6
Acelerando PG em GPU - Manipulando a Linguagem Intermediária

A abordagem de compilação em linguagem intermediária proposta neste trabalho é chamada de Pseudo-assembly e é baseada no algoritmo evolutivo com inspiração quântica QILGP [62]. As principais diferenças entre Pseudo-assembly e QILGP estão na representação dos indivíduos e na maneira como são avaliados. Pseudo-assembly representa os indivíduos usando a linguagem PTX e evolui os programas em paralelo utilizando a GPU. A principal vantagem de Pseudo-assembly é diminuir o alto *overhead* de compilação gerado pelo compilador *nvcc* e evitar o *overhead* em tempo de execução de interpretação do código.

6.1
A linguagem PTX

A linguagem PTX define uma máquina virtual e um conjunto de instruções para a execução de *threads* em paralelo nas GPUs da nVidia, em aplicações de propósito geral [22]. Este conjunto de instruções é compatível com múltiplas gerações de GPUs. Assim, a linguagem PTX fornece a compatibilidade do código para gerações futuras de GPUs da nVidia, e códigos implementados e testados para arquiteturas antigas de GPUs poderão ser executados em arquiteturas de GPUs futuras.

O código fonte em linguagem PTX é composto por módulos de texto em caracteres ASCII. As linhas são separadas pelo carácter "`\n`". Os espaços em branco são ignorados, a menos que estejam separando os termos de uma instrução. As linhas começando com "`#`"são consideradas como sendo diretivas para o pré-processador. Dentre as quais podemos ter: `#include`, `#define`, `#if`, `#ifdef`, `#else`, `#endif`.

A utilização de comentários segue o padrão utilizado na sintaxe da linguagem C/C++, onde múltiplas linhas podem ser comentadas utilizando os carácteres "`/*`" no início e os carácteres "`*/`" no final. Da mesma forma, para comentar uma linha basta utilizar os carácteres "`//`" no início da linha a ser comentada.

Uma instrução é formada por uma palavra reservada, utilizada para informar a operação a ser realizada, a qual é seguida por nenhum, um ou mais operandos separados por vírgula e ao final é utilizado um ponto e vírgula. Os operandos podem ser registradores, expressões constantes, endereços de

memória ou etiquetas para saltos no programa. Além disso, as instruções podem ter um predicado opcional que controla a execução condicional de instruções. Este predicado opcional, quando utilizado, é escrito antes da palavra reservada da instrução, da seguinte forma: "`@p`", sendo que "`p`" é o registrador de predicados. Este predicado também pode ser escrito por "`@!p`", para negar o conteúdo do registrador de predicados "`p`".

Entre as palavras reservadas utilizadas para informar as instruções, as principais são:

- `ld` e `st`: utilizadas, respectivamente, para realizar a leitura e a escrita de valores das memórias global e compartilhada;
- `add`, `sub` e `mul`: utilizadas, respectivamente, para adição, subtração e multiplicação de dois valores inteiros ou de ponto flutuante;
- `div`: utilizada para a divisão de um valor por outro, inteiros ou ponto flutuante;
- `fma`: é uma instrução que realiza duas operações de ponto flutuante, de forma combinada, sobre três valores de entrada. Uma multiplicação é realizada entre dois dos valores e o resultado intermediário é somado ao terceiro valor de entrada;
- `mov`: utilizada para atribuir a um registrador o valor de outro, ou de uma constante ou um valor lido de uma posição de memória;
- `abs`: retorna o valor absoluto ou módulo para inteiros ou ponto flutuante;
- `sqrt`, `sin` e `cos`: utilizadas, respectivamente, para calcular a raiz quadrada, o seno e cosseno para valores de ponto flutuante;
- `lg2` e `ex2`: utilizadas, respectivamente, para calcular o logaritmo e a exponencial, ambos com base 2 para valores de ponto flutuante;
- `setp`: utilizada para comparar dois valores e, como saída, atribuir o valor de um registrador de predicados `p`;
- `bra`: utilizada para realizar um salto no código, o qual pode ser condicionado ou não por um predicado opcional `@p`. É necessário uma etiqueta para informar a posição do código para onde será realizado o salto;
- `max` e `min`: utilizadas para calcular, respectivamente, os valores de máximo e de mínimo entre dois valores inteiros ou de ponto flutuante;
- `and`, `or`, `xor` e `not`: realizam, respectivamente, as operações booleanas AND, OR, XOR e NOT, bit a bit, sobre valores inteiros de 32 ou 64 bits;
- `bar`: representa uma barreira de sincronização entre *threads* de um mesmo bloco;

– `exit`: é a instrução de finalização da execução das *threads*.

Para exemplificar a utilização de instruções da linguagem PTX, é apresentado, na Figura 6.1, um pequeno trecho de código escrito utilizando a linguagem C e, ao lado, é apresentado o código equivalente em linguagem PTX. O código apresenta um laço que realiza a adição de uma variável de ponto flutuante "`f2`" para o conteúdo da variável "`f1`". A execução do laço será interrompida quando o conteúdo desta variável "`f1`" for maior do que o conteúdo de uma terceira variável de ponto flutuante "`f3`". Em linguagem PTX, a palavra reservada `add` é utilizada com o especificador de tipo `.f32` para indicar que a operação de adição é realizada sobre valores com precisão simples de ponto flutuante de 32 bits. A instrução `setp` é utilizada com o indicador da operação lógica de comparação `.le`, o qual representa uma comparação para menor ou igual. Além disso, o especificador de tipo `.f32` indica que as variáveis a serem comparadas apresentam precisão simples de ponto flutuante. O resultado da comparação é armazenado no registrador de predicados `p1`. Em seguida, a instrução `bra` é utilizada para realizar um salto condicional para a etiqueta `$Lt_0_1060`, retornando a execução do código para a posição anterior e, assim, implementa o laço em PTX. A execução condicional do salto é controlada pelo predicado opcional `@%p1` que antecede a palavra reservada para a instrução `bra`.

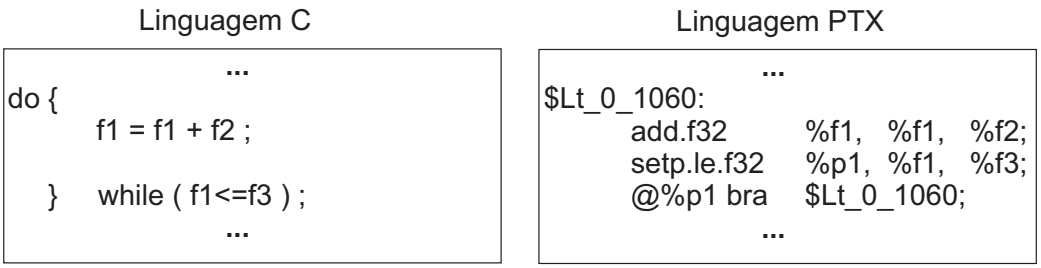

Figura 6.1: Exemplo de código em PTX. Na esquerda, trecho de código em instruções da liguagem C e na direita, o código equivalente utilizando instruções da linguagem PTX.

Conforme explicado anteriormente, o compilador *nvcc* apresenta duas etapas de compilação, onde o código fonte CUDA é compilado para as instruções da máquina virtual PTX e depois as instruções em PTX são compiladas para código de máquina da GPU. Esta segunda etapa de compilação pode ser realizada pelo *driver* CUDA em tempo de execução, compilando as instruções de PTX para código de máquina para serem executadas. Este processo de compilação em tempo de execução é chamado

de *Just In Time* – JIT. Apesar de ser uma forma rápida de compilação, ainda assim, pode causar um atraso no tempo de inicialização das aplicações.

Uma das formas de resolver este problema é através do armazenamento em disco do código binário gerado pela compilação JIT. Assim, quando o *driver* CUDA realiza uma compilação JIT de um código PTX de uma aplicação, ele automaticamente armazena uma cópia do código binário gerado, evitando repetir o processo de compilação JIT novamente para futuras execuções da mesma aplicação.

6.2 Modelo Básico

O modelo básico da metodologia Pseudo-assembly para PG em GPU segue a inspiração quântica conforme mostra a Figura 6.2. Os indivíduos quânticos são inicializados e geram uma população de indivíduos quânticos. Os indivíduos quânticos são observados, gerando os indivíduos clássicos (ou somente indivíduos). Os indivíduos clássicos são avaliados e, pelo resultado da avaliação, seleciona-se os melhores. A partir desta seleção, o operador P é aplicado para incrementar as probabilidades dos indivíduos quânticos e o processo se repete. As operações marcadas com a cor azul na Figura 6.2 são realizadas pela CPU e a avaliação dos indivíduos, marcada na cor amarela é realizada em paralelo pela GPU.

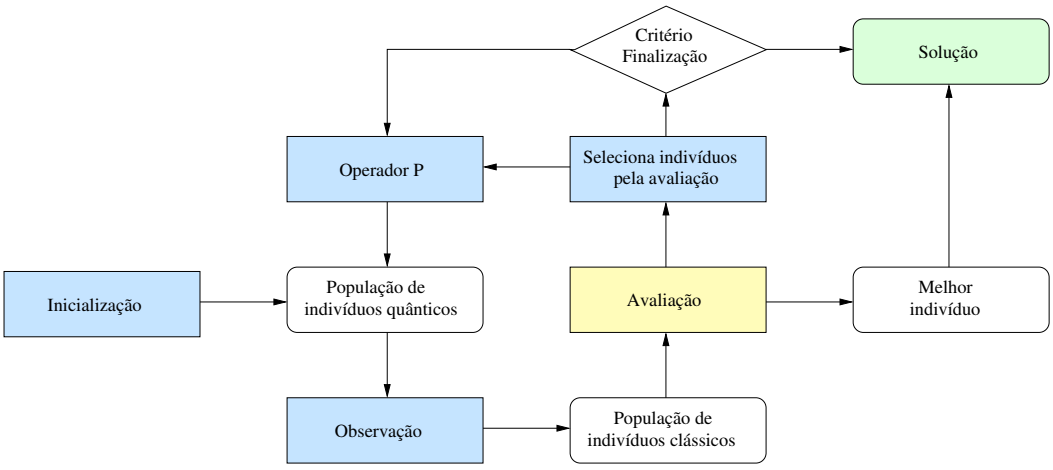

Figura 6.2: Esquema básico da PG utilizado pela metodologia Pseudo-Assembly. Em azul estão representadas as operações realizadas pela CPU e em amarelo a avaliação dos indivíduos realizada em paralelo na GPU.

Na metodologia Pseudo-assembly, a geração dos indivíduos clássicos é realizada utilizando-se as instruções da linguagem PTX. A população gerada

é composta de um conjunto de programas em PTX. A avaliação destes programas compreende a etapa mais custosa em termos computacionais do algoritmo de PG.

A avaliação dos indivíduos é realizada em paralelo explorando o processamento maciçamente paralelo oferecido pela GPU, conforme mostra a Figura 6.3. As demais funções do modelo são executadas na CPU da mesma forma que no algoritmo de QILGP [62]. Pseudo-assembly explora o paralelismo em dois níveis: no nível das amostras de dados e no nível dos indivíduos. A população inteira é avaliada em uma única chamada do *kernel*. Os indivíduos da população são avaliados em diferentes blocos de *threads*. O paralelismo de dados é empregado dentro de cada bloco de *threads*, onde todas as *threads* de um mesmo bloco executam as mesmas instruções sobre diferentes amostras de dados. Com isto, buscamos evitar a divergência de código. A Figura 6.4 ilustra estes dois níveis de paralelismo empregados na avaliação dos indivíduos na GPU na metodologia Pseudo-assembly.

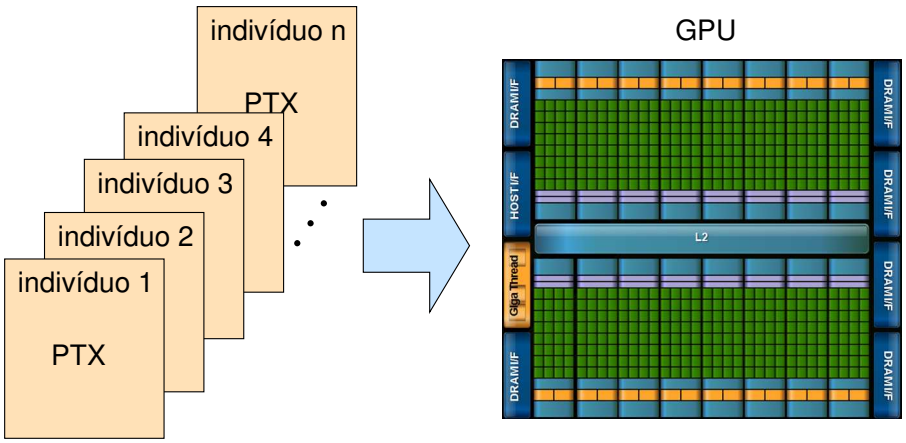

Figura 6.3: Esquema de paralelização da PG na metodologia Pseudo-Assembly. A avaliação dos indivíduos é realizada em paralelo na GPU, explorando o processamento maciçamente paralelo oferecido pela GPU.

6.3 Conjunto de Funções

O conjunto de funções utilizado por Pseudo-assembly é composto por adição, subtração, multiplicação, divisão, transferência de dados e instruções trigonométricas e aritméticas. A metodologia utiliza instruções que podem trabalhar com variáveis de entrada e valores de constantes (X), e oito registradores auxiliares ($Ri \mid i \in [0\,..\,7]$). Uma vez que a linguagem PTX não possui uma instrução atômica *exchange* (`FXCH ST(i)`), uma sequência de

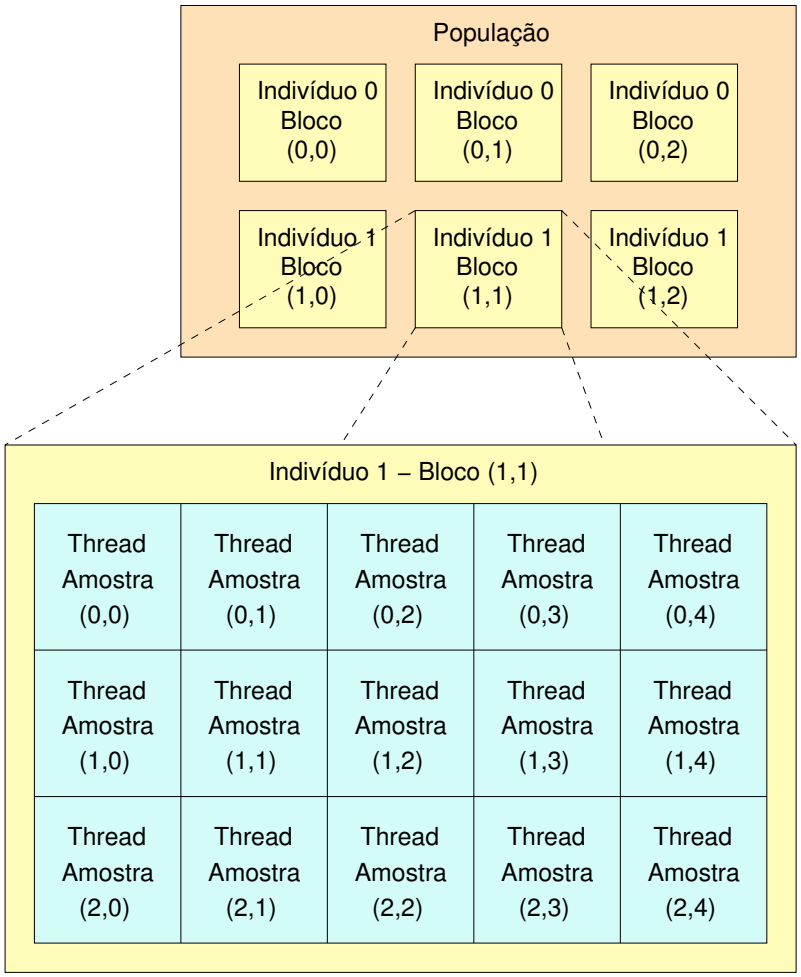

Figura 6.4: Níveis de paralelismo de Pseudo-assembly na GPU. Diferentes indivíduos são avaliados em paralelo por diferentes blocos de *threads*. Cada Bloco de *thread* avalia diferentes amostras de dados para o mesmo indivíduo em paralelo.

$mov.f32$ e um registrador auxiliar $R8$ foram utilizados para criar uma função similar. A Tabela 6.1 mostra estas instruções em linguagem PTX, as instruções equivalentes em linguagem de alto nível CUDA, a descrição da ação realizada pelas instruções e os seus argumentos.

6.4 Indivíduo

Um programa em linguagem PTX, evoluído pela metodologia Pseudo-assembly representa uma solução para um indivíduo. Ele é composto por três segmentos: cabeçalho ou *header*, corpo ou *body* e rodapé ou *footer*. O cabeçalho e o rodapé não são afetados pelo processo evolutivo. Cada um dos segmentos é descrito por:

Tabela 6.1: Descrição funcional das instruções em precisão simples de ponto flutuante. A primeira coluna apresenta o comando na linguagem CUDA; a segunda apresenta a instrução PTX; a terceira descreve a ação realizada pela instrução; e a quarta coluna apresenta o argumento utilizado pela instrução (j é utilizado para indexar as posições de memória e i é utilizado para selecionar os registradores). As últimas duas instruções, __sinf e __cosf, são instruções *fast_math*, as quais apresentam uma precisão menor e são versões mais rápidas de sinf e cosf.

CUDA	PTX	Descrição	A
		Nenhuma Operação	-
R0+=Xj ;	add.f32 R0, R0, Xj ;	$R(0) \leftarrow R(0) + X(j)$	j
R0+=Ri ;	add.f32 R0, R0, Ri ;	$R(0) \leftarrow R(0) + R(i)$	i
Ri+=R0 ;	add.f32 Ri, Ri, R0 ;	$R(i) \leftarrow R(i) + R(0)$	i
R0-=Xj ;	sub.f32 R0, R0, Xj ;	$R(0) \leftarrow R(0) - X(j)$	j
R0-=Ri ;	sub.f32 R0, R0, Ri ;	$R(0) \leftarrow R(0) - R(i)$	i
Ri-=R0 ;	sub.f32 Ri, Ri, R0 ;	$R(i) \leftarrow R(i) - R(0)$	i
R0*=Xj ;	mul.f32 R0, R0, Xj ;	$R(0) \leftarrow R(0) \times X(j)$	j
R0*=Ri ;	mul.f32 R0, R0, Ri ;	$R(0) \leftarrow R(0) \times R(i)$	i
Ri*=R0 ;	mul.f32 Ri, Ri, R0 ;	$R(i) \leftarrow R(i) \times R(0)$	i
R0/=Xj ;	div.full.f32 R0, R0, Xj ;	$R(0) \leftarrow R(0) \div X(j)$	j
R0/=Ri ;	div.full.f32 R0, R0, Ri ;	$R(0) \leftarrow R(0) \div R(i)$	i
Ri/=R0 ;	div.full.f32 Ri, Ri, R0 ;	$R(i) \leftarrow R(i) \div R(0)$	i
R8=R0 ;	mov.f32 R8, R0 ;	$R(0) \overset{\leftarrow}{\rightarrow} R(i)$ (swap)	i
R0=Ri ;	mov.f32 R0, Ri ;		
Ri=R8 ;	mov.f32 Ri, R8 ;		
R0=abs(R0) ;	abs.f32 R0, R0 ;	$R(0) \leftarrow \lvert R(0) \rvert$	-
R0=sqrt(R0) ;	sqrt.approx.f32 R0, R0 ;	$R(0) \leftarrow \sqrt{R(0)}$	-
R0=__sinf(R0) ;	sin.approx.f32 R0, R0 ;	$R(0) \leftarrow \sin R(0)$	-
R0=__cosf(R0) ;	cos.approx.f32 R0, R0 ;	$R(0) \leftarrow \cos R(0)$	-

- Cabeçalho – contém instruções em PTX para carregar os valores das variáveis de entrada da memória global para os registradores da GPU e para inicializar os oito registradores auxiliares com zero.
- Corpo – contém as instruções em PTX do programa evoluído.
- Rodapé – contém instruções em PTX para transferir o conteúdo de $R0$ para a memória global, uma vez que este é o registrador padrão de saída dos programas evoluídos. Contém também as instruções para finalizar a execução do programa, retornando o controle para o algoritmo principal de PG na CPU.

O cabeçalho possui instruções que transferem os valores das variáveis de entrada da memória global para registradores da GPU. Isto é possível

porque a GPU conta com um conjunto de registradores relativamente grande, sendo de 65536 registradores por SM na GeForce GTX TITAN. Além disso, 9 registradores por *thread* são utilizados como registradores auxiliares e o cabeçalho possui instruções para inicializar estes registradores com zero. Os valores das constantes não são carregados para registradores, quando for necessário a sua utilização, elas são lidas diretamente na memória global. Após a primeira utilização, uma cópia é mantida na cache de constantes.

Durante a avaliação, instruções como $div.full.f32$ e $sqrt.approx.f32$ podem gerar erros de execução. A instrução $div.full.f32$, quando ocorre uma divisão por zero, e a instrução $sqrt.approx.f32$, quando é realizado o cálculo de uma raiz quadrada com números negativos. Estes erros afetam diretamente o resultado da avaliação de um indivíduo e são tratados por Pseudo-Assembly. Em ambos os casos o valor atribuído como resultado da avaliação é zero ($Ri \leftarrow 0$) para cada uma das amostras de dados que apresentar divisão por zero ou cálculo de raiz quadrada com números negativos. Esta é a mesma abordagem utilizada por QILGP [62], promovendo a neutralidade na comparação dos resultados.

6.5 Kernel

A metodologia Pseudo-assembly emprega o paralelismo das GPUs para acelerar a função de avaliação. Antes de inciar o processo evolutivo, os conjuntos de dados de treinamento, validação e teste são transferidos uma única vez para a memória global da GPU e são utilizados para avaliar todos os indivíduos. A avaliação na metodologia Pseudo-assembly é realizada em duas etapas, uma de compilação JIT e uma de execução das instruções dos indivíduos. Em um primeiro momento, o código de todos os indivíduos da população, em instruções da linguagem PTX, é agrupado em um único *kernel* e compilado para código de máquina de GPUs. Em seguida, em uma segunda etapa, este código de máquina é carregado para a memória de programas da GPU e é executado em paralelo, conforme apresentado na Figura 6.4.

Quando o número de amostras de dados é menor do que o número de *threads* de um bloco, é utilizado um *grid* unidimensional, contendo um bloco para cada indivíduo. Porém, se o número de amostras de dados for maior do que o número de *threads* de um bloco, será utilizado um *grid* bidimensional, com múltiplos blocos para cada indivíduo. A identificação do código de cada indivíduo é feita através da variável interna `blockIdx.y` e cada amostra de dados é identificada por `(blockIdx.x * blockDim.x + threadIdx.x)`. Assim, é possível mapear todos os indivíduos da população para a execução

em paralelo de todas as amostras de dados na GPU, evitando a divergência de código. Os indivíduos da população são avaliados em paralelo, em diferentes blocos de *threads*. Enquanto que o paralelismo de dados é empregado dentro de cada bloco, onde todas as *threads* executam as mesmas instruções, de um mesmo indivíduo, sobre amostras diferentes de dados. Ao final, resta apenas o *overhead* da compilação JIT das instruções em PTX para código de máquina de GPUs.

6.6 Detalhes de Implementação

A utilização da linguagem PTX e da compilação JIT no processo evolutivo da PG requer alguns cuidados. Em aplicações normais, é vantajoso permitir que o *driver* CUDA armazene em disco uma cópia do código binário gerado pela compilação JIT, para evitar o *overhead* de recompilação em uma segunda execução da mesma aplicação.

No caso da PG, na abordagem Pseudo-assembly proposta neste trabalho, cada indivíduo é representado em código PTX e compilado para código de máquina de GPU para ser executado. Após a execução, a GPU retorna um valor para a avaliação de cada indivíduo.

A diferença de uma aplicação de PG para as demais aplicações é que um indivíduo que foi compilado e executado, muito provavelmente nunca mais será executado. Esta é uma característica do modelo evolutivo que altera as instruções dos indivíduos em busca de melhores soluções.

Além disso, durante o processo evolutivo, são gerados, compilados e executados milhões de indivíduos diferentes. Se uma versão em código de máquina for armazenada para cada um deles, ao final da evolução, ocupará um espaço considerável em disco. Dispendendo tempo para a escrita do código binário de cada indivíduo em disco e não apresentando a vantagem de evitar uma recompilação em execuções futuras do modelo de PG.

Após algumas execuções do modelo de PG, o número de indivíduos armazenados em disco pode ser tão grande que o tempo gasto no processo de busca por um indivíduo em código de máquina que possa ser reaproveitado, evitando uma recompilação, pode se tornar maior do que o tempo necessário para realizar uma compilação JIT.

Por estes motivos, em uma aplicação de PG, é necessário desabilitar esta função de salvar automaticamente o código binário de cada indivíduo gerado após uma compilação JIT do *driver* CUDA . Isto pode ser feito ajustando o valor da variável de ambiente "CUDA_CACHE_DISABLE" para 1.

Outro ponto a ser considerado quando estamos utilizando a linguagem

PTX em aplicações de PG é o nível de otimização do compilador. Geralmente o programador utiliza o nível de otimização mais alto para gerar o código mais otimizado e mais rápido. Entretanto, o tempo de compilação é o principal gargalo para a metodologia de Pseudo-assembly. É necessário incluir o tempo de compilação JIT dos indivíduos no processo evolutivo.

Realizamos alguns experimentos preliminares para determinar o melhor nível de otimização. Comparamos a evolução utilizando os níveis de otimização -O0 e -O4 para um *benchmark* de PG de regressão simbólica. Os resultados são apresentados na Figura 6.5. O número de gerações utilizado foi 400.000; com 6 indivíduos na população; probabilidade inicial de NOP 0,9; tamanho do passo de 0,004; e 128 instruções para o comprimento máximo do programa. Conforme pode ser observado, o nível de otimização -O0 sempre resultou em um processo evolucionário da PG mais rápido do que o nível -O4, para todos os tamanhos de problema estudados. Isto pode ser explicado pelo tempo de compilação. No caso da evolução utilizando a linguagem PTX, existem milhões de indivíduos a serem compilados e cada indivíduo será executado uma única vez durante a evolução. O código binário gerado com -O4 é mais rápido para executar as operações de ponto flutuante dos indivíduos da PG, contudo, a aceleração obtida na fase de execução não é suficiente para compensar o tempo perdido otimizando o código, na fase de compilação.

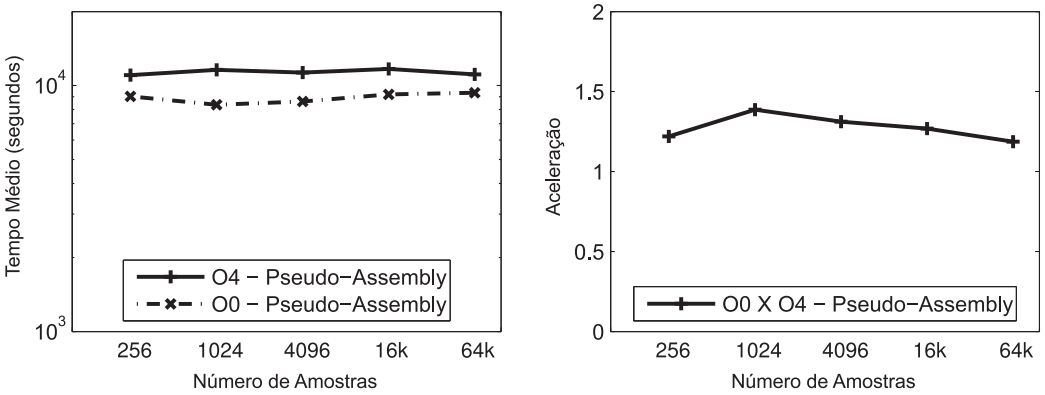

Figura 6.5: O gráfico da esquerda apresenta tempo de execução (em segundos) da metodologia Pseudo-assembly com os níveis de otimização -O0 e -O4, para o *benchmark* de regressão simólica. O gráfico da direita apresenta aceleração obtida quando a metodologia Pseudo-assembly utiliza o nível de otimização -O0, tomando o nível -O4 como referência. O número de gerações utilizado foi 400.000, com tamanho da população de 6 indivíduos e 0,9 de probabilidade inicial de NOP, tamanho do passo de 0,004 e 128 instruções para o comprimento máximo do programa.