8
Resultados Experimentais

Este capítulo apresenta os resultados da execução das metodologias propostas neste trabalho baseadas em compilação em linguagem intermediária e criação de indivíduos em código de máquina de GPUs. Comparamos nossas metodologias com outras metodologias de PG em GPU propostas na literatura: compilação e interpretação. Apresentamos os resultados de desempenho e de qualidade da solução para um conjunto de sete *benchmarks*: duas regressões simbólicas, uma previsão de séries temporais, dois problemas de processamento de imagens, uma classificação e uma regressão booleana.

8.1
Ambiente de Execução

A GPU usada nos experimentos foi uma GeForce GTX TITAN. Esta GPU possui 2.688 CUDA *cores* (em 837 MHz) e 6 GB de RAM (sem ECC) com uma largura de banda da memória de 288,4 GB/s através de um barramento de dados de 384 bits. A GPU GTX TITAN é baseada na arquitetura Kepler da nVidia. Seu pico de desempenho teórico é caracterizado através do uso da instrução *fused multiply-add* (FMA), a qual é contabilizada como sendo uma multiplicação e uma adição de ponto flutuante combinadas. Assim, a GTX TITAN pode atingir um pico de desempenho teórico em precisão simples de ponto flutuante de 4,5 TFLOPs.

Para estudar os ganhos obtidos com o paralelismo maciço da GPU na avaliação da PG, a execução na GPU foi comparada com a execução serial em uma CPU. Para este experimento, utilizamos um núcleo do processador Intel Xeon CPU X5690, com 32 KB de memória cache de dados L1, 1.5 M de cache L2, 12 MB de cache L3 e 24 GB de RAM, executando em 3,46 GHz. O Intel Xeon X5690 pode atingir um pico de desempenho teórico em precisão simples de ponto flutuante de aproximadamente 84 GFLOPs sem utilizar as instruções SSE, tendo um desempenho de aproximadamente 16 GFLOPS/núcleo. Neste experimento, não temos a intenção de comparar o poder computacional do Intel Xeon X5690 com o de uma GPU na execução da PG. Estamos interessados na aceleração que pode ser obtida através da implementação paralela para GPU. O código que executa no processador Intel Xeon, portanto, não foi paralelizado.

As metodologias de PG foram implementadas usando C, CUDA 5.5 e PTX 3.2. Os compiladores utilizados foram o gcc 4.4.7, o nvcc *release* 5.5, V5.5.0 e o ptxas *release* 5.5, V5.5.0. O nível de otimização utilizado nos

experimentos foi -O0. Realizamos experimentos preliminares, apresentados na seção 6.6, e obtivemos melhores resultados para a otimização -O0. Com este nível de otimização não é necessário dispender tempo otimizando o código dos indivíduos que serão executados uma única vez, resultando em um tempo de compilação JIT menor. O tempo de compilação JIT é o principal gargalo da execução da metodologia Pseudo-assembly.

8.2 Benchmarks

Utilizamos sete *benchmarks* comumente empregados em PG que resolvem os seguintes problemas:

1. Regressão simbólica: *Chapéu Mexicano* e *Salutowicz*;
2. Previsão de séries temporais: *Mackey-Glass*;
3. Processamento de imagens: *Filtro Sobel* e *Restauração Salt-and-Pepper*;
4. Classificação: *Detecção de Intrusos na Rede*;
5. Regressão Booleana: *Multiplexador de 20 bits*.

Os resultados experimentais apresentados foram obtidos fazendo-se uma média de 10 experimentos. São fornecidos os tempos em segundos e também as operações de programação genética por segundo (GPops), que têm sido frequentemente empregadas em trabalhos anteriores de PG. Embora o principal foco deste trabalho sejam as acelerações obtidas na execução da avaliação da PG, também discutimos brevemente a qualidade dos resultados produzidos pelas metodologias estudadas.

Utilizamos os sete *benchmarks* para realizar diferentes avaliações. Os seis primeiros *benchmarks* foram utilizados para avaliar instruções com precisão simples de ponto flutuante, enquanto que o último *benchmark* foi utilizado para avaliar instruções booleanas. As regressões simbólicas são aplicações sintéticas que permitem o aumento linear do tamanho das amostras de dados. Utilizamos estas regressões com diferentes tamanhos de entrada para analisar detalhadamente o desempenho das metodologias de PG em GPU existentes na literatura com as metodologias propostas neste trabalho. Os *benchmarks Mackey-Glass*, *Filtro Sobel* e *Restauração Salt-and-Pepper* representam aplicações reais e foram utilizados para comparar o desempenho das três versões de GMGP (GMGP-h, GMGP-gpu e GMGP-gpu+). Os *benchmarks Detecção de Intrusos na Rede* e *Multiplexador de 20 bits* representam problemas reais extremamente grandes e apresentam tempo

de execução muito elevado para 10 experimentos. Estes benchmarks foram utilizados para demonstrar a eficiência de GMGP em problemas de larga escala. Neste caso, utilizamos para estes experimentos a versão de GMGP que consideramos mais eficiente para o *benchmark*, a *Detecção de Intrusos na Rede* foi avaliada com GMGP-gpu+ e o *Multiplexador de 20 bits* foi avaliado com GMGP-h.

Os *benchmarks Mackey-Glass*, *Multiplexador de 20 bits*, *Filtro Sobel* e *Detecção de Intrusos na Rede* foram utilizados em trabalhos anteriores de PG acelerada por GPUs [20, 84, 85, 7, 16]. Entretanto, não é possível fazer uma comparação direta com os resultados da literatura, uma vez que eles utilizaram um modelo de PG diferente (com representação dos indivíduos em árvore) e hardware diferente para a avaliação de desempenho.

Foram utilizadas 256 *threads* por bloco nos experimentos. O *grid* é bidimensional e depende do número de indivíduos e do número de amostras de dados. Para um experimento com *(número_de_amostras_de_dados, número_de_indivíduos)* o tamanho do *grid* é *(número_de_amostras_de_dados/256, número_de_indivíduos)*.

8.3 Base de Comparação

Para colocarmos nossos resultados em perspectiva, segundo o que já foi desenvolvido para PG em GPU, implementamos as duas outras metodologias propostas na literatura: **compilação** e **interpretação**. As metodologias para PG em GPU encontradas na literatura, entretanto, não são baseadas em programação genética linear, nem em algoritmos com inspiração quântica. Para realizar comparações diretas e justas dentro do modelo proposto neste trabalho, implementamos as metodologias de compilação e interpretação utilizando programação genética linear baseada em algoritmos com inspiração quântica. Seguimos, contudo, os algoritmos encontrados na literatura.

A abordagem que trabalha com a compilação de código CUDA de GPUs é baseada no trabalho de Harding e Banzhaf [16] e será chamada de **Compiler**. Já a abordagem que realiza a interpretação em tempo de execução é baseada no trabalho de Langdon e Banzhaf [18] e será chamada de **Interpreter**. Também foi realizada uma comparação com uma implementação de PG em código de máquina para CPU, que executa sequencialmente. A implementação para CPU é baseada na QILGP [63] e será chamada de **Serial**.

A metodologia Compiler realiza a montagem do programa para executar a função de avaliação na GPU de forma similar às metodologias GMGP-h e GMGP. Os indivíduos são criados na CPU e enviados para serem executados

na GPU. A principal diferença está na montagem do corpo dos programas. Na metodologia Compiler, o corpo dos programas é criado utilizando instruções da linguagem CUDA. Quando a população está completa, é realizada a compilação utilizando o compilador *nvcc* para gerar o código binário da GPU.

Na metodologia Interpreter, o interpretador é escrito na linguagem PTX, ao invés de RapidMind, como havia sido proposto por Langdon e Banzhaf [18]. O interpretador é automaticamente criado uma única vez, no início da evolução da PG e é reutilizado para avaliar todos os indivíduos. O Algoritmo 1 apresenta uma descrição de alto nível do processo de interpretação. Uma vez que a linguagem pseudo-assembly não possui o comando *switch-case*, foi utilizado uma combinação de instruções `setp.eq.s32` (comparação) e `bra` (saltos condicionados) para obter a mesma funcionalidade. Estas comparações e saltos representam a principal desvantagem da metodologia Interpreter. O interpretador precisa executar mais instruções do que o mínimo necessário para as operações de PG. Para cada operação de PG, é necessário executar pelo menos uma comparação, para identificar a operação de PG, executar a operação de PG propriamente dita e executar um salto para o início do laço, para identificar a próxima operação de PG. Além disso, de uma forma similar, são necessárias comparações e saltos adicionais para identificar os argumentos das operações de PG.

A abordagem Serial é diferente. O programa e o conjunto de dados não precisam ser transferidos para outra memória e o programa é evoluído diretamente no espaço da memória RAM da CPU. Para cada indivíduo, o código de máquina é gerado varrendo-se cada token serialmente e criando o corpo do programa em código de máquina de CPU. Então, o programa é sequencialmente executado para todas as amostras de dados.

As metodologias de PG implementadas empregam um conjunto de funções equivalente e usam o mesmo número de registradores. Na abordagem Serial, o conjunto de funções tem uma instrução atômica *exchange* (`FXCH ST(i)`) que a GPU não tem. Para manter a compatibilidade do conjunto de funções nos experimentos, foi criado uma operação *exchange* na GPU utilizando três operações *move*. A troca de dados entre os registradores Ri e $R0$ é feita utilizando um registrador intermediário $R8$ através das operações $R8 = R0$; $R0 = Ri$; $Ri = R8$, como mostrado na Tabela 6.1.

8.4 Regressão Simbólica

A regressão simbólica é um problema típico na avaliação do desempenho da PG. Também é chamada de ajuste de curvas. Pode ser vista como

Algorithm 1: Pseudo-código do interpretador de PG em GPU.

```
1: TBX← dimensão X do identificador do bloco de threads.
2: TBY← dimensão Y do identificador do bloco de threads.
3: INDIV← número do indivíduo (TBY).
4: N← tamanho do programa (INDIV).
5: THREAD← identificador da thread da GPU.
6: X0← variável 1 de entrada (THREAD + TBX * Número de threads num bloco).
7: X1← variável 2 de entrada (THREAD + TBX * Número de threads num bloco).
8: for k ← 1 to N do
9:    INSTRUCT← número da instrução (k) (INDIV).
10:   ARG← número do argumento (k) (INDIV).
11:   switch (INSTRUCT)
12:   case 0:
13:      Nenhuma operação.
14:   case 1:
15:      switch (ARG)                    % Descrição: R(0) ← R(0) + X(j)
16:      case 0:
17:         add.f32 R0, R0, X0
18:      case 1:
19:         add.f32 R0, R0, X1
20:      → Aqui temos mais casos similares para todas as variáveis de entrada
         e para todas as constantes (X).
21:      end switch
22:   case 2:
23:      switch (ARG)                    % Descrição: R(0) ← R(0) + R(i)
24:      case 0:
25:         add.f32 R0, R0, R0
26:      case 1:
27:         add.f32 R0, R0, R1
28:      → Aqui temos mais casos similares para todos os oito registradores
         auxiliares (Ri).
29:      end switch
30:   case 3:
31:      switch (ARG)                    % Descrição: R(i) ← R(i) + R(0)
32:      case 0:
33:         add.f32 Ri, R0, R0
34:      case 1:
35:         add.f32 Ri, R1, R0
36:      → Aqui temos mais casos similares para todos os oito registradores
         auxiliares (Ri).
37:      end switch
38:   → Aqui temos mais casos similares para as outras instruções tais como
      operações de subtração, multiplicação, divisão, transferência de dados,
      trigonométricas e aritméticas.
39:   default:
40:      exit
41:   end switch
42:   → Escrever o resultado para a memória global.
43: end for
```

a busca por um expressão que melhor se ajusta (com um certo erro) a todos os pontos de amostras de dados. A tarefa da PG é reconstruir uma superfície baseada em alguns pontos dados. Utilizamos dois *benchmarks* bastante conhecidos: *Chapéu Mexicano* e *Salutowicz*. A avaliação destes

benchmarks permite a comparação detalhada das metodologias de PG em GPU: Compiler, Interpreter, Pseudo-assembly e GMGP. Nesta comparação, utilizamos diferentes tamanhos do conjunto de dados. Não estamos interessados aqui em avaliar as diferentes versões de GMGP e os experimentos foram realizados com GMGP-h.

O *benchmark Chapéu Mexicano* [24] é representado por uma função bidimensional dada pela equação (8-1):

$$f(x,y) = \left(1 - \frac{x^2}{4} - \frac{y^2}{4}\right) \times e^{\left(-x^2-y^2\right)/8}. \tag{8-1}$$

O *benchmark Salutowicz* [68] é representado pela equação (8-2). Usamos a versão bidimensional para este *benchmark* .

$$f(x,y) = (y-5) \times e^{-x} \times x^3 \times cos\,(x) \times sin\,(x) \times \left[cos\,(x) \times sin\,(x)^2 - 1\right]. \tag{8-2}$$

Para o *Chapéu Mexicano*, as variáveis x e y são amostradas uniformemente na faixa de $[-4, 4]$. Para *Salutowicz*, elas são amostradas uniformemente na faixa de $[0, 10]$. Esta amostragem gera os conjuntos de dados de treinamento, validação e teste. Os conjuntos de dados de validação e teste representam, cada um deles, cerca de 15% dos dados e são gerados com um intervalo de amostragem diferente do usado para o conjunto de treinamento. Nos nossos experimentos, variamos o número de amostras de dados para investigar o comportamento sobre diferentes tamanhos de conjuntos de dados de entrada. O número de subdivisões de cada variável para o conjunto de dados de treinamento pode ser de 16, 32, 64, 128, 256 e 512, o que é chamado de número de subdivisões, N. Para cada um dos possíveis valores de N, ambas as variáveis x e y empregam o mesmo número de subdivisões, formando uma grade. Quando $N = 16$, é formada uma grade de 16×16, a qual possui 256 amostras de dados. Desta forma, o número de amostras de dados varia de acordo com o seguinte conjunto: $S = \{256, 1024, 4096, 16K, 64K, 256K\}$.

Estes dois *benchmarks* representam duas superfícies diferentes e a PG tem a tarefa de reconstruir estas superfícies a partir de um dado conjunto de pontos.

O valor da avaliação de um indivíduo é o seu Erro Absoluto Médio (MAE) sobre todas as amostras de dados do conjunto de treinamento, como dado pela Equação (8-3):

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |t_i - V[0]_i|, \tag{8-3}$$

onde t_i é o valor alvo esperado para a amostra de dados de número i e $V[0]_i$ é o valor de saída do indivíduo avaliado para a mesma amostra de dados.

8.4.1 Parâmetros

A Tabela 8.1 apresenta os parâmetros usados para executar o *Chapéu Mexicano* e *Salutowicz*. Usamos uma população pequena. Isto é uma das características dos algoritmos evolucionários com inspiração quântica. O estado evolucionário dos algoritmos com inspiração quântica é representado por uma distribuição de probabilidades. Por esse motivo, não há necessidade de se incluir muitos indivíduos. A superposição dos estados fornece uma boa capacidade de busca global devido à diversidade obtida pela representação probabilística.

Tabela 8.1: Parâmetros utilizados para *Chapéu Mexicano* e *Salutowicz*. Os valores do número de gerações, tamanho da população, probabilidade inicial de NOP e tamanho do passo foram obtidos de experimentos anteriores.

Parâmetro	Valor	
	Chapéu Mexicano	Salutowicz
Número de Gerações	400.000	400.000
Tamanho da População	36	36
Probabilidade inicial de NOP ($p_{0,0}$)	0,9	0,9
Tamanho do passo (s)	0,0003	0,002
Comprimento máximo do programa	128	128
Conjunto de funções	Tabela 6.1	Tabela 6.1
Constantes	{1;2;3;4;5;6;7;8;9}	{1;2;3;4;5;6;7;8;9}

8.4.2 Experimentos Preliminares com a Metodologia Compiler

A Tabela 8.2 apresenta os tempos intermediários de todas as metodologias de GPU para o *benchmark Chapéu Mexicano* com um conjunto de amostras de dados de tamanho 16K. Os tempos intermediários de execução são apresentados e organizados nas seguintes categorias:

- `nvcc`: representa o tempo gasto pelo compilador *nvcc* para gerar o código PTX a partir do código fonte CUDA;
- `upload`: representa o tempo gasto compilando código PTX para código binário de GPU (na metodologia GMGP, `upload` significa o tempo gasto carregando os binários da GPU para a placa gráfica antes da execução); na metodologia Interpreter, `upload` é o tempo necessário para transferir os tokens através do barramento PCIe;
- `avaliação`: representa o tempo gasto processando as amostras de dados;

– `interpret`: é o tempo de interpretação para a metodologia Interpreter;

– `download`: é o tempo gasto na cópia dos resultados dos cálculos da GPU para a CPU;

– `CPU`: representa o restante do tempo de execução, incluindo o tempo necessário para executar a metodologia de PG na CPU.

Tabela 8.2: Tempos intermediários de execução de todas as metodologias de PG em GPU (em segundos). A tabela apresenta os tempos para: `Total`, o total da execução; `nvcc`, compilação com o compilador *nvcc*; `upload`, compilação de código PTX (Compiler e Pseudo-Assembly), ou carregar os binários da GPU para memória (GMGP), ou transferir os tokens através do barramento PCIe (Interpreter); `avaliação`, processar as amostras de dados; `interpret`, interpretação; `download`, cópia dos resultados da GPU para a CPU; e `CPU`, execução da metodologia de PG na CPU.

Versão	Total	nvcc	upload	avaliação	interpret	download	CPU
GMGP	292,6	–	73,2	76,9	–	5,13	137,2
Interpreter	636,8	–	3,14	–	542,4	4,35	86,8
Pseudo-Assembly	40.777	–	40.414	118,8	–	6,13	238,8
Compiler	242.186,7	135.027,5	106.458	283,6	–	6,74	410,9

Como pode ser observado na Tabela 8.2, a metodologia Compiler é a única que gasta tempo com o compilador *nvcc*. O tempo gasto com o compilador *nvcc* é muito grande quando comparado a todos os outros tempos. Isto torna a metodologia Compiler três ordens de magnitude mais lenta do que GMGP e Interpreter. Embora trabalhos anteriores tenham apresentado resultados para a metodologia Compiler para acelerar PG em GPUs [15, 14, 16, 17], estes resultados não são diretamente comparáveis com os nossos. Harding e Banzhaf [15] e Chitty [14] não usaram CUDA, mas a liguagem anterior a CUDA, *shader*. Dessa forma, evitaram o overhead do compilador *nvcc*. Harding e Banzhaf [16] usaram CUDA, mas empregaram um cluster para realizar a compilação e reduzir este overhead. Langdon e Harman [17] também usaram CUDA, porém o tempo de compilação dos nossos experimentos é maior do que o tempo de compilação deles por duas razões. Primeiro, o pequeno número de indivíduos na população da abordagem com inspiração quântica requer um número maior de chamadas ao compilador. Segundo, o número total de indivíduos usados nos nossos experimentos (número de gerações $\times$ tamanho da população) é pelo menos uma ordem de grandeza maior do que o utilizado nos experimentos deles.

O tempo de `download` é aproximadamente o mesmo para todas as metodologias porque o mesmo conjunto de dados foi utilizado em todas as abordagens. Dessa forma, os resultados a serem copiados pelo barramento PCIe são os mesmos. O tempo `CPU` para a metodologia Interpreter é um pouco

menor do que para as metodologias GMGP, Compiler e Pseudo-Assembly porque Interpreter não tem que montar os indivíduos na CPU antes de transferir para GPU. Ao invés disso, os tokens são copiados diretamente. O tempo de `avaliação` é da mesma ordem de grandeza para as metodologias Compiler e Pseudo-Assembly e um pouco menor para a metodologia GMGP porque esta utiliza cabeçalho e rodapé otimizados. O tempo `interpret` é aproximadamente uma ordem de magnitude mais lento do que o tempo de `avaliação` para a metodologia GMGP porque Interpreter precisa executar instruções adicionais, tais como comparações e saltos. O tempo de `upload` para GMGP é aproximadamente três ordens de magnitude mais rápido do que o tempo de `upload` para Compiler e Pseudo-Assembly porque GMGP monta diretamente os binários da GPU sem chamar o compilador. O tempo necessário para transferir os tokens através do barramento PCIe na metodologia Interpreter é menor do que o tempo necessário para carregar o código binário de GPU em GMGP.

Uma vez que as outras metodologias resolvem o mesmo problema consideravelmente mais rápido, descartamos a metodologia Compiler para os demais experimentos deste trabalho.

8.4.3 Análise de Desempenho

Comparamos os tempos de execução das metodologias à medida que o número de amostras de dados varia de acordo com o seguinte conjunto: $S = \{256, 1024, 4096, 16K, 64K, 256K\}$. Os tempos totais de execução de *Chapéu Mexicano* e *Salutowicz* para as metodologias Pseudo-Assembly, Interpreter e GMGP são apresentados na Figura 8.1. As curvas foram plotadas em escala logarítmica. Os tempos de execução para a metodologia Pseudo-Assembly permanecem quase constantes à medida que o tamanho do problema aumenta para ambos os *benchmarks*. A metodologia Pseudo-Assembly gasta a maior parte do tempo compilando o código da população de indivíduos e o tempo de compilação não depende do tamanho do problema. Os tempos totais de execução para as metodologias Interpreter e GMGP aumentam quase linearmente à medida que o número de amostras de dados aumenta de 256 para 256K. Para o maior conjunto de dados, 256K, o tempo de execução de Pseudo-Assembly se aproxima ao tempo de execução de Interpreter. Entretanto, a linguagem Pseudo-Assembly tem um desempenho muito inferior ao das outras metodologias quando poucas amostras de dados são consideradas.

Na Tabela 8.3, apresentamos o desempenho das três metodologias para o conjunto de dados com 256K amostras, usando a métrica empregada na

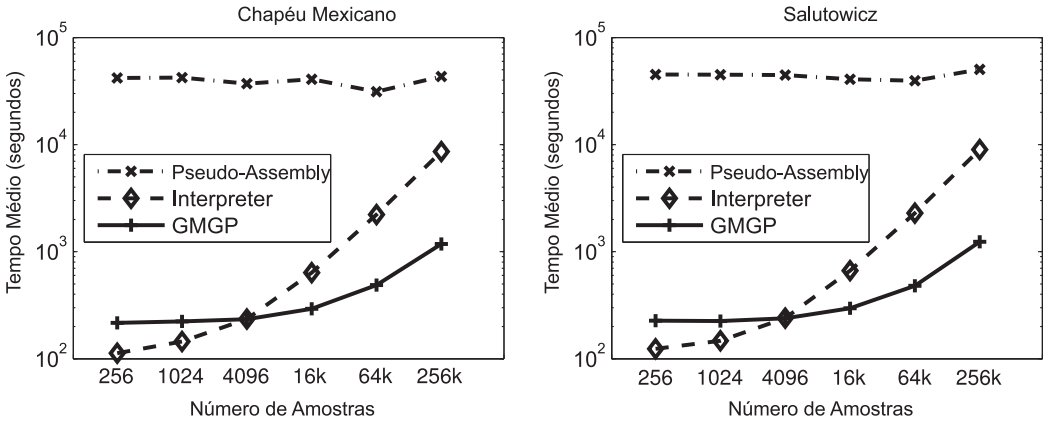

Figura 8.1: Tempo de execução (em segundos) das metodologias Pseudo-Assembly, Interpreter e GMGP para os *benchmarks Chapéu Mexicano* e *Salutowicz*, à medida que o tamanho do conjunto de dados aumenta.

literatura de PG que contabiliza o número de operações de PG por segundo (GPops). Considerando o tempo total de evolução, incluindo o tempo gasto na GPU e na CPU, GMGP obteve 194,4 bilhões de GPops para o *Chapéu Mexicano* e 200,5 bilhões de GPops para *Salutowicz*. Interpreter obteve 26,6 bilhões de GPops para o *Chapéu Mexicano* e 27,5 bilhões de GPops para *Salutowicz*. Pseudo-Assembly obteve os menores valores de GPops, 5,3 bilhões de GPops para o *Chapéu Mexicano* e 4,9 bilhões de GPops para *Salutowicz*. A Tabela 8.3 também apresenta valores de GPops para a avaliação na GPU do melhor indivíduo encontrado ao término do processo evolutivo. Na avaliação do melhor indivíduo, GMGP e Interpreter obtiveram melhores resultados em GPops quando comparados aos da evolução da PG. Esses melhores resultados são explicados pelo fato de que o tempo de CPU e o overhead de transferência de dados não são considerados. Para Pseudo-Assembly, o tempo de compilação também não é considerado e os valores de GPops obtidos para o melhor indivíduo são próximos aos obtidos por GMGP.

A Figura 8.2 apresenta as acelerações obtidas por GMGP e Interpreter usando a metodologia Pseudo-Assembly como referência. Para o *Chapéu Mexicano* e *Salutowicz*, os menores conjuntos de dados geram as maiores acelerações. Para o *Chapéu Mexicano*, Interpreter chega a executar até 371 vezes mais rápido do que Pseudo-Assembly, enquanto que GMGP executa até 193 vezes mais rápido do que Pseudo-Assembly. Os ganhos são similares para *Salutowicz*: Interpreter executa até 363 vezes mais rápido do que Pseudo-Assembly e GMGP executa até 199 vezes mais rápido do que Pseudo-Assembly. Quando o tamanho do problema aumenta, as acelerações em relação a Pseudo-Assembly tornam-se menores para ambos os *benchmarks*.

Tabela 8.3: Desempenho de GMGP, Interpreter e Pseudo-Assembly para *Chapéu Mexicano* e *Salutowicz* em GPops. A tabela apresenta os resultados para o processo evolutivo, incluindo os tempos gastos por GPU e por CPU e os resultados para execução na GPU do melhor indivíduo após o término da evolução.

		Evolução da PG	Melhor Indivíduo
Chapéu Mexicano	GMGP	194,4 bilhões	245,5 bilhões
	Interpreter	26,6 bilhões	29,9 bilhões
	Pseudo-Assembly	5,3 bilhões	161,0 bilhões
Salutowicz	GMGP	200,5 bilhões	240,2 bilhões
	Interpreter	27,5 bilhões	27,0 bilhões
	Pseudo-Assembly	4,9 bilhões	158,4 bilhões

No restante desta análise, compararemos apenas Interpreter e GMGP.

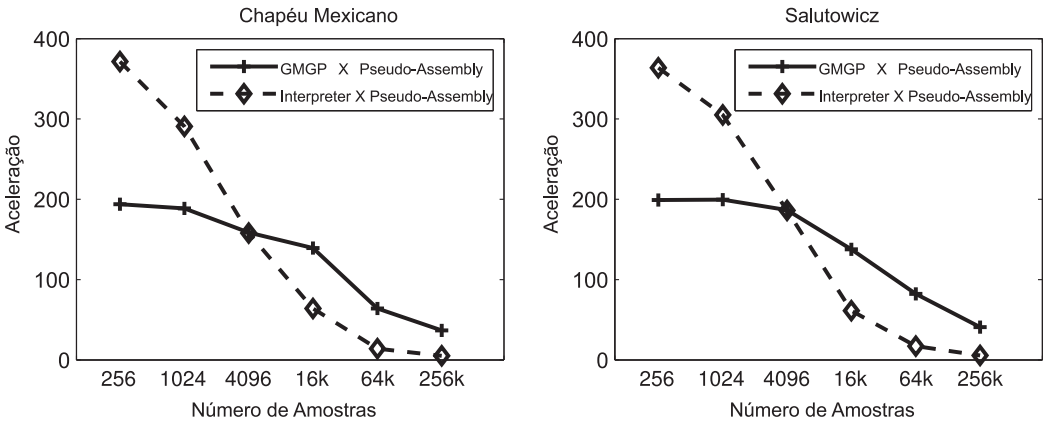

Figura 8.2: Acelerações de GMGP e Interpreter usando Pseudo-Assembly como referência para os *benchmarks Chapéu Mexicano* e *Salutowicz*, à medida que o número de amostras aumenta.

A Figura 8.3 apresenta as acelerações obtidas por GMGP em relação a Interpreter para *Chapéu Mexicano* e *Salutowicz*. A metodologia GMGP tem um desempenho melhor para os conjuntos de dados maiores em ambos os *benchmarks*. Em GMGP, para os menores conjuntos de dados, o número de amostras de dados usadas para operações de ponto flutuante não é suficiente para compensar o overhead de carregar os indivíduos e a metodologia Interpreter é mais rápida. GMGP passa a apresentar um desempenho melhor do que Interpreter quando o número de amostras de dados é maior do que 4.096. GMGP pode ser até 7,3 vezes mais rápida do que Interpreter para o *benchmark Chapéu Mexicano* quando o conjunto de dados contém 256K amostras. Resultados similares foram obtidos para *Salutowicz*. Como seria

esperado, GMGP apresenta-se promissor para aplicações com conjuntos de dados grandes.

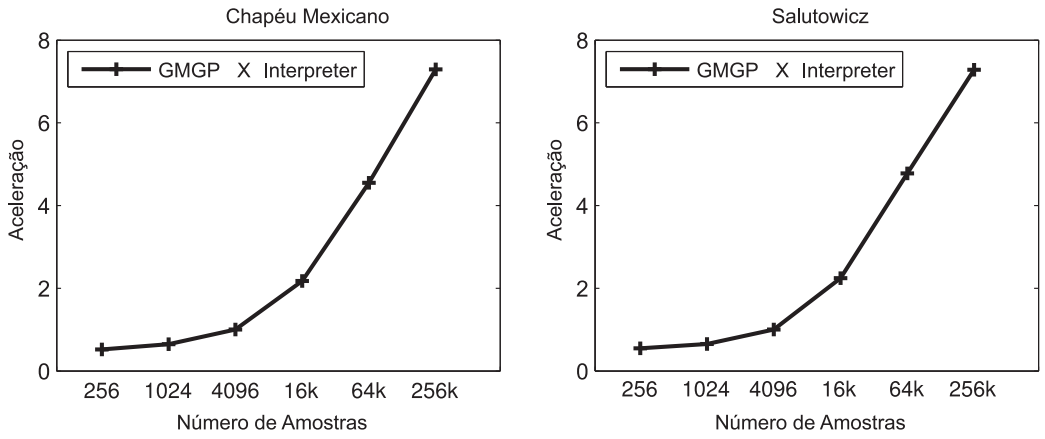

Figura 8.3: Acelerações de GMGP com relação a Interpreter para os *benchmarks Chapéu Mexicano* e *Salutowicz*, à medida que o número de amostras de dados aumenta.

Para explicar o fato de GMGP apresentar um desempenho melhor do que Interpreter para conjuntos de dados grandes, analisamos os tempos intermediários de cada abordagem em detalhes. As Figuras 8.4 e 8.5 apresentam os tempos intermediários de GMGP e Interpreter para os *benchmarks Chapéu Mexicano* e *Salutowicz* à medida que o número de amostras de dados aumenta. Os tempos intermediários apresentados são organizados de acordo com as mesmas categorias utilizadas na Tabela 8.2.

Comparando o tempo de `upload` de GMGP da Figura 8.4 com o tempo de `upload` de Interpreter da Figura 8.5 é possível verificar que o custo para carregar os binários da GPU na placa gráfica é maior do que o custo para transferir os tokens através do barramento PCIe. Entretanto, estes tempos permanecem constantes à medida que o tamanho do problema aumenta. Os tempos de `download` para GMGP e para Interpreter são aproximadamente os mesmos, mas ambos aumentam à medida que o tamanho do problema aumenta. Este resultado é esperado, uma vez que as duas abordagens usam exatamente os mesmos conjuntos de dados e o processamento produz o mesmo número de resultados a serem copiados através do barramento PCIe. O resultado da execução de cada *thread* é um valor com precisão simples de ponto flutuante. Todos os resultados de todas as *threads* de um mesmo bloco são reduzidos para um único valor, o qual é finalmente escrito na memória global. Então, os resultados de todos os blocos são reduzidos, na CPU, para um único valor por indivíduo. O número de resultados transferidos da GPU para a CPU

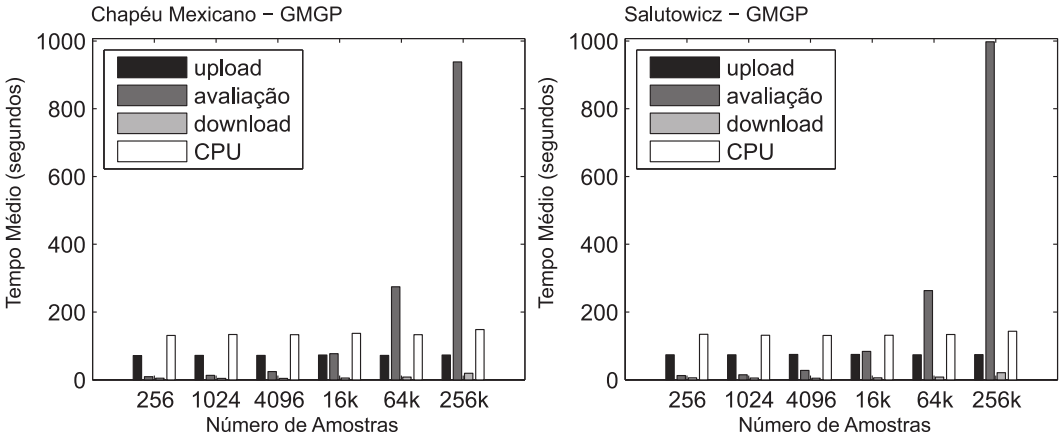

Figura 8.4: Tempos intermediários de GMGP. O gráfico apresenta os tempos intermediários organizados da seguinte forma: upload, tempo gasto carregando os binários da GPU para a memória; avaliação, tempo gasto processando as amostras de dados; download, tempo gasto para copiar os resultados da GPU para a CPU; e CPU, tempo durante o qual a metodologia de PG é executada na CPU.

depende do número de blocos usados para processar todas as amostras de dados. O overhead de `CPU` tem um comportamento similar porque o tempo gasto executando a metodologia de PG na CPU deveria ser aproximadamente o mesmo para GMGP e Interpreter, uma vez que a parte paralelizada do código é a função de avaliação. Podemos comparar os tempos da função de avaliação para GMGP e para Interpreter através da comparação do tempo de `avaliação` da Figura 8.4 com o tempo `interpret` da Figura 8.5. Para conjuntos de dados pequenos, o tempo de `avaliação` de GMGP é menor do que o tempo `interpret` de Interpreter, porém a diferença é pequena. Contudo, à medida que o tamanho do problema aumenta, o tempo `interpret` aumenta significativamente porque a metodologia Interpreter precisa executar um número excessivo de instruções adicionais de comparações e saltos. Para GMGP, o tempo de `avaliação` aumenta pouco porque esta metodologia executa somente as instruções de PG necessárias. Então, a diferença no tempo total entre GMGP e Interpreter aumenta para problemas com conjuntos de dados grandes.

8.4.4 Speedups

Analisamos a melhora no desempenho gerada pelo uso do paralelismo maciço da GPU na PG para as metodologias propostas neste trabalho. GMGP e Pseudo-Assembly são comparadas com a abordagem Serial (implementação

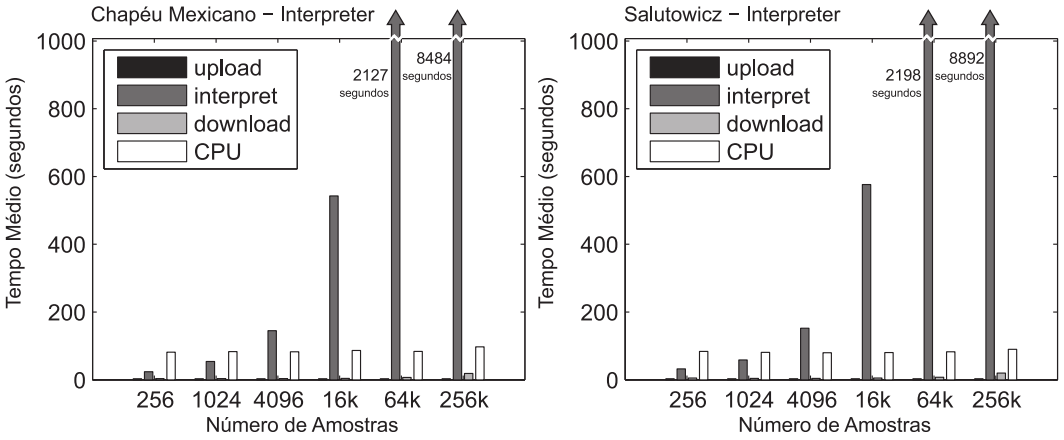

Figura 8.5: Tempos intermediários de Interpreter. O gráfico apresenta os tempos intermediários organizados da seguinte forma: upload, tempo necessário para transferir os tokens através do barramento PCIe; interpret, tempo de interpretação; download, tempo gasto para copiar os resultados da GPU para a CPU; e CPU, tempo durante o qual a metodologia de PG é executada na CPU.

sequencial da QILGP que executa código de máquina da arquitetura Intel x86). Comparamos GMGP e Pseudo-Assembly com Serial na Figura 8.6, apresentando os tempos totais de execução para *Chapéu Mexicano* e *Salutowicz*, variando o tamanho do conjunto de dados. Para a metodologia Pseudo-Assembly, os tempos de execução permanecem quase constantes à medida que o tamanho do problema aumenta em ambos os estudos de caso. Isto ocorre porque Pseudo-Assembly dispende a maior parte do tempo compilando o código PTX dos indivíduos para código binário de GPU e o tempo de compilação não depende do tamanho do problema. Os tempos de execução de ambas as metodologias GMGP e Serial aumentam quase linearmente com o tamanho do problema. Porém, os tempos para GMGP são sempre consideravelmente menores do que os tempos para Serial.

As acelerações de Pseudo-Assembly em relação à abordagem Serial são apresentadas na Figura 8.7. A metodologia Pseudo-Assembly tem um desempenho melhor para os conjuntos de dados maiores em ambos os *benchmarks*. Para os menores conjuntos de dados, quando existem poucas amostras de dados, o ganho com a aceleração do processamento não é suficiente para compensar o tempo de compilação de código PTX para código binário de GPU. Neste caso, a metodologia Pseudo-Assembly tem um desempenho inferior ao da abordagem Serial. Pseudo-Assembly passa a apresentar um desempenho melhor do que Serial quando o número de amostras de dados é maior do que 4.096. Para o maior tamanho do conjunto de dados,

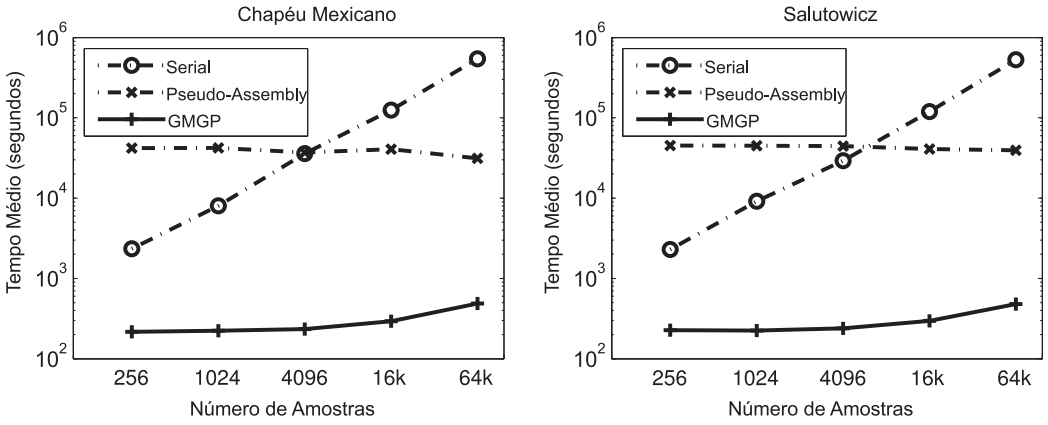

Figura 8.6: Tempo de execução (em segundos) de GMGP, Pseudo-Assembly e Serial para os *benchmarks Chapéu Mexicano* e *Salutowicz*, à medida que o número de amostras aumenta.

Pseudo-Assembly pode ser até 17,3 vezes mais rápida do que Serial para o *benchmark Chapéu Mexicano* e até 13,4 vezes para o *benchmark Salutowicz*. Assim, Pseudo-Assembly apresenta vantagens em relação à execução sequencial na CPU para problemas muito grandes.

As acelerações de GMGP em relação a Serial são apresentadas na Figura 8.8. Para o menor conjunto de dados (256), GMGP é aproximadamente 10,8 vezes mais rápida do que Serial. Para o maior tamanho do problema (64K), GMGP é 1.112 vezes mais rápida do que a abordagem Serial. Ao aumentar o tamanho do problema, com uma quantidade maior de dados a ser processada, o paralelismo da GPU é melhor aproveitado, resultando em maiores acelerações de GMGP quando comparado ao Serial. Estes resultados comprovam que GMGP explora eficientemente o alto grau de paralelismo oferecido pela arquitetura da GPU.

8.4.5 Qualidade dos Resultados

Para comparar a qualidade dos resultados das metodologias Compiler, Pseudo-Assembly, Interpreter e GMGP na GPU, utilizamos a mesma semente inicial do gerador de números aleatórios para iniciar o primeiro experimento de cada uma das abordagens. Comparamos os resultados intermediários e finais. Todas as abordagens de GPU produziram resultados idênticos, comparando todos os dígitos de precisão disponíveis. A única diferença entre eles foi o tempo de execução.

Na Tabela 8.4, analisamos os resultados para 10 execuções diferentes de Compiler, Pseudo-Assembly, Interpreter e GMGP. A Tabela 8.4 apresenta

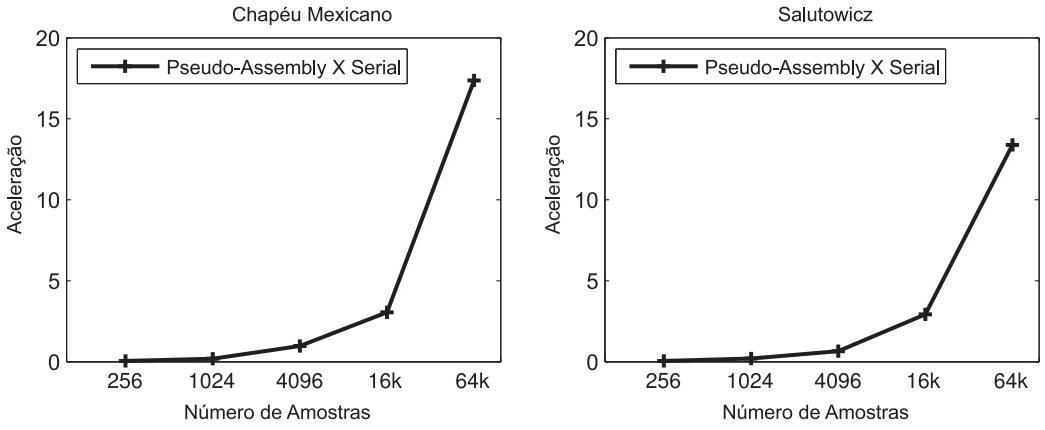

Figura 8.7: Acelerações de Pseudo-Assembly em relação ao Serial para os *benchmarks Chapéu Mexicano* e *Salutowicz*, à medida que o número de amostras aumenta.

a média do melhor indivíduo e o desvio padrão (σ) para os conjuntos de dados de treinamento, validação e teste, para *Chapéu Mexicano* e *Salutowicz*, considerando o conjunto de dados com tamanho 16K. Uma vez que os experimentos foram repetidos 10 vezes, pode se considerar que os desvios padrões apresentados estão baixos para o número de execuções utilizadas.

Tabela 8.4: Erro Absoluto Médio (MAE) na evolução da GPU para *Chapéu Mexicano* e *Salutowicz*. A tabela apresenta a média do melhor indivíduo e o desvio padrão (σ) para os conjuntos de treinamento, validação e teste com tamanho do problema de 16K.

		Treinamento		Validação		Teste	
		Média	σ	Média	σ	Média	σ
Chapéu Mexicano	GMGP	0,046	0,007	0,048	0,008	0,053	0,008
	Interpreter	0,046	0,007	0,048	0,008	0,053	0,008
	Pseudo-Assembly	0,046	0,007	0,048	0,008	0,053	0,008
	Compiler	0,046	0,007	0,048	0,008	0,053	0,008
Salutowicz	GMGP	0,17	0,10	0,19	0,12	0,15	0,08
	Interpreter	0,17	0,10	0,19	0,12	0,15	0,08
	Pseudo-Assembly	0,17	0,10	0,19	0,12	0,15	0,08
	Compiler	0,17	0,10	0,19	0,12	0,15	0,08

8.5 Avaliando Instruções Booleanas: Multiplexador de 20 bits

Um multiplexador é um circuito combinacional que possui diversas entradas e um única saída, e variáveis de seleção que permitem conectar uma das entradas à saída. Desta forma, as informações de duas ou mais fontes são transmitidas através de um único canal. É utilizado em situações onde o custo da implementação de canais separados para cada fonte de

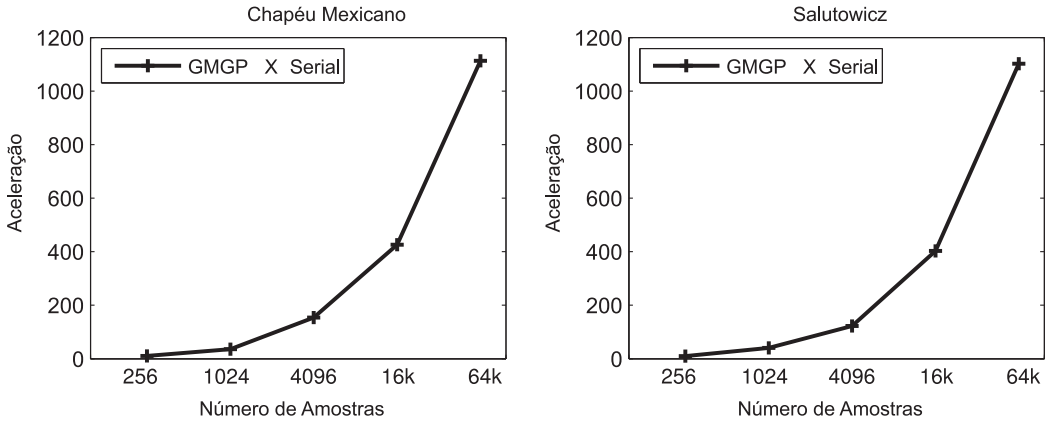

Figura 8.8: Acelerações de GMGP com relação a Serial para os *benchmarks* *Chapéu Mexicano* e *Salutowicz*, à medida que o número de amostras aumenta.

dados é maior do que o custo e/ou a inconveniência de se implementar as funções de multiplexação/demultiplexação. A tarefa da PG é a de encontrar automaticamente as portas lógicas que seriam utilizadas para montar este circuito combinacional de um multiplexador.

Neste trabalho foi utilizado o **Multiplexador de 20 bits** [86] para avaliar a contribuição do paralelismo ao nível de bits na metodologia GMGP através da utilização de instruções booleanas. Ou seja, a PG estava sendo utilizada para evoluir automaticamente um circuito combinacional que possuía 4 variáveis de seleção binárias capazes de controlar a conexão de 16 variáveis entrada em única saída, totalizando 20 variáveis e, por isso, o nome **Multiplexador de 20 bits**. Este benchmark representa um problema em larga escala. Vale ressaltar que este é o primeiro trabalho que consegue evoluir o **Multiplexador de 20 bits** utilizando todas as 2^{20} amostras de dados para todos os indivíduos. Outros trabalhos que evoluem o **Multiplexador de 20 bits** utilizam um número aleatório de amostras, muito menor do que 2^{20}. O maior Multiplexador evoluído que utiliza toda as amostras para todos os indivíduos utiliza 11 bits. Este *benchmark* foi utilizado para demonstrar o potencial de GMGP em tratar problemas muito grandes. Os experimentos foram realizados somente com GMGP-h.

A exploração do paralelismo ao nível de bits na CPU foi proposta por Poli e Langdon [87]. Esta metodologia consiste basicamente em reinterpretar o tipo inteiro como sendo um vetor de bits. Neste caso, cada CPU poderia ser vista como sendo um conjunto SIMD de processadores de 1 bit. As operações bit a bit, tais como AND, OR, NAND, NOR e NOT, são aplicadas simultaneamente a todos os bits do tipo inteiro. Um exemplo foi

apresentado por Poli [88], onde o número inteiro 193 é representado utilizando um tipo inteiro de 32 bits (binário 00000000000000000000000011000001) e o número inteiro 252 (binário 00000000000000000000000011111100) é utilizado para a operação bit a bit AND, resultando no número 192 (binário 00000000000000000000000011000000). Neste caso, são contabilizadas 32 GPops executadas simultaneamente pela CPU.

Langdon [89] utilizou esta metodologia em GPUs para explorar o paralelismo ao nível de bits com um modelo de PG interpretado e representação dos indivíduos em árvore. Assim, conseguiu obter para problemas booleanos um pico de desempenho de 665 bilhões de GPops.

No *Multiplexador de 20 bits*, existem 1.048.576 possíveis combinações dos 20 argumentos. Em nossos experimentos utilizamos todas as 1.048.576 amostras de dados para avaliar cada um dos indivíduos. Isto só foi possível porque a metodologia proposta neste trabalho é capaz de avaliar rapidamente cada indivíduo, através da utilização da representação em código de máquina de GPUs e da utilização do paralelismo ao nível de bits, reinterpretando um tipo inteiro de 32 bits como sendo um vetor de 32 amostras de dados.

8.5.1 Parâmetros

Os parâmetros utilizados para o *Multiplexador de 20 bits* são apresentados na Tabela 8.5. Um número maior de indivíduos foi utilizado na população para este *benchmark*. Isto foi feito porque ele trabalha com um número maior de variáveis de entrada e com um conjunto de dados maior. O número de gerações foi calculado para produzir um número total de indivíduos similar aos números encontrados na literatura. Entretanto, a solução com erro zero foi encontrada antes que o número máximo de gerações tivesse sido atingido para todas os experimentos de PG. O comprimento máximo do programa foi obtido através da verificação do comprimento mínimo necessário para resolver o problema deste *benchmark*.

8.5.2 Análise de Desempenho

Avaliamos o desempenho deste *benchmark* pela métrica GPops. A Tabela 8.6 apresenta o número de GPops obtidos por GMGP para a evolução da PG na GPU (execução da avaliação de todos os indivíduos considerando todas as operações não-`NOP`); a evolução da PG incluindo o tempo necessário para carregar o código dos indivíduos na memória da GPU; a evolução da PG, incluindo o tempo necessário para carregar os indivíduos e para transferir os

Tabela 8.5: Parâmetros para o Multiplexador de 20 bits. Os valores do número de gerações e tamanho da população foram calculados para produzir o mesmo número total de indivíduos encontrados na literatura. A probabilidade inicial de NOP e o comprimento máximo do programa foram obtidos de experimentos anteriores, onde os valores foram variados até que a solução fosse encontrada.

Parâmetro	Valor
Número de Gerações	Primeira Solução ou 14.000.000
Tamanho da População	40
Probabilidade inicial de NOP ($p_{0,0}$)	0,9
Tamanho do passo (s)	0,004
Comprimento máximo do programa	512
Conjunto de funções	Tabela 7.2
Constantes	–

resultados para a memória da CPU; o tempo total de cálculo, incluindo os tempos dispendidos na CPU e na GPU; e a execução do melhor indivíduo na GPU.

A Tabela 8.6 apresenta um desempenho de 5,88 trilhões de GPops para GMGP, considerando somente a avaliação dos indivíduos na GPU. Quando o tempo para carregar os indivíduos é considerado, um valor de 5,24 trilhões de GPops foi obtido. Este *benchmark* tem um conjunto de dados grande. A execução rápida de operações booleanas é suficiente para amortizar o tempo necessário para carregar os indivíduos na memória da GPU. Então, o número de GPops não apresenta um redução muito elevada com a inclusão do tempo necessário para carregar os indivíduos na memória da GPU. Quando o tempo necessário para transferir os resultados para memória da CPU é considerado, o número de GPops permanece quase o mesmo, e GMGP atinge 5,19 trilhões de GPops. Quando o tempo de CPU é considerado, o desempenho é reduzido para 2,74 trilhões de GPops. Este resultado sugere que para este *benchmark* o tempo necessário para processar o modelo na CPU torna-se significativo devido ao número maior de instruções utilizadas. A execução do melhor indivíduo de GMGP na GPU atingiu um desempenho 4,87 trilhões de GPops.

8.5.3 Qualidade dos Resultados

Conforme explicado anteriormente, GMGP foi capaz de encontrar uma solução com erro zero para o *Multiplexador de 20 bits* antes que o número máximo de gerações fosse atingido para todos os 10 experimentos de PG. A Tabela 8.7 apresenta o número de gerações e o número total de indivíduos

Tabela 8.6: Resultados da execução de GMGP para o Multiplexador de 20 bits em GPops. A tabela apresenta o número de GPops dispendidos na evolução da PG na GPU, progressivamente incluindo os *overheads* do tempo necessário para carregar os indivíduos na memória da GPU e do tempo necessário para transferir os resultados para a memória da CPU. Ao final, são apresentados os resultados para o tempo total de cálculo, incluindo o tempo gasto na CPU e na GPU, e os resultados para a execução do melhor indivíduo na GPU.

	GMGP GPops
Evolução da PG na GPU	5,88 trilhões
+ carregar binários na mem. GPU	5,24 trilhões
+ transferir resultados para CPU	5,19 trilhões
Tempo total de cálculo	2,74 trilhões
Melhor Indivíduo	4,87 trilhões

necessário para encontrar a solução.

Tabela 8.7: Geração na qual GMGP resolveu o *Multiplexador de 20 bits* e número total de indivíduos utilizado na evolução. O tamanho da população é 40 indivíduos.

Execução	Geração	Número Total de Indivíduos
1	2.413.505	96.540.200
2	1.246.979	49.879.160
3	3.238.394	129.535.760
4	7.802.509	312.100.360
5	8.892.873	355.714.920
6	10.737.990	429.519.600
7	5.255.728	210.229.120
8	2.576.655	103.066.200
9	5.469.381	218.775.240
10	3.395.730	135.829.200

8.6 O Problema de Classificação: Detecção de Intrusos na Rede

O *benchmark* de *Detecção de Intrusos na Rede* também foi utilizado para avaliar GMGP com um problema real de grandes dimensões. Este *benchmark* apresenta um número de amostras cuja ordem de grandeza é algumas vezes maior em relação aos demais *benchmarks* utilizados nesta tese. A avaliação deste *benchmark* também demonstra o potencial de GMGP em tratar problemas muito grandes. Os experimentos, porém, foram realizados com GMGP-gpu+.

Em nossos experimentos, a classificação de intrusos na rede foi realizada utilizando um conjunto de dados contendo 4.898.431 amostras de dados da *KDD Cup 1999* [90]. Cada amostra de dados era composta por 42 colunas, das quais 38 representavam características numéricas, 3 representam características simbólicas e a última coluna era utilizada para indicar a classe dos dados. Esta última coluna, utilizada para indicar a classe dos dados, informava cinco classes diferentes. A classe *Normal* era utilizada para indicar que não havia ocorrido tentativa de invasão na rede. Além disso, quatro tipos de diferentes de ataque poderiam ocorrer: *Probing*, *denial-of-service* (*DoS*), *user-to-root* (*U2R*) e *remote-to-local* (*R2L*).

Em nossos estudos adotamos uma abordagem diferenciada em relação aos competidores da *KDD Cup 1999* [90] e em relação a Yeung e Chow [91]. Uma vez que o nosso objetivo é avaliar o desempenho de GMGP para problemas de grandes dimensões, ao invés de trabalharmos com uma pequena percentagem das amostras de dados, utilizamos todas as 4.898.431 amostras, as quais foram divididas em 70% para treinamento e 30% para validação. Além disso, pudemos observar que algumas das classes apresentavam um desbalanceamento muito grande entre amostras de dados classificadas como verdadeiro e falso. Para corrigir este problema, foi realizado um balanceamento dos dados, buscando equalizar o número de amostras classificadas como verdadeiro com o número de amostras classificas como falso. Isto foi feito para os conjuntos de dados de treinamento e validação. Para o teste foram utilizados os dados fornecidos em um segundo arquivo, contendo 311.029 amostras de dados, as quais foram utilizadas sem realizar balanceamento, para permitir a comparação com os resultados da literatura. Assim, a Tabela 8.8 apresenta o número total de amostras, após o balanceamento do treinamento e da validação, utilizadas para cada uma das cinco classes, para os conjuntos de dados de treinamento, validação e teste. Conforme podemos observar, o máximo número de amostras de dados utilizadas para treinamento é de 7.001.540 amostras de dados para a classe *U2R*, mostrando que GMGP pode trabalhar um número substancialmente grande de amostras de dados.

A métrica utilizada para avaliar a classe *Normal* foi a métrica TAR (*true acceptance rate*) que representa o percentual de conexões normais dentro do conjunto de dados que são realmente classificadas como normais. Para as classes que representam os diferentes tipos de invasão na rede, *Probing*, *DoS*, *U2R* e *R2L*, foi utilizada a métrica TDR (*true detection rate*) que representa o percentual de conexões de invasão dentro do conjunto de dados que são realmente classificadas como conexões para invasão na rede.

Tabela 8.8: Número total de amostras, após balanceamento dos dados de treinamento e validação, utilizadas para cada uma das classes *Normal*, *Probing*, *DoS*, *U2R* e *R2L* do *benchmark* de *Detecção de Intrusos na rede*, para os conjuntos de treinamento, validação e teste.

	Número de Amostras				
	Normal	Probing	DoS	U2R	R2L
Treinamento	5.472.122	6.787.595	3.429.578	7.001.540	6.873.194
Validação	2.344.652	2.919.770	1.468.853	2.784.839	2.921.085
Teste	311.029	311.029	311.029	311.029	311.029

8.6.1 Parâmetros

Os parâmetros utilizados para a evolução da PG para o *benchmark* de *Detecção de Intrusos na Rede* são apresentados na Tabela 8.9. Conforme pode ser observado, foi utilizado um tamanho pequeno para a população e um grande número de gerações que representam uma característica dos modelos com inspiração quântica. A probabilidade inicial de NOP e o tamanho do passo foram obtidos de experimentos anteriores, verificando os valores adequados para a resolução deste *benchmark*.

Tabela 8.9: Parâmetros para o *benchmark* de *Detecção de Intrusos na Rede*. O tamanho da população e o número de gerações foram definidos de acordo com o utilizado para algoritmos de inspiração quântica. A probabilidade inicial de NOP e o tamanho do passo foram obtidos em experimentos anteriores.

Parâmetro	Valor
Número de Gerações	2.000.000
Tamanho da População	6
Probabilidade inicial de NOP ($p_{0,0}$)	0,9
Tamanho do passo (s)	0,004
Comprimento máximo do programa	128
Conjunto de funções	Tabela 7.1
Constantes	{1;2;3;4;5;6;7;8;9}

8.6.2 Análise de Desempenho

Para avaliar o desempenho deste *benchmark* foi utilizado a métrica das GPops, frequentemente encontrada na literatura de PG. A Tabela 8.10 apresenta o desempenho de GMGP em GPops para cada uma das cinco classes *Normal*, *Probing*, *DoS*, *U2R* e *R2L*. Na primeira linha é apresentado

o desempenho considerando somente o tempo necessário para a avaliação da função de avaliação na GPU, durante todo o processo evolutivo. Uma vez que o *benchmark* de *Detecção de Intrusos na Rede* apresenta um grande número de amostras de dados, a inclusão dos tempos necessários para carregar o código binário na memória da GPU e dos tempos necessários para transferir os resultados para a memória da CPU não degradam significativamente o desempenho em GPops, conforme o apresentado, respectivamente, na segunda e na terceira linhas da tabela. A quarta linha apresenta o desempenho em GPops considerando o tempo de total de execução, incluindo os tempos dispendidos por CPU e GPU, durante todo o processo evolutivo. Conforme pode ser observado, estes valores são muito próximos dos valores apresentados na primeira linha, indicando que para problemas com muitas amostras de dados, GMGP não apresenta seu desempenho afetado pelo tempo de carregamento do código binário na memória da GPU, nem pelo tempo de transferência dos resultados e nem pelo tempo de processamento do modelo de PG. Na última linha da tabela, é apresentado o desempenho em GPops obtido pelo melhor indivíduo, de cada uma das classes *Normal*, *Probing*, *DoS*, *U2R* e *R2L*, executado na GPU ao final da evolução. O desempenho máximo, considerando o tempo total de cálculo, obtido neste *benchmark* foi de 134,30 bilhões de GPops para a classe *Probing*.

Tabela 8.10: Resultados da execução de GMGP para o *benchmark* de *Detecção de Intrusos na Rede* em GPops, para cada uma das classes *Normal*, *Probing*, *DoS*, *U2R* e *R2L*. A tabela apresenta o número de GPops dispendidos na evolução da PG na GPU, progressivamente incluindo os *overheads* do tempo necessário para carregar os indivíduos na memória da GPU e do tempo necessário para transferir os resultados para a memória da CPU. Ao final, são apresentados os resultados para o tempo total de cálculo, incluindo o tempo gasto na CPU e na GPU, e os resultados para a execução do melhor indivíduo na GPU.

	GMGP (bilhões de GPops)				
	Normal	Probing	DoS	U2R	R2L
Evolução da PG na GPU	100,38	136,47	128,28	132,77	129,17
+ carregar binários na mem. GPU	99,65	135,68	126,81	132,02	128,41
+ transferir resultados para CPU	99,50	135,53	126,52	131,87	128,26
Tempo total de cálculo	98,37	134,30	124,32	130,67	127,05
Melhor Indivíduo	101,53	128,46	101,39	109,88	111,36

Harding e Banzhaf [16] trabalharam com este *benchmark* de *Detecção de Intrusos na Rede*, utilizando um modelo de PG com representação dos indivíduos em árvore e uma metodologia de compilação de código, através da utilização da compilação em paralelo num cluster de 14 computadores.

Estes autores também utilizaram todas as 4.898.431 amostras de dados, com o objetivo de manter a ocupação máxima do recursos computacionais utilizados. Após testar diversas configurações, o melhor desempenho médio obtido foi de 2,25 bilhões de GPops e o melhor desempenho de pico foi de 3,44 bilhões de GPops. Contudo, não podemos fazer uma comparação direta com os resultados obtidos no nosso trabalho porque foram utilizados modelos diferentes de GPUs e porque estamos utilizando um modelo de PG diferente, com representação linear dos indivíduos e com inspiração quântica.

8.6.3 Qualidade dos Resultados

Analisamos a qualidade dos resultados produzidos por GMGP através da utilização das mesmas métricas utilizadas no trabalho de Yeung e Chow [91]. Ou seja, através da utilização de TAR (*true acceptance rate*) para a classe *Normal* e de TDR (*true detection rate*) para as classes *Probing*, *DoS*, *U2R* e *R2L*. A primeira linha da Tabela 8.11 é utilizada para apresentar dos resultados da metodologia GMGP, exibindo os valores de TAR ou TDR obtidos para cada uma das classes *Normal*, *Probing*, *DoS*, *U2R* e *R2L*. A segunda linha desta tabela é utilizada para apresentar os resultados obtidos no trabalho de Yeung e Chow [91]. E a última linha é utilizada para apresentar os resultados obtidos pelos competidores vencedores da *KDD Cup 1999* [90]. Conforme pode ser observado, GMGP obteve melhores resultados para quatro das cinco classes: *Normal*, *Probing*, *DoS* e *R2L*. Para a classe *U2R*, GMGP obteve um melhor resultado quando comparado aos competidores da *KDD Cup 1999* [90]. Porém, seu desempenho foi inferior, nesta classe, ao obtido por Yeung e Chow [91].

Tabela 8.11: Resultados da metodologia GMGP e resultados da literatura para o *benchmark* de *Detecção de Intrusos na Rede*, apresentando TAR para a classe *Normal* e TDR para as classes *Probing*, *DoS*, *U2R* e *R2L*.

Método	TAR	TDR			
	Normal	Probing	DoS	U2R	R2L
GMGP	99,57%	99,39%	99,71%	34,18%	71,01%
Yeung e Chow [91]	97,38%	99,17%	96,71%	93,57%	31,17%
KDD [90]	99,45%	87,73%	97,69%	26,32%	10,27%

8.7 Comparando as Três Versões de GMGP

Apresentamos os resultados da comparação de GMGP-h, GMGP-gpu e GMGP-gpu+ para três *benchmarks* que representam problemas reais de

previsão de séries temporais e processamento de imagens: *Mackey-Glass*, *Filtro Sobel* e *Restauração Salt-and-Pepper*. Avaliamos nestes experimentos as duas formas de divisão de trabalho entre a CPU e a GPU e o problema da seleção do melhor indivíduo.

8.7.1 Mackey-Glass

Mackey-Glass [92] é um *benchmark* de previsão de séries temporais caóticas. O sistema caótico *Mackey-Glass* é dado por uma equação diferencial não linear com deslocamento no tempo representada por:

$$\frac{dx(t)}{dt} = \frac{0.2x(t-\tau)}{1+x^{10}(t-\tau)} - 0.1x(t). \tag{8-4}$$

O sistema *Mackey-Glass* tem sido usado como um *benchmark* de PG em vários trabalhos [93, 84]. Em nossos experimentos, as séries temporais consistem de 1.200 pontos de dados e a PG tem a tarefa de prever o próximo valor quando uma série temporal é fornecida. As entradas da PG são oito valores anteriores da série em 1, 2, 4, 8, 16, 32, 64 e 128 passos atrás.

Parâmetros

Os parâmetros utilizados na evolução da PG para o *benchmark* do *Mackey-Glass* são apresentados na Tabela 8.12. Utilizamos também um tamanho pequeno para a população e um grande número de gerações, conforme explicado anteriormente. O número de gerações foi definido de forma a manter a coerência com o número total de indivíduos proposto por Langdon e Banzhaf [84].

Tabela 8.12: Parâmetros para o *benchmark Mackey-Glass*. O número total de indivíduos avaliados (tamanho da população × número de gerações) foi definido de acordo com a literatura. A probabilidade inicial de NOP e o tamanho do passo foram obtidos em experimentos anteriores.

Parâmetro	Valor
Número de Gerações	512.000
Tamanho da População	20
Probabilidade inicial de NOP ($p_{0,0}$)	0,9
Tamanho do passo (s)	0,004
Comprimento máximo do programa	128
Conjunto de funções	Tabela 6.1
Constantes	$\{0;\ 0,01;\ 0,02;\ ...;\ 1,27\}$

Análise de Desempenho

A Tabela 8.13 apresenta os tempos intermediários de execução e o tempo total para GMGP-h, GMGP-gpu e GMGP-gpu+. A execução da função de avaliação de todos os indivíduos na GPU, durante todo o processo evolutivo, consumiu 11,57 segundos para GMGP-h e 16,04 segundos para GMGP-gpu e 10,53 segundos para GMGP-gpu+. Quando comparamos GMGP-gpu+ com as outras duas versões observamos que a diferença no mecanismo de seleção gerou indivíduos com comprimentos diferentes. GMGP-h gerou indivíduos com um comprimento médio do programa de 82 instruções não-`NOP`, enquanto que GMGP-gpu+ evoluiu indivíduos com um comprimento médio do programa de 55 instruções não-`NOP`. Por isso, a diferença no tempo de avaliação dos indivíduos. Porém, a diferença entre GMGP-h e GMGP-gpu+ no comprimento médio dos indivíduos foi de até 49% e o tempo necessário para avaliar as instruções na GPU aumentou somente 9,8%. Isso significa que quanto maior for o número de instruções do indivíduo, maior será a eficiência da GPU ao avaliar estas instruções.

Além da diferença dos comprimentos dos indivíduos, os caminhos evolutivos seguidos por GMGP-h e GMGP-gpu+ são diferentes. A utilização de indivíduos menores por um número maior de gerações pode resultar em um volume de código menor a ser carregado na memória da GPU, resultando em um tempo menor para GMGP-gpu+. GMGP-h consumiu 90,05 segundos para carregar o código binário dos indivíduos na memória da GPU e GMGP-gpu+ apenas 29,58 segundos. Para a transferência dos resultados das avaliações dos indivíduos para a memória da CPU, GMGP-h dispendeu 5,42 segundos e GMGP-gpu+ dispendeu 10,11 segundos. Em GMGP-gpu+ é realizado um pré-processamento na GPU e um volume de resultados menor é copiado para memória da CPU, os quais serão utilizados pela CPU para controlar a chamada dos próximos *kernels*. Este tempo de transferência é maior para GMGP-gpu+ porque foram incluídas as transferências dos tokens da GPU para a CPU, antes da montagem dos indivíduos em código de máquina. O modelo GMGP-h não precisa realizar esta transferência porque os tokens são gerados na CPU.

Com relação ao tempo de execução do modelo de PG, GMGP-h apresentou um tempo de 143,21 segundos, GMGP-gpu apresentou um tempo de 88,69 segundos e GMGP-gpu+ apresentou um tempo de execução de 71,15 segundos. Além do processamento do modelo de PG, nestes tempos estão incluídos um overhead fixo da montagem dos indivíduos, a qual é realizada na CPU para todas as três implementações de GMGP. A montagem dos indivíduos (transformar os tokens em código de máquina) tem que ser realizada na CPU para todas as implementações porque a única opção disponibilizada pela nVidia

para carregar código binário na memória de programas da GPU é através da função `cuModuleLoadDataEx`, a qual utiliza a memória RAM da CPU para fazer a leitura do código binário. Assim, os tokens que representam um volume menor de dados quando comparado ao código de máquina do indivíduo, são transferidos da GPU para a CPU. Cada token é representado por um byte e são necessários dois tokens para representar uma instrução, um token de função de um token de terminal. Ou seja, uma instrução é representada por dois tokens, dois bytes, e em código de máquina, uma instrução precisa de um mínimo de oito bytes para ser representada. Quanto menor o volume de dados, menor o tempo de transferência através do barramento PCIe. Por isso, uma vez que a função `cuModuleLoadDataEx` realiza a leitura do código de máquina a partir da memória RAM da CPU, é vantagem transferir os tokes da GPU para a CPU, realizar a montagem do código de máquina na CPU e em seguida chamar a função `cuModuleLoadDataEx`, para execução em paralelo na GPU. A diferença entre a versão híbrida e a versão GPU poderia ser maior se o comprimento máximo do programa fosse maior do que 128 instruções. Um aumento no número de operações independentes no processamento do modelo de PG poderia melhorar o aproveitamento do paralelismo.

O tempo total de execução inclui o tempo de execução na GPU e na CPU. GMGP-h executou em 250,26 segundos, GMGP-gpu executou em 161,15 segundos e GMGP-gpu+ executou em 121,39 segundos. GMGP-gpu obteve melhores resultados do que GMGP-h pois foram atribuídas à GPU as funções de inicialização, observação, seleção pela avaliação e atualização da evolução. Entretanto, os ganhos não foram tão expressivos. Isto ocorre principalmente devido à função de seleção dos indivíduos. Nesta seleção, é realizada uma ordenação entre as avaliações dos indivíduos da população clássica atual e as avaliações da nova população de indivíduos clássicos que foram observados. Assim, para o *Mackey-Glass*, no qual foi utilizado um tamanho da população de 20 indivíduos, isso significa que existem 40 valores para serem ordenados, 20 das avaliações da população atual e 20 da nova população de indivíduos observados. Quando esta ordenação é realizada pela GPU, alguns problemas são encontrados. Em primeiro lugar, esta pequena quantidade de 40 valores a serem ordenados não é suficiente para ocupar eficientemente uma GPU TITAN que possui 2.688 CUDA *cores*. Em segundo lugar, os indivíduos a serem ordenados estão na memória global da GPU. Como o paralelismo não é bem explorado, a latência de acesso aos dados não é escondida. Em terceiro lugar, a GPU possui dificuldades com códigos com desvios e desbalanceamento de carga, o que é o caso do código de ordenação. Nesta situação, a CPU executa a seleção dos indivíduos de forma mais eficiente.

GMGP-gpu+, por sua vez, obteve melhor desempenho que GMGP-gpu e GMGP-h. Quando comparamos GMGP-gpu+ com GMGP-gpu vemos que a nova proposta de seleção dos indivíduos de GMGP-gpu+ permite reduzir o tempo total de execução pela redução do número de operações realizadas com baixa eficiência na GPU durante a seleção dos indivíduos. Se a seleção dos indivíduos é realizada com poucos valores (40 valores), então quanto menor for a complexidade da função de seleção, menor será o tempo dispendido. Na comparação par a par, cada uma das avaliações dos 20 indivíduos da população clássica atual é comparada com uma das avaliações dos 20 indivíduos observados. Se a avaliação do indivíduo observado for menor, é feita a substituição do indivíduo clássico atual pelo indivíduo observado, substituindo os tokens de função e de terminal. Caso contrário, nada é feito. Além disso, uma vez que na comparação par a par cada indivíduo da população observada pode substituir um indivíduo da população atual, a pressão seletiva por indivíduos com tamanhos menores aumenta. Sempre que houver empate no valor da avaliação e o comprimento do programa do indivíduo observado for menor do que o indivíduo da população clássica, haverá a substituição do indivíduo. Indivíduos com comprimentos menores do tamanho do programa contribuirão para um tempo total de execução menor.

Com relação ao tempo total para a execução na GPU do melhor indivíduo encontrado ao final da evolução, GMGP-h consumiu $1,32 \times 10^{-5}$ segundo, GMGP-gpu dispendeu $1,224 \times 10^{-5}$ segundo e GMGP-gpu+ dispendeu $1,23 \times 10^{-5}$ segundo. O melhor indivíduo de GMGP-h continha 95 instruções não-`NOP`, de GMGP-gpu continha 68 instruções e de GMGP-gpu+ continha 47 instruções. Para o melhor indivíduo, estamos considerando somente o tempo gasto pela GPU na execução da função de avaliação e, portanto, o código usado para executar a função de avaliação na GPU é exatamente o mesmo para as três versões de GMGP. Podemos observar, porém, que ao aumentar o número de instruções de 47 para 95, mais de duas vezes, o tempo de execução permaneceu praticamente o mesmo. A função de avaliação em código de máquina da GPU é mais eficiente para um número maior de instruções. O número de entradas a serem lidas da memória global é exatamente o mesmo para os dois casos e um número maior de instruções a serem executadas sobre os mesmos dados contribui para esconder, de forma eficiente, a latência de acesso à memória global. Além do custo fixo de inicialização do *kernel* ser melhor distribuído entre um número maior de instruções a serem executadas.

A Tabela 8.14 apresenta os valores de GPops obtidos por GMGP-h, GMGP-gpu e GMGP-gpu+. Apresentamos os valores de GPops para a evolução da PG na GPU considerando as operações gastas para executar

Tabela 8.13: Tempos de execução de GMGP-h, GMGP-gpu e GMGP-gpu+ para *Mackey-Glass* em segundos. A tabela apresenta o tempo de execução da função de avaliação na GPU, o tempo necessário para carregar o código binário dos indivíduos na memória da GPU, o tempo necessário para transferir os resultados para a CPU através do barramento PCIe, o tempo de execução do modelo de PG e o tempo total de execução do modelo de PG durante todo o processo evolutivo, incluindo os tempos de CPU e GPU. É apresentado também o tempo para a avaliação na GPU do melhor indivíduo.

	GMGP-h tempo (s)	GMGP-gpu tempo (s)	GMGP-gpu+ tempo (s)
Função de avaliação na GPU	11,57	16,04	10,53
Carregar binários na mem. GPU	90,05	35,65	29,58
Transferir resultados para CPU	5,42	20,76	10,11
Tempo de execução do modelo PG	143,21	88,69	71,15
Tempo total de cálculo	250,26	161,15	121,39
Aptidão do Melhor Indivíduo na GPU	$1{,}32\text{x}10^{-5}$	$1{,}224\text{x}10^{-5}$	$1{,}230\text{x}10^{-5}$

a função de avaliação para todos os indivíduos e contabilizando todas as operações não-`NOP`. GMGP-h obteve 77,7 bilhões de GPops, GMGP-gpu obteve 86,9 bilhões de GPops e GMGP-gpu+ obteve 49,1 bilhões de GPops. GMGP-h evoluiu indivíduos com um comprimento médio do programa maior. A evolução de programas com um número menor de instruções em GMGP-gpu+ leva a um menor número de GPops, apesar do tempo total ser menor para GMGP-gpu+. A função de avaliação consegue esconder melhor a latência de acesso à memória global quando está disponível um número maior de instruções.

Quando consideramos a avaliação da PG combinada com o tempo necessário para carregar o código binário dos indivíduos na memória da GPU, GMGP-h obteve 8,85 bilhões de GPops, GMGP-gpu obteve 26,97 bilhões de GPops e GMGP-gpu+ obteve 12,9 bilhões de GPops. Neste caso, o tempo menor dispendido por GMGP-gpu+ para carregar o código binário de indivíduos menores resultou em um maior desempenho de GPops para GMGP-gpu+, quando comparado a GMGP-h. Além disso, podemos observar também que nas três versões ocorreu uma grande redução no número de GPops quando incluímos o tempo necessário para carregar os binários dos indivíduos na memória da GPU. O tempo para carregar os indivíduos é fixo, não importando o tamanho do conjunto de dados. A ideia é amortizar o custo fixo inicial através de uma execução rápida de operações de ponto flutuante para um conjunto de dados grande. Entretanto, o *benchmark Mackey-Glass* não possui um grande número de amostras de dados.

Quando levamos em consideração o tempo de transferência dos resultados

para a memória da CPU, os valores de GPops são reduzidos para 8,4 bilhões em GMGP-h, 19,24 bilhões em GMGP-gpu e 10,3 bilhões em GMGP-gpu+. Ao considerar o tempo total de cálculo, incluindo os tempos gastos na CPU e na GPU, GMGP-h atinge um desempenho de 3,59 bilhões de GPops, GMGP-gpu atinge 8,65 bilhões de GPops e GMGP-gpu+ 4,27 bilhões de GPops. Para a avaliação do melhor indivíduo, GMGP-h obteve 8,6 bilhões de GPops, GMGP-gpu obteve 6,66 bilhões de GPops e GMGP-gpu+ obteve 4,58 bilhões de GPops. Embora o tempo de avaliação do melhor indivíduo seja menor em GMGP-gpu+, na métrica GPops ele fica menor porque os indivíduos são menores, havendo um menor número de operações de PG para executar.

Tabela 8.14: Resultados de GMGP-h, GMGP-gpu e GMGP-gpu+ para *Mackey-Glass* em GPops. A tabela apresenta o número de GPops necessários para a evolução da PG na GPU, progressivamente incluindo o overhead para carregar o código dos indivíduos na memória da GPU e para transferir os resultados para a CPU. Ao final, fornecemos os resultados para o tempo total de cálculo, incluindo os tempos gastos na CPU e na GPU, e resultados para a execução do melhor indivíduo.

	GMGP-h GPops	GMGP-gpu GPops	GMGP-gpu+ GPops
Evolução da PG na GPU	77,7 bilhões	86,9 bilhões	49,18 bilhões
+ carregar binários na mem. GPU	8,85 bilhões	26,97 bilhões	12,9 bilhões
+ transferir resultados para CPU	8,4 bilhões	19,24 bilhões	10,3 bilhões
Tempo total de cálculo	3,59 bilhões	8,65 bilhões	4,27 bilhões
Melhor Indivíduo	8,6 bilhões	6,66 bilhões	4,58 bilhões

Não é possível fazer uma comparação direta entre as implementações propostas neste trabalho com os resultados encontrados na literatura porque utilizam tipos diferentes de GPU e porque o tipo de algoritmo evolutivo empregado também é diferente. Contudo, vale ressaltar que o valor médio de GPops reportado por Langdon e Banzhaf [18] para o problema do Mackey-Glass foi 895 milhões de GPops.

Qualidade dos Resultados

Analisamos a qualidade dos resultados produzidos por GMGP-h, GMGP-gpu e GMGP-gpu+ calculando o erro RMS e o desvio padrão. Conforme apresentado na Tabela 8.15, a média do erro foi de 0,0077 para GMGP-h, 0,00746 para GMGP-gpu e 0,0053 para GMGP-gpu+. O desvio padrão foi de 0,0021 para GMGP-h, 0,0032 para GMGP-gpu e 0,00078 para GMGP-gpu+. O erro para as metodologias propostas é baixo quando comparado aos erros encontrados na literatura devido à diferença nos modelos

de PG utilizados. Os resultados apresentados na literatura são baseados em um modelo de PG com representação dos indivíduos em árvore, com um tamanho da árvore limitado em 15 e uma profundidade limitada em 4. Enquanto que as metodologias propostas neste trabalho podem evoluir indivíduos com representação linear de 128 instruções. Desta forma, foi possível encontrar um indivíduo que se ajusta melhor a este benchmark.

Tabela 8.15: Erro RMS para a evolução na GPU do benchmark Mackey-Glass. A tabela apresenta a média e o desvio padrão (σ) para os melhores indivíduos evoluídos por GMGP-h, GMGP-gpu e GMGP-gpu+.

		RMS	
		Média	σ
Mackey-Glass	GMGP-h	0,0077	0,0021
	GMGP-gpu	0,00746	0,0032
	GMGP-gpu+	0,0053	0,00078

8.7.2 Filtro Sobel

O *Filtro Sobel* é muito utilizado para a detecção de bordas. As bordas caracterizam os limites dos objetos e por isso são cruciais no processamento de imagens. A detecção de bordas pode auxiliar na segmentação de imagens, compressão de dados e reconstrução de imagens. O operador Sobel calcula o gradiente aproximado da imagem para cada pixel através da convolução da imagem com um par de filtros 3×3. Estes filtros estimam os gradientes nas direções horizontal (x) e vertical (y) e a magnitude do gradiente é a soma destes gradientes. Todas as bordas da imagem original são destacadas na imagem resultante e os contrastes que variam suavemente são suprimidos.

A evolução de um filtro de imagens utiliza uma abordagem de engenharia reversa. O problema é encontrar o mapeamento entre a imagem original e a imagem resultante depois de aplicado o filtro [7, 16]. A tarefa da PG é descobrir as operações que transformam a imagem de entrada na imagem filtrada. Nos nossos experimentos, utilizamos seis imagens 512×512 extraídas do repositório de imagens USC-SIPI [94]. As versões em escala de cinza de todas as seis imagens e as imagens resultantes depois de aplicado o *Filtro Sobel* foram computadas com a ferramenta de processamento de imagens GIMP [95]. A Figura 8.9 apresenta as três imagens utilizadas para treinamento. A Figura 8.10 mostra as duas imagens usadas para validação. A Figura 8.11 mostra a imagem utilizada para o teste.

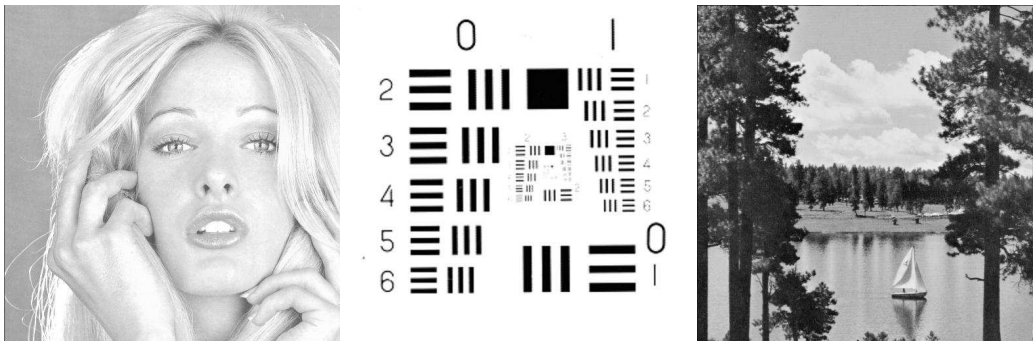

Figura 8.9: As três imagens utilizadas para treinamento do Filtro Sobel em tons de cinza. A resolução de cada imagem é de 512 × 512 pixels.

Figura 8.10: As duas imagens utilizadas para validação do Filtro Sobel em tons de cinza. A resolução de cada imagem é de 512 × 512 pixels.

Parâmetros

Os parâmetros utilizados para a evolução através de PG do *Filtro Sobel* são apresentados na Tabela 8.16. O tamanho da população também emprega um número pequeno de indivíduos. O número de gerações, a probabilidade inicial de NOP, o tamanho do passo e o comprimento máximo do programa foram obtidos em experimentos anteriores.

Análise de Desempenho

A Tabela 8.17 apresenta os tempos intermediários de execução e o tempo total para GMGP-h, GMGP-gpu e GMGP-gpu+. Para este *benchmark*, o tempo de execução da função de avaliação na GPU foi de 1.436 segundos para GMGP-h, 1.752 segundos para GMGP-gpu e 1.244 segundos para GMGP-gpu+. GMGP-h evoluiu indivíduos que atingiram um comprimento médio do programa de 73 instruções não-`NOP`, GMGP-gpu gerou um comprimento médio de 83 instruções e GMGP de 39 instruções. GMGP-h

Figura 8.11: Imagem utilizada para teste do Filtro Sobel em tons de cinza. A resolução da imagem é de 512 × 512 pixels.

Tabela 8.16: Parâmetros utilizados para o *Filtro Sobel*. Os valores do número de gerações, tamanho da população, probabilidade inicial de NOP e tamanho do passo foram obtidos em experimentos anteriores.

Parâmetro	Valor
Número de Gerações	400.000
Tamanho da População	20
Probabilidade inicial de NOP ($p_{0,0}$)	0,9
Tamanho do passo (s)	0,001
Comprimento máximo do programa	128
Conjunto de funções	Tabela 6.1
Constantes	{1;2;3;4;5;6;7;8;9}

evoluiu indivíduos com um tamanho médio do programa 87% maior do que GMGP-gpu+ e o tempo de execução aumentou apenas 15,3%.

O tempo necessário para carregar o código binário dos indivíduos na memória da GPU é de 68,6 segundos para GMGP-h, 28,1 segundos para GMGP-gpu e de 24,2 segundos para GMGP-gpu+. GMGP-gpu+ evoluiu indivíduos que atingiram um menor tamanho médio do programa e o volume de código de máquina a ser carregado na memória da GPU é menor. O tempo necessário para transferir os resultados para a CPU é de 31,7 segundos para GMGP-h, 11,0 segundos para GMGP-gpu e 8 segundos para GMGP-gpu+. Se compararmos o tempo de transferência de GMGP-h para o *Filtro Sobel* com o tempo de transferência no *benchmark* Mackey-Glass (Tabela 8.13), podemos verificar que o tempo de transferência aumentou bastante para o *Filtro Sobel*. Este *benchmark* possui um número maior de amostras de dados. A redução dos resultados não é realizada por completo na GPU. A GPU realiza uma redução parcial, por blocos de threads. Cada bloco de threads escreve um único valor com precisão simples de ponto flutuante na memória global, a

etapa final da redução é realizada na CPU. Quando o número de amostras de dados aumenta, o número de blocos de threads necessários para processar um indivíduo aumenta e o volume de dados de resultados a ser transferido para CPU aumenta também. No caso das versões GMGP-gpu e GMGP-gpu+, isto não acontece. A redução é realizada inteiramente dentro da GPU. Além disso, não é necessário transferir, para a CPU, um valor com precisão simples de ponto flutuante de 32 bits para cada indivíduo. Para cada indivíduo, é transferido apenas um carácter de 8 bits, com a informação mínima necessário para a CPU escolher e controlar os próximos *kernels* a serem executados pela GPU.

GMGP-gpu e GMGP-gpu+ requerem a transferência dos tokens de cada indivíduo para a montagem dos indivíduos em código máquina na CPU. Porém, esta transferência dos tokens representa um custo fixo e já está inclusa nos tempos apresentado na Tabela 8.17. Quando comparamos os tempos de transferência destas versões com os tempos do *benchmark Mackey-Glass*, podemos verificar que o tempo de transferência de resultados é menor para o problema com um número maior de amostras do *Filtro Sobel* . Isto acontece porque o tempo de transferência de resultados para a CPU não depende do número de amostras de dados, dependendo apenas do tamanho da população e do número de gerações. O *Filtro Sobel* foi executado com 400 mil gerações e o *Mackey-Glass* com 512 mil, sendo que ambos usam o mesmo tamanho da população. O *Mackey-Glass* utilizou um número de gerações 28% maior do que o *Filtro Sobel* e teve um tempo de transferência de resultados 26,4% maior para o GMGP-gpu+.

Com relação ao tempo de execução do modelo de PG, GMGP-h dispendeu 115 segundos, GMGP-gpu dispendeu 56,0 segundos e GMGP-gpu+ dispendeu 47,3 segundos. Mesmo incluindo o tempo fixo da montagem dos indivíduos na CPU, GMGP-gpu e GMGP-gpu+ obtiveram reduções no tempo de execução do modelo de PG. O tempo total de execução de GMPG-h foi de 1.651 segundos, de GMGP-gpu foi de 1.847,1 segundos e de GMGP-gpu+ foi de 1.324 segundos. O principal gargalo computacional de um modelo de PG é a função de avaliação. Na metodologia GMGP-h, mesmo executando a função de avaliação na GPU, a avaliação ainda continua representado 86,9% do tempo total de execução para o *Filtro Sobel*. Contudo, a utilização da GPU para a execução das operações do modelo de PG contribui para reduzir o tempo total do processo evolutivo para GMGP-gpu e GMGP-gpu+.

O tempo para a execução na GPU do melhor indivíduo obtido ao final da evolução é de $6,4182 \times 10^{-5}$ segundos para GMGP-h, $6,352 \times 10^{-5}$ segundos para GMGP-gpu e $5,277 \times 10^{-5}$ segundos para GMGP-gpu+. GMGP-h evoluiu

seu melhor indivíduo com 73 instruções, GMGP-gpu evoluiu seu melhor indivíduo com 84 instruções e GMGP-gpu+ obteve seu melhor indivíduo com 42 instruções. Podemos observar que a diferença entre os tempos de execução é muito pequena, apesar de GMGP-gpu+ possuir apenas 42 instruções e GMGP-h possuir 73. Mostrando que a execução da função de avaliação em código de máquina é mais eficiente para um número maior de instruções. Pequenas diferenças entre GMGP-h e GMGP-gpu podem ser atribuídas as diferenças entre as latências de execução de cada instrução empregada em cada um dos indivíduos. Pois, um SM possui 192 SPs capazes de realizar instruções de adição, multiplicação e subtração; e apenas 32 unidades para funções especiais tais como seno, coseno, logaritmo e exponencial. Além disso, a instrução de divisão, mesmo na sua forma mais simples e aproximada, requer um tempo de execução cerca de 10 vezes maior do que instruções de adição, subtração e multiplicação. Assim, indivíduos que possuam um número maior de instruções de divisão, ou funções especiais, terão um tempo de execução um pouco maior do que indivíduos que possuam um número maior de instruções de multiplicação, adição e subtração.

Tabela 8.17: Tempos de execução de GMGP-h, GMGP-gpu e GMGP-gpu+ para o *Filtro Sobel* em segundos. A tabela apresenta os tempos para: execução da função de avaliação na GPU, carregar o código binário dos indivíduos na memória da GPU, transferir os resultados para a CPU através do barramento PCIe, execução do modelo de PG e o total de execução do modelo de PG durante todo o processo evolutivo, incluindo os tempos de CPU e GPU. É apresentado também o tempo para a avaliação na GPU do melhor indivíduo.

	GMGP-h tempo (s)	GMGP-gpu tempo (s)	GMGP-gpu+ tempo (s)
Função de avaliação na GPU	1.436	1.752	1.244
Carregar binários na mem. GPU	68,6	28,1	24,2
Transferir resultados para CPU	31,7	11,0	8,0
Tempo de execução do modelo PG	115,0	56,0	47,3
Tempo total de cálculo	1.651	1.847	1.324
Aptidão do Melhor Indivíduo na GPU	$6{,}4182\text{x}10^{-5}$	$6{,}352\text{x}10^{-5}$	$5{,}277\text{x}10^{-5}$

O desempenho do *Filtro Sobel* em GPops é apresentado na Tabela 8.18. Considerando somente o tempo gasto na avaliação da PG das instruções não-`NOP`, GMGP-h atingiu um desempenho de 287,3 bilhões de GPops, GMGP-gpu atingiu um desempenho de 310,9 bilhões de GPops e GMGP atingiu 240,4 bilhões de GPops. Quando é incluído o tempo necessário para carregar os binários da GPU na memória, as GPops de GMPG-h são reduzidas para 274,2 bilhões, de GMGP-gpu para 306,0 bilhões e de GMGP-gpu+ para

235,8 bilhões. A redução das GPops, com a inclusão do tempo necessário para carregar os binários na memória da GPU, apresentaram uma magnitude menor neste problema do que no *Mackey-Glass* porque o conjunto de dados é maior, compensando o overhead inicial. Quando incluímos o tempo necessário para transferir os resultados dos cálculos para a CPU através do barramento PCIe, GMGP-h obteve 268,6 bilhões de GPops, GMGP-gpu obteve 304,1 bilhões de GPops e GMGP-gpu+ obteve 234,39 bilhões de GPops. Quando é considerado o tempo total do processamento da PG, incluindo o tempo gasto na CPU e na GPU, GMGP-h obteve um desempenho de 249,9 bilhões de GPops, GMGP-gpu obteve 294,9 bilhões de GPops e GMGP-gpu+ obteve 226,03 bilhões de GPops. Ao final da evolução, o melhor indivíduo foi executado na GPU com um desempenho de 295,8 bilhões de GPops para o GMGP-h, 343,9 bilhões de GPops para GMGP-gpu e 207,01 bilhões de GPops para o GMGP-gpu+.

Apesar de GMGP-gpu+ apresentar um tempo total de execução menor do que o tempo total de execução de GMGP-h e GMGP-gpu, em GPops, GMGP-gpu obteve melhores resultados. Isto ocorre porque as implementações levam a caminhos evolutivos diferentes, resultando em tamanhos médios do comprimento do programa diferentes. GMGP-gpu+ evoluiu indivíduos com os menores tamanhos médios. O maior número em GPops para a função de avaliação em código de máquina de GPUs é obtida com programas que apresentam um número maior de instruções.

Tabela 8.18: Resultados de GMGP-h, GMGP-gpu e GMGP-gpu+ executando o *Filtro Sobel* em GPops. A tabela apresenta o número de GPops atingido na evolução da PG na GPU e progressivamente incluindo os overheads de carregar o código binário dos indivíduos na GPU e a transferência dos resultados para a CPU. Ao final, apresentamos os resultados para o tempo total de cálculo, incluindo o tempo gasto na CPU e os resultados para a execução do melhor indivíduo.

	GMGP-h GPops	GMGP-gpu GPops	GMGP-gpu+ GPops
Evolução da PG na GPU	287,3 bilhões	310,9 bilhões	240,4 bilhões
+ carregar binários na mem. GPU	274,2 bilhões	306,0 bilhões	235,8 bilhões
+ transferir resultados para CPU	268,6 bilhões	304,1 bilhões	234,3 bilhões
Tempo total de cálculo	249,9 bilhões	294,9 bilhões	226,0 bilhões
Melhor Indivíduo	295,8 bilhões	343,9 bilhões	207,0 bilhões

Harding e Banzhaf [16] utilizaram um modelo de PG com representação dos indivíduos em árvore e uma metodologia de compilação de código dos indivíduos em um cluster de GPUs com 14 computadores para avaliar a evolução de um filtro de imagens. O filtro de imagens evoluído pelos autores é um filtro de "emboss", porém o mesmo algoritmo de PG que foi utilizado

para evoluir um filtro de "emboss" poderia ser utilizado para evoluir um *Filtro Sobel*. A diferença é o resultado da evolução, ou seja, as instruções do melhor indivíduo obtido ao final da evolução, mantendo-se os mesmos passos a serem seguidos durante o processo evolucionário do algoritmo de PG. Diversas configurações foram testadas, sendo que o melhor desempenho médio obtido durante a evolução foi de 7,06 bilhões de GPops e o melhor desempenho de pico foi de 10,56 bilhões de GPops. Contudo não é possível fazer uma comparação direta com os nossos resultados porque estamos trabalhando com GPUs diferentes e também empregamos um modelo evolutivo diferente, baseado em PG linear com inspiração quântica.

Qualidade dos Resultados

A qualidade dos resultados produzidos por GMGP-h, GMGP-gpu e GMGP-gpu+ para o *Filtro Sobel* foi analisada, calculando-se o MAE e o desvio padrão. A Tabela 8.19 apresenta o MAE e o desvio padrão para os conjuntos de dados de treinamento, validação e teste. Os erros são baixos quando comparados aos valores encontrados na literatura porque os parâmetros da nossa PG foram ajustados para evoluir um filtro com melhor qualidade.

Tabela 8.19: MAEs para a evolução na GPU do *Filtro Sobel*. A tabela apresenta a média e o desvio padrão (σ) para os melhores indivíduos evoluídos por GMGP-h e GMGP aplicados aos conjuntos de dados de treinamento, validação e teste.

		Treinamento		Validação		Teste	
		Média	σ	Média	σ	Média	σ
Sobel filter	GMGP-h	2,11	0,61	2,21	0,64	2,03	0,599
	GMGP-gpu	2,55	0,489	2,68	0,547	2,50	0,529
	GMGP-gpu+	2,18	0,508	2,21	0,502	2,20	0,54

A qualidade dos filtros evoluídos por GMGP também pode ser verificada visualmente. A imagem apresentada no canto superior à esquerda da Figura 8.12 foi obtida utilizando-se o *Filtro Sobel* da ferramenta GIMP. A imagem no canto superior à direita foi obtida aplicando o melhor indivíduo evoluído com GMGP-h à imagem de teste. A imagem no canto inferior à esquerda foi obtida aplicando-se o melhor indivíduo evoluído por GMGP-gpu à imagem de teste. E a imagem no canto inferior à direita foi produzida ao aplicar-se na imagem de teste o melhor indivíduo evoluído por GMGP-gpu+. Estas imagens podem ser comparadas visualmente, verificando que GMGP-h, GMGP-gpu e GMGP-gpu+ foram capazes de alcançar o resultado esperado para este problema, realizando a engenharia reversa automática e chegando

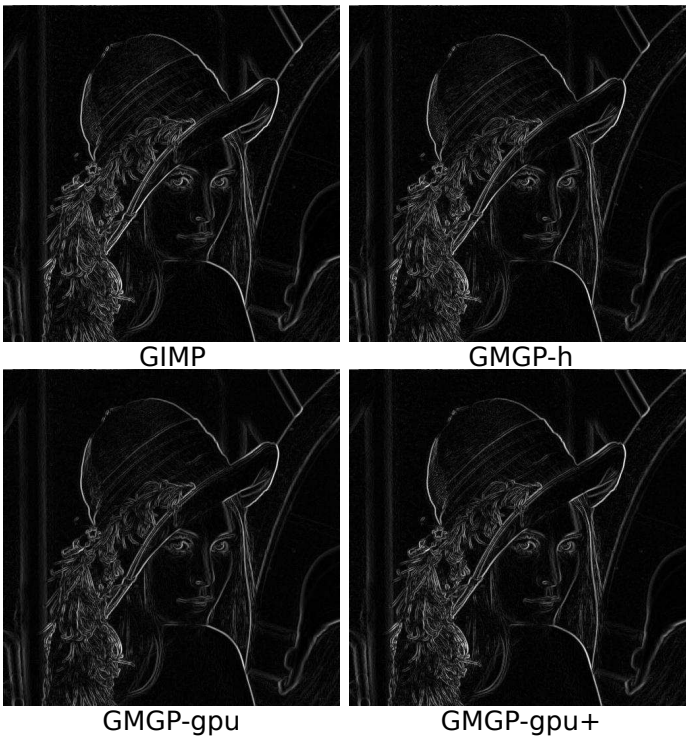

Figura 8.12: Resultados obtidos ao aplicar os melhores indivíduos evoluídos por GMGP-h, GMGP-gpu e GMGP-gpu+ na imagem de teste. A imagem no canto superior à esquerda foi obtida utilizando-se o *Filtro Sobel* da ferramenta GIMP. A imagem no canto superior à direita foi produzida pelo melhor indivíduo evoluído por GMGP-h aplicado à imagem de teste. A imagem no canto inferior à esquerda foi obtida aplicando-se o melhor indivíduo evoluído por GMGP-gpu à imagem de teste. E a imagem no canto inferior à direita foi obtida pelo melhor indivíduo evoluído por GMGP-gpu+ aplicado à imagem de teste.

a um conjunto de instruções capaz de mapear eficientemente a imagem de entrada em tons de cinza para a imagem de saída esperada para o *Filtro Sobel*.

8.7.3 Restauração Salt-and-Pepper

O ruído *Salt-And-Pepper* é caracterizado pela substituição de alguns pixels aleatoriamente selecionados em imagens por pixels com a coloração branca ou preta. Durante a aquisição de imagens, uma conjunto de pixels com a coloração muito escura pode ocorrer devido à utilização de sensor defeituoso. Por outro lado, pixels com a coloração muito clara também podem ocorrer devido ao tempo prolongado de exposição. Nestes casos, o objetivo da utilização do filtro é remover este tipo de ruído, restaurando a imagem.

A abordagem adotada neste trabalho consiste em utilizar PG para evoluir um filtro para *Restauração Salt-and-Pepper*. O filtro de ruído evoluído por PG é composto por duas etapas. Uma primeira etapa é utilizada para classificar os pixels da imagem entre pixels que apresentam ruído e pixels que não

apresentam ruído. A segunda etapa é utilizada para suavizar a imagem, através de um modelo de regressão linear evoluído por PG, melhorando o seu aspecto visual.

O modelo evoluído por PG consiste na primeira etapa de um classificador que separa os pixels que apresentam ruído daqueles que não apresentam. Neste caso, para cada pixel, a saída é a classificação do pixel e as entradas são os valores, em escala de cinza, de 9 pixels, o pixel central e os seus oito vizinhos. Esta saída é utilizada da seguinte forma. Os pixels que foram classificados como não apresentando ruído são mantidos de acordo com a imagem original e os pixels classificados como apresentando ruído são substituídos pela média dos seus vizinhos que foram classificados como não apresentando ruído. Este procedimento gera uma imagem intermediária.

Esta imagem intermediária é utilizada numa segunda etapa, onde a PG evolui um modelo de regressão para suavizar a imagem intermediária. Dessa forma, o seu aspecto visual é melhorado, tornando esta imagem intermediária mais próxima da imagem sem ruído. Neste caso, para cada pixel, a saída é o valor alvo da imagem sem ruído e as entradas são os valores, em escala de cinza, de 9 pixels da imagem intermediária, o pixel central e os seus oito vizinhos. Após esta segunda etapa, o filtro para *Restauração Salt-and-Pepper* evoluído por PG apresenta imagens com melhor qualidade.

Nos nossos experimentos, utilizamos seis imagens 512×512 extraídas do repositório de imagens USC-SIPI [94]. As versões em escala de cinza de todas as seis imagens foram computadas com a ferramenta de processamento de imagens GIMP [95]. Cada uma destas imagens foi modificada pela adição de ruído *Salt-And-Pepper* num percentual de 80%. Utilizamos as mesmas imagens de treinamento, validação e teste que as utilizadas na avaliação do *Filtro Sobel*, conforme mostram as Figuras 8.9, 8.10 e 8.11 da seção anterior.

A métrica utilizada para avaliar a *Restauração Salt-and-Pepper* evoluída por GMGP foi o PSNR (*Peak Signal-to-Noise Ratio*). A definição de PSNR requer a definição do erro MSE (*Mean Squared Error*), o qual é dado pela equação (8-5):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(t_i - V[0]_i)^2, \tag{8-5}$$

onde t_i é o valor alvo esperado para a amostra de dados de número i e $V[0]_i$ é o valor de saída do indivíduo avaliado para a mesma amostra de dados.

Assim, o PSNR pode ser definido para imagens em termos do MSE de acordo com a equação (8-6):

$$\text{PSNR} = 10 \times log_{10}\frac{(255)^2}{MSE}, \tag{8-6}$$

Parâmetros

Os parâmetros utilizados para a evolução através de PG da *Restauração Salt-and-Pepper* são apresentados na Tabela 8.20, para as etapas de classificação e de regressão. O tamanho da população utiliza um número pequeno de indivíduos e o número de gerações é elevado de acordo com o que foi visto anteriormente. O número de gerações, a probabilidade inicial de NOP, o tamanho do passo e o comprimento máximo do programa foram obtidos em experimentos anteriores.

Tabela 8.20: Parâmetros utilizados para a *Restauração Salt-and-Pepper*, para as etapas de classificação e de regressão. Os valores utilizados para o número de gerações, tamanho da população, probabilidade inicial de NOP e tamanho do passo foram obtidos de experimentos anteriores.

Parâmetro	Valor	
	Classificação	Regressão
Número de Gerações	400.000	4.000.000
Tamanho da População	6	6
Probabilidade inicial de NOP ($p_{0,0}$)	0,9	0,9
Tamanho do passo (s)	0,004	0,004
Comprimento máximo do programa	256	128
Conjunto de funções	Tabela 7.1	Tabela 6.1
Constantes	{1;2;3;4;5;6;7;8;9}	{1;2;3;4;5;6;7;8;9}

Análise de Desempenho

A Tabela 8.21 apresenta a execução do classificador usando cada uma das três metodologias GMGP-h, GMGP-gpu e GMGP-gpu+. São apresentados os tempos intermediários, através dos tempos necessários para a avaliação na GPU da função de avaliação para todos os indivíduos durante todo o processo evolutivo. O tempo necessário para carregar os binários dos indivíduos em código de máquina para a memória da GPU. O tempo necessário para transferir os resultados para a CPU e o tempo de execução do modelo de PG. Ao final, é apresentado o tempo total da execução, incluindo os tempos dispendidos por CPU e GPU. É apresentado também o tempo dispendido para a execução na GPU do melhor indivíduo encontrado ao final do processo evolutivo.

Conforme pode ser observado, o tempo de execução do modelo de PG para a metodologia GMGP-gpu é menor do que para a metodologia GMGP-h, devido basicamente à utilização da GPU para o processamento do modelo de PG. GMGP-gpu+ apresenta o menor tempo para a execução do modelo de PG dentre as três metodologias, uma vez que a utilização da comparação par

Tabela 8.21: Tempos de execução para o classificador da *Restauração Salt-and-Pepper* evoluído por GMGP-h, GMGP-gpu e GMGP-gpu+, em segundos. A tabela apresenta os tempos para: execução da função de avaliação na GPU, carregar o código binário dos indivíduos na memória da GPU, transferir os resultados para a CPU através do barramento PCIe, execução do modelo de PG e o total de execução do modelo de PG durante todo o processo evolutivo, incluindo os tempos de CPU e GPU. É apresentado também o tempo para a avaliação na GPU do melhor indivíduo.

	GMGP-h tempo (s)	GMGP-gpu tempo (s)	GMGP-gpu+ tempo (s)
Função de avaliação na GPU	723,57	533,06	414,77
Carregar binários na mem. GPU	63,61	20,36	19,53
Transferir resultados para CPU	13,97	7,54	7,32
Tempo de execução do modelo PG	74,66	49,7	43,33
Tempo total de cálculo	875,83	610,67	484,95
Aptidão do Melhor Indivíduo na GPU	$6{,}63\text{x}10^{-5}$	$10{,}58\text{x}10^{-5}$	$4{,}36\text{x}10^{-5}$

a par é menos dispendiosa do que a execução da ordenação dos indivíduos. Com relação ao tempo total de execução durante todo o processo evolutivo, a metodologia GMGP-gpu+ apresenta o menor tempo total, seguida pela metodologia GMGP-gpu.

A Tabela 8.22 apresenta os resultados obtidos em GPops para a execução do classificador em GMGP-h, GMGP-gpu e GMGP-gpu+. Conforme pode ser observado, considerando o tempo total de cálculo, GMGP-gpu obteve um melhor resultado em GPops quando comparado a GMGP-gpu+, apesar de GMGP-gpu+ apresentar um tempo total de execução menor. Conforme discutido nas seções anteriores, isto se deve aos diferentes caminhos evolutivos seguidos por GMGP-gpu e GMGP-gpu+, ao utilizar um mecanismo diferente de seleção dos indivíduos, levando GMGP-gpu+ a evoluir indivíduos com um menor comprimento médio dos programas, o que se reflete no cálculo das GPops.

Os tempos intermediários e totais de execução para a evolução da etapa de suavização da imagem, gerando um modelo de regressão linear para a *Restauração Salt-and-Pepper*, para GMGP-h, GMGP-gpu e GMGP-gpu+, são apresentados na Tabela 8.23. Conforme pode ser observado, novamente o tempo de execução do modelo de PG para a metodologia GMGP-gpu é menor do que este tempo para a metodologia GMGP-h. Além disso, o tempo de execução de GMGP-gpu+ é o menor dentre as três metodologias. O menor tempo total de cálculo, durante todo o processo evolutivo foi apresentado pelo metodologia GMGP-gpu+.

Tabela 8.22: Resultados de GMGP-h, GMGP-gpu e GMGP-gpu+ para a evolução do classificador da *Restauração Salt-and-Pepper* em GPops. A tabela apresenta o número de GPops necessários para a evolução da PG na GPU, progressivamente incluindo o overhead para carregar o código dos indivíduos na memória da GPU e para transferir os resultados para a CPU. Ao final, fornecemos os resultados para o tempo total de cálculo, incluindo os tempos gastos na CPU e na GPU, e resultados para a execução do melhor indivíduo.

	GMGP-h GPops	GMGP-gpu GPops	GMGP-gpu+ GPops
Evolução da PG na GPU	292,3 bilhões	413,7 bilhões	405,1 bilhões
+ carregar binários na mem. GPU	268,7 bilhões	398,5 bilhões	386,9 bilhões
+ transferir resultados para CPU	264,0 bilhões	393,1 bilhões	380,5 bilhões
Tempo total de cálculo	241,5 bilhões	361,1 bilhões	346,5 bilhões
Melhor Indivíduo	450,6 bilhões	282,4 bilhões	192,3 bilhões

Tabela 8.23: Tempos de execução para a etapa de suavização da imagem através de um modelo de regressão linear para a *Restauração Salt-and-Pepper* evoluído por GMGP-h, GMGP-gpu e GMGP-gpu+, em segundos. A tabela apresenta os tempos para: execução da função de avaliação na GPU, carregar o código binário dos indivíduos na memória da GPU, transferir os resultados para a CPU através do barramento PCIe, execução do modelo de PG e o total de execução do modelo de PG durante todo o processo evolutivo, incluindo os tempos de CPU e GPU. É apresentado também o tempo para a avaliação na GPU do melhor indivíduo.

	GMGP-h tempo (s)	GMGP-gpu tempo (s)	GMGP-gpu+ tempo (s)
Função de avaliação na GPU	3.871,4	4.155,4	3.221,8
Carregar binários na mem. GPU	626,3	195,2	182,9
Transferir resultados para CPU	114,8	74,4	73,3
Tempo de execução do modelo PG	576,4	451,1	392,0
Tempo total de cálculo	5.188,8	4.876,1	3.870,0
Aptidão do Melhor Indivíduo na GPU	$6.207\text{x}10^{-5}$	$6{,}54\text{x}10^{-5}$	$6{,}33\text{x}10^{-5}$

A Tabela 8.24 apresenta os resultados em GPops para a etapa de suavização da imagem através de um modelo de regressão linear para a *Restauração Salt-and-Pepper*, para GMGP-h, GMGP-gpu e GMGP-gpu+. Conforme pode ser observado, a etapa de suavização apresentou resultados similares em GPops aos da etapa de classificação. GMGP-gpu apresentou maiores valores de GPops que GMGP-gpu+, pelos mesmos motivos explicados anteriormente.

Tabela 8.24: Resultados de GMGP-h, GMGP-gpu e GMGP-gpu+ para a evolução da etapa de suavização da imagem através de um modelo de regressão linear para a *Restauração Salt-and-Pepper* em GPops. A tabela apresenta o número de GPops necessários para a evolução da PG na GPU, progressivamente incluindo o overhead para carregar o código dos indivíduos na memória da GPU e para transferir os resultados para a CPU. Ao final, fornecemos os resultados para o tempo total de cálculo, incluindo os tempos gastos na CPU e na GPU, e resultados para a execução do melhor indivíduo.

	GMGP-h GPops	GMGP-gpu GPops	GMGP-gpu+ GPops
Evolução da PG na GPU	360,0 bilhões	383,5 bilhões	280,1 bilhões
+ carregar binários na mem. GPU	309,8 bilhões	366,3 bilhões	265,1 bilhões
+ transferir resultados para CPU	302,1 bilhões	360,1 bilhões	259,5 bilhões
Tempo total de cálculo	268,6 bilhões	326,8 bilhões	233,2 bilhões
Melhor Indivíduo	388,5 bilhões	312,4 bilhões	285,5 bilhões

Qualidade dos Resultados

A qualidade dos resultados para a *Restauração Salt-and-Pepper* evoluída por GMGP foi analisada através do cálculo da métrica PSNR, para GMGP-h, GMGP-gpu e GMGP-gpu+. O valor da métrica PSNR calculada foi de 27,35 para GMGP-h, 27,44 para GMGP-gpu e 27,34 para GMGP-gpu+. Os valores obtidos para estes resultados são comparáveis aos valores encontrados na literatura para o ruído *Salt-And-Pepper* com um percentual de 80%.

Além disso, a qualidade dos resultados pode ser analisada visualmente, através dos resultados apresentados para a imagem de teste restaurada pelas metodologias GMGP-h, GMGP-gpu e GMGP-gpu+, apresentadas na Figura 8.13. A imagem apresentada no canto superior à esquerda corresponde à imagem de teste com 80% de ruído. A imagem apresentada no canto superior à direita foi obtida aplicando o melhor indivíduo evoluído com GMGP-h para restaurar a imagem de teste. A imagem no canto inferior à esquerda foi obtida aplicando-se o melhor indivíduo evoluído por GMGP-gpu para restaurar a imagem de teste. A imagem no canto inferior à direita corresponde à restauração obtida ao aplicar-se na imagem de teste o melhor indivíduo evoluído por GMGP-gpu+. Estas imagens podem ser comparadas visualmente verificando que as metodologias GMGP-h, GMGP-gpu e GMGP-gpu+ foram eficientes para evoluir um filtro para *Restauração Salt-and-Pepper* capaz de recuperar as imagens, restabelecendo uma adequada condição visual.

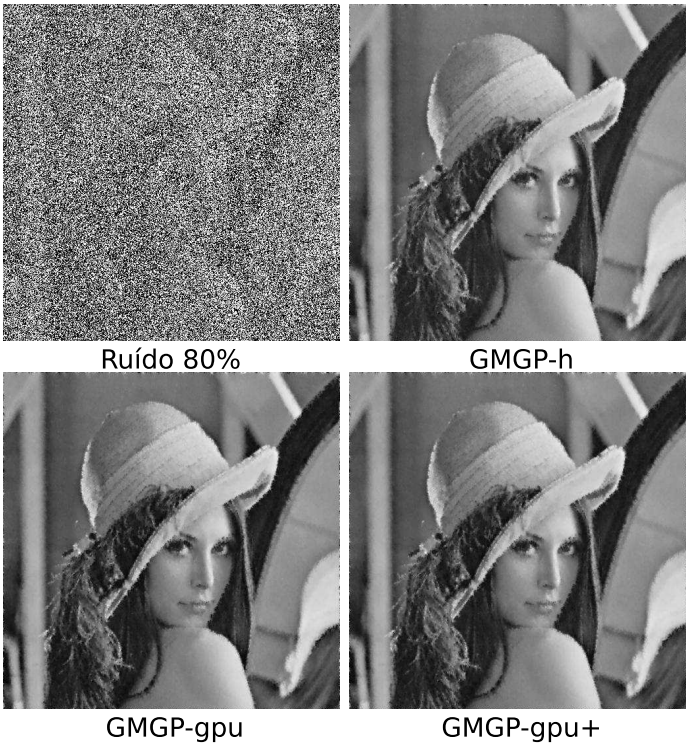

Figura 8.13: Resultados obtidos ao aplicar os melhores indivíduos evoluídos por GMGP-h, GMGP-gpu e GMGP-gpu+ na imagem de teste. A imagem no canto superior à esquerda corresponde a imagem com 80% de ruído *Salt-And-Pepper*. A imagem no canto superior à direita foi produzida ao utilizar o melhor indivíduo evoluído por GMGP-h para restaurar a imagem de teste. A imagem no canto inferior à esquerda foi obtida aplicando-se o melhor indivíduo evoluído por GMGP-gpu para restaurar a imagem de teste. E a imagem no canto inferior à direita foi obtida quando o melhor indivíduo evoluído por GMGP-gpu+ foi utilizado para restaurar a imagem de teste.