utiliza compilação dos indivíduos é beneficiada, uma vez que os códigos-fonte dos modelos são compilados juntamente com o programa OpenCL. Desta forma, o cálculo da aptidão na abordagem que utiliza compilação apresenta maior precisão e, consequentemente, valor distinto do obtido pela avaliação via interpretação.

Tal otimização não é possível na EG utilizando interpretação dos modelos, pois as operações são executadas passo a passo, ocorrendo portanto arredondamento após cada operação aritmética.

Para confirmar esta hipótese, foram desabilitadas as otimizações promovidas pelo compilador. Dessa forma os modelos gerados pelas duas implementações foram iguais, assim como era esperado.

[7] http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/fma.html

É importante destacar que apesar da diferença no valor das aptidões ser pequena (da ordem de $10^{-5}$, utilizando ponto flutuante de precisão simples) para a maioria dos casos, em alguns modelos foram observados erros grandes.

Ao avaliar, por exemplo, o programa candidato

$$programa(x) = \left( \frac{x}{((1+1) * (\boldsymbol{(x + (x * x))} * (1 - x)))} \right)$$

utilizando a estratégia de compilação (com otimizações habilitadas) e assumindo como modelo alvo a (Equação 5.1), então a expressão $(x + (x * x))$ (destacada) será substituída por FMA$(x, x, x)$. Sob esta perspectiva, tem-se que $programa(-1,000117) = -2137,64$. Caso a abordagem de interpretação seja utilizada, ou as otimizações sejam desabilitadas, a função FMA não será utilizada e, assim, obtém-se $programa(-1,000117) = -2137,89$. Observa-se, portanto, uma diferença numérica de $0,25$ entre os resultados obtidos. Apesar de pequena, esta diferença numérica já seria suficiente para influenciar o processo de seleção, alterando o comportamento da busca. Entretanto, este erro é agravado ao calcular a aptidão do indivíduo, utilizando a Equação 2.2. Para este caso, a diferença numérica entre os 2 cálculos de aptidão passa a ser de $1068,89$, sendo o resultado mais preciso numericamente quando as otimizações são habilitadas.

Uma possibilidade que permitiria comparar o desempenho das duas metodologias considerando os efeitos das otimizações do compilador é (i) desligar a pressão de seleção do algoritmo de busca e (ii) aplicar mutação em todos os indivíduos obrigatoriamente, de forma a prevenir otimizações de cache. Desta forma, como todas as decisões ocorrem de forma aleatória e, considerando as mesmas sementes para geração de números aleatórios, os modelos gerados ao longo das iterações do algoritmo serão os mesmos nas duas metodologias comparadas. Entretanto, o item (ii) não é sempre efetivo no caso da Evolução Gramatical, uma vez que podem ocorrer mutações neutras, isto é, que alteram o genótipo sem modificar o fenótipo (programa). Além disso, o objetivo dos experimentos foi comparar o custo computacional da técnica em situações próximas à utilização real da Evolução Gramatical, o que não ocorreria se esta abordagem fosse adotada.

Como citado anteriormente, uma forma de eliminar esta diferença observada no cálculo da aptidão dos indivíduos é executar a técnica desativando-se as otimizações rea-

lizadas pelo compilador. Isto pode ser facilmente alcançado, a partir da utilização de uma $flag$ durante a compilação do programa OpenCL. Entretanto, esta alternativa é inviável, uma vez que aqui pretende-se analisar o desempenho computacional da EG e o desligamento das otimizações contribui para o crescimento do tempo de execução, sobretudo considerando o grande volume de dados analisados.

Por outro lado, testes mostraram que é possível forçar o arredondamento acrescentando uma operação de adição após a ocorrência de uma multiplicação ou divisão, impedindo que o compilador substitua o termo $(a * b) + c$ pela função FMA. Para não modificar o valor da aptidão, foi acrescentada uma adição da constante zero. É importante ressaltar que esta operação de adição é inserida durante a etapa de mapeamento do programa, não alterando portanto o mecanismo de busca.

Desta forma, é possível obter os mesmos valores de aptidão para os indivíduos nas duas metodologias e, portanto, gerar os mesmos modelos ao longo das gerações. Logo, para os experimentos e comparações realizadas aqui, essa estratégia para forçar o arredondamento foi adotada para que o processo de busca fosse igual em ambas as estratégias, mas pode-se obter resultados mais precisos e com desempenho computacional superior sem utilizar essa alternativa.

# 6 Conclusões e trabalhos futuros

Neste trabalho foram estudados e propostos modelos de computação massivamente paralela para aceleração da Evolução Gramatical com OpenCL, de forma a viabilizar a utilização da técnica para problemas com grandes bases de dados. Foram apresentados três modelos: dois em que somente a etapa de avaliação foi paralelizada, e outro no qual todas as etapas da busca são realizadas em paralelo. Também foram consideradas duas possibilidades para avaliação dos indivíduos: avaliação utilizando um interpretador ou através da compilação dos modelos candidatos. Ao total foram realizados quatro experimentos para avaliação do desempenho computacional das propostas.

Os dois primeiros experimentos permitiram a comparação entre o desempenho computacional das estratégias de avaliação via interpretação ou compilação dos modelos candidatos. Os experimentos indicaram que a adoção da compilação para avaliação dos modelos é vantajosa para problemas com grandes conjuntos de treinamento (a partir de $4 \times 10^6$ registros, para o problema testado). Nos demais casos considerados é indicada a utilização de interpretação.

No terceiro experimento foi obtido um ganho de desempenho de até 66.3× para grandes conjuntos de dados (com $10^6$ registros), quando utilizada uma GPU, em relação à execução da técnica em um núcleo de processamento da CPU. No último experimento foram testadas as melhorias promovidas na proposta inicial, sobretudo a utilização de granularidade mais fina nos *kernels*, atingindo-se ganhos de desempenho de até 50×, em relação à execução sequencial em CPU. Neste experimento, tal *speedup* foi alcançado mesmo utilizando poucos dados de treinamento (2560, no máximo). Acredita-se, portanto, que esta abordagem seja superior à primeira testada, incentivando a realização de novos experimentos com problemas mais complexos e/ou maiores bases de dados.

Em trabalhos futuros, problemas mais complexos poderão ser utilizados, tanto para avaliar os ganhos de desempenho obtidos pela adoção de GPUs como aceleradores, quanto para analisar o comportamento das estratégias de interpretação e compilação em cenários em que a avaliação é mais cara computacionalmente. Além disso, como

o OpenCL permite a execução em dispositivos de diferentes arquiteturas e fabricantes, novos experimentos poderão ser realizados contemplando esta possibilidade. Finalmente, na metodologia de compilação, observou-se que a compilação de blocos de indivíduos, ao invés de toda a população em um único passo pode contribuir para a redução do tempo total de execução. Experimentos poderão ser realizados para identificação da melhor estratégia neste caso.

# Referências Bibliográficas


Ando, J.; Nagao, T. **Fast evolutionary image processing using multi-GPUs.** In: SMC, p. 2927–2932. IEEE, 2007.

Arora, R.; Tulshyan, R. ; Deb, K. **Parallelization of binary and real-coded genetic algorithms on GPU using CUDA.** In: IEEE Congress on Evolutionary Computation, p. 1–8. IEEE, 2010.

Augusto, D.; Barbosa, H. Accelerated parallel genetic programming tree evaluation with OpenCL. **Journal of Parallel and Distributed Computing**, v.73, n.1, p. 86–100, 2013.

Bernardino, H. S.; Barbosa, H. J. C. Grammar-based immune programming. **Natural Computing**, v.10, n.1, p. 209–241, 2011.

Buyya, R. High performance cluster computing: Architecture and systems. **Prentice Hall, Upper SaddleRiver, NJ, USA**, v.1, 1999.

Chitty, D. **A data parallel approach to genetic programming using programmable graphics hardware**. In: GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, volume 2, p. 1566–1573. ACM Press, 2007.

Chomsky, N. **Syntactic structures**. Mouton de Gruyter, 2002.

Colorni A., D. M.; V, M. Distributed optimization by ant colonies. **In Proc. of the European Conference on Artificial Life, pages 134-142, Paris, France. Elsevier**, 1991.

Darema, F.; George, D. A.; Norton, V. A. ; Pfister, G. F. A single-program-multiple-data computational model for epex/fortran. **Parallel Computing**, v.7, n.1, p. 11–24, 1988.

Farmer, J.; Packard, N. ; Perelson, A. The immune system, adaptation, and machine learning. **Physica D, 2(1-3):187-204.**, 1986.

Flynn, M. J. Very high-speed computing systems. **Proceedings of the IEEE**, v.54, n.12, p. 1901 – 1909, December 1966.

Fogel, L. J.; Owens, A. J. ; Walsh, M. J. Artificial intelligence through simulated evolution. **John Wiley & Sons, Inc**, 1966.

Gaster, B.; Kaeli, D.; Howes, L. ; Mistry, P. **Heterogeneous Computing with OpenCL**. Morgan Kaufmann Pub, 2011.

Glover, F. W.; Kochenberger, G. A. **Handbook of Metaheuristics**. Hardcover, January 2003.

Gothandaraman, A.; Hinde, R. ; Peterson, G. Comparing hardware accelerators in scientific applications: A case study. **Parallel and Distributed Systems, IEEE Transactions**, v.22, n.1, p. 58 – 68, 2011.

Harding, S.; Banzhaf, W. **Fast genetic programming on GPUs.** In: EuroGP, volume 4445 de **Lecture Notes in Computer Science**, p. 90–101. Springer, 2007.

Holland, J. Adaptation in natural and artificial systems. **University of Michigan Press**, 1975.

Juille, H.; Pollack, J. B. **Massively parallel genetic programming**. In: Advances in Genetic Programming 2, chapter 17, p. 339–358. MIT Press, Cambridge, MA, USA, 1996.

Kennedy, J.; Eberhart, R. Particle swarm optimization. **In Proc. IEEE International Conf. on Neural Networks (Perth, Australia), IEEE Service Center, Piscataway, NJ**, 1995.

Khronos. **The OpenCL specification**. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, 2011. [acessado em Novembro de 2014].

Kish, L. B. End of moore's law: thermal (noise) death of integration in micro and nano electronics. **Physics Letters A**, v.305, n.3, p. 144–149, 2002.

Knuth, D. E. Backus normal form vs. Backus Naur form. **Communications of the ACM**, v.7, n.12, p. 735–736, 1964.

Koza, J. R. Genetic programming: On the programming of computers by means of natural selection. **The MIT Press**, 1992.

Langley, P. **Lessons for the computational discovery of scientific knowledge**. In: Proc. of the Intl. Workshop on Data Mining Lessons Learned, p. 9–12, 2002.

Langdon, W.; Banzhaf, W. **A SIMD interpreter for genetic programming on GPU graphics cards.** In: EuroGP, volume 4971 de **Lecture Notes in Computer Science**, p. 73–85. Springer, 2008.

Michalewicz, Z. **Genetic algorithms and data structures - evolution programs.** Springer, 1996, I-XX, 1-387p.

Montana, D. Strongly typed genetic programming. **Evolutionary computation**, v.3, n.2, p. 199–230, 1995.

O'Neill, M.; Ryan, C. Grammatical evolution. **IEEE Trans. on Evolutionary Computation, 5(4):349-358**, 2001.

O'Neill, M.; Ryan, C. **Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language**, volume 4 de **Genetic programming**. Kluwer Academic Publishers, 2003.

O'Neill, M.; Brabazon, A. Grammatical swarm: The generation of programs by social programming. **Natural Computing**, , n.4, p. 443–462, 2006.

Owens, J. D.; Houston, M.; Luebke, D.; Green, S.; Stone, J. E. ; Phillips, J. C. GPU computing. **Proceedings of the IEEE Volume:96, Issue:5**, 2008.

Pacheco, P. S. **An Introduction to Parallel Programming.** Morgan Kaufmann, 2011.

Park, S. K.; Miller, K. W. Random number generators: Good ones are hard to find. **Commun. ACM**, v.31, n.10, p. 1192–1201, 1988.

Poli, R.; Langdon, W. B. ; McPhee, N. F. **A field guide to genetic programming**. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

Pospíchal, P.; Jaros, J. GPU-based acceleration of the genetic algorithm. **GECCO competition**, 2009.

Pospichal, P.; Murphy, E.; O'Neill, M.; Schwarz, J. ; Jaros, J. **Acceleration of grammatical evolution using graphics processing units: Computational intelligence on consumer games and graphics hardware**. In: Proc. of the Genetic and Evolutionary Computation Conference (GECCO), p. 431–438. ACM, 2011.

Rechenberg, I. Evolutionsstrategie : Optimierung technischer systeme nach prinzipien der biologischen evolution. **Simulationsmethoden in der Medizin und Biologie, Medizinische Informatik und Statistik Volume 8, 1978, pp 83-114**, 1973.

Robilliard, D.; Marion-Poty, V. ; Fonlupt, C. Population parallel GP on the G80 GPU. **In Proc. of the European Conf. on Genetic Programming (EuroGP). LNCS. pps (98-109). Springer.**, 2008.

Russo, Igor L. S.; Bernardino, H. S.; Barbosa ; C., H. J. **Evolução gramatical massivamente paralela com OpenCL via compilação dos modelos candidatos**. In: Ibero Latin American Congress on Computational Methods in Engineering (CILAMCE), 2014.

Russo, Igor L. S.; Bernardino, H. S.; Barbosa ; C., H. J. **Evolução gramatical massivamente paralela**. In: XI Simpósio de Mecânica Computacional e II Encontro Mineiro de Modelagem Computacional, 2014.

Salmon, J. K.; Moraes, M. A.; Dror, R. O. ; Shaw, D. E. **Parallel random numbers: as easy as 1, 2, 3.** In: Lathrop, S.; Costa, J. ; Kramer, W., editors, SC, p. 16. ACM, 2011.

Soares, G. L. Algoritmos genéticos: estudo, novas técnicas e aplicações. **Belo Horizonte: Universidade Federal de Minas Gerais**, 1997.

Talbi, E.-G. **Metaheuristics - From Design to Implementation.** Wiley, 2009, I-XXIX, 1-593p.

Tanenbaum, A. S.; Machado Filho, N. **Sistemas operacionais modernos**, volume 3. Prentice-Hall, 1995.

Whigham, P. A. **Grammatically-based genetic programming**. In: Rosca, J. P., editor, Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications, p. 33–41, 9 July 1995.