7
Acelerando PG em GPU - Manipulando a Linguagem de Máquina

A abordagem de criação de indivíduos em código de máquina proposta neste trabalho é chamada de **GMGP** (GPU Machine-code Genetic Programming). Tal qual Pseudo-Assembly, ela também é baseada no algoritmo evolutivo com inspiração quântica QILGP [62]. Porém, os indivíduos são representados na linguagem de máquina da GPU nVidia da arquitetura Kepler, chamada CUBIN. GMGP possui três versões, que diferem na exploração do ambiente heterogêneo composto de CPU e GPU e nas otimizações empregadas. GMGP evita tanto *overhead* de compilação como o *overhead* em tempo de execução de interpretação do código.

7.1
A Linguagem CUBIN

A Linguagem CUBIN possui a representação hexadecimal para as instruções que serão executadas pelo hardware das GPUs. Um arquivo CUBIN é um arquivo que contém seções com o código executável CUDA, utilizando o formato ELF [83]. O formato ELF (*Executable and Linkable Format*) é um formato de arquivo padrão, flexível e extensível, comumente utilizado para armazenar códigos executáveis e para bibliotecas, não sendo limitado a um tipo particular de arquitetura ou processador, podendo ser empregado por sistemas operacionais diferentes, em diferentes plataformas.

O compilador *nvcc*, ao compilar código C/C++ contendo código para execução na GPU, gera arquivos CUBIN, os quais são introduzidos no arquivo executável para CPU, para que estejam disponíveis para serem transferidos para a GPU no momento em que for necessário executar as funções da GPU. Contudo, é possível gerar arquivos CUBIN separados, utilizando a opção "`-cubin`" do compilador *nvcc*.

A nVidia disponibiliza as ferramentas `cuobjdump`, `nvdisasm` e `nvprune` para desmontar arquivos CUBIN. Na saída são fornecidos mnemônicos para as instruções e estes mnemônicos são acompanhados do código hexadecimal das instruções. Porém, cada instrução de 8 bytes apresenta seu código hexadecimal dividido ao meio e a ordem dos bytes, em cada metade, é invertida. De tal forma que, se forem utilizadas para montar um programa CUBIN, na ordem em que são fornecidas na saída da ferramenta `cuobjdump`, elas não serão executadas pelo hardware da GPU.

A nVidia não disponibiliza documentação com o código hexadecimal de cada instrução que possam ser utilizados para montar programas CUBIN. Também não explica como corrigir a ordem dos bytes em hexadecimal das instruções fornecidas na saída das ferramentas disponibilizadas para desmontar um arquivo CUBIN. Assim, para realizar este trabalho, foi necessário capturar o código hexadecimal de cada instrução, realizando engenharia reversa para as instruções da arquitetura Kepler da nVidia.

A evolução em código de máquina para gerações futuras de GPUs da nVidia poderá ser realizada assim que os valores em hexadecimal de cada instrução sejam adquiridos através da metodologia proposta neste trabalho ou mesmo, se a nVidia mudar sua política externa e disponibilizar documentação pública para CUBIN.

7.1.1 Aquisição do Código de Máquina de GPUs

Desenvolvemos um procedimento semiautomático para adquirir as instruções em código de máquina de GPUs. Nosso procedimento consiste em criar um programa PTX contendo todas as instruções PTX utilizadas por GMGP (listadas na Tabela 7.1 ou na Tabela 7.2). Neste programa PTX, cada uma das instruções PTX é colocada no interior de um laço. Para este laço, não é possível prever a condição de parada antes de iniciar a sua execução. Com isso, evitamos que o compilador *ptxas*, com nível de otimização -O4, remova as instruções de interesse. A Figura 7.1 mostra um exemplo de como é implementado este laço para a instrução PTX `add`.

O programa PTX é compilado e a ferramenta *cuobjdump* da nVidia é usada para desmontar (*disassemble*) o código binário. A saída fornece todas as instruções do programa PTX convertidas para código de máquina. O desafio é remover as instruções que pertencem a cada laço de controle, e isto é feito encontrando-se um padrão que se repete ao longo do código. Uma vez que os laços de controle sejam removidos, é localizado o valor em hexadecimal que representa cada uma das instruções do conjunto.

O cabeçalho e o rodapé são obtidos através da utilização da ferramenta do Linux chamada *xxd*, a qual converte o programa em código binário para código hexadecimal, porém apresenta uma diferença em relação à ferramenta *cuobjdump*. Neste caso, o programa inteiro é convertido para hexadecimal e não apenas as instruções do corpo do programa. O cabeçalho é o código localizado antes da primeira instrução obtida por *cuobjdump* e o rodapé é o código localizado depois da última.

Tabela 7.1: Descrição funcional das instruções em precisão simples de ponto flutuante, incluindo as instruções adicionais para problemas de classificação. 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).

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 ;	$R(0) \leftrightarrows R(i)$ (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;	$R(0) \leftarrow \cos R(0)$	-
R0=(R0>Xj)?R0:Xj;	max.f32 R0, R0, Xj ;	$R(0) \leftarrow \max[R(0), X(j)]$	j
R0=(R0>Ri)?R0:Ri;	max.f32 R0, R0, Ri ;	$R(0) \leftarrow \max[R(0), R(i)]$	i
Ri=(Ri>R0)?Ri:R0;	max.f32 Ri, Ri, R0 ;	$R(i) \leftarrow \max[R(i), R(0)]$	i
R0=(R0<Xj)?R0:Xj;	min.f32 R0, R0, Xj ;	$R(0) \leftarrow \min[R(0), X(j)]$	j
R0=(R0<Ri)?R0:Ri;	min.f32 R0, R0, Ri ;	$R(0) \leftarrow \min[R(0), R(i)]$	i
Ri=(Ri<R0)?Ri:R0;	min.f32 Ri, Ri, R0 ;	$R(i) \leftarrow \min[R(i), R(0)]$	i
if (booleano) {	@!p bra $Lt_0_13829;	Se (bool) {	-
próxima instrução	próxima instrução	próxima instrução	
}	$Lt_0_13829:	} Fim	
booleano=(R0<Xj);	setp.lt.f32 p, R0,Xj;	$bool \leftarrow R(0) < X(j)$	j
booleano=(R0<Ri);	setp.lt.f32 p, R0,Ri;	$bool \leftarrow R(0) < R(i)$	i
booleano=(Ri<R0);	setp.lt.f32 p, Ri,R0;	$bool \leftarrow R(i) < R(0)$	i

Tabela 7.2: Descrição funcional das instruções booleanas. A primeira coluna apresenta o comando na linguagem CUDA; a segunda apresenta a instrução em PTX; a terceira descreve a ação realizada pela instrução; e a quarta coluna apresenta o argumento para a instrução (j é utilizado para indexar as posições de memória e i seleciona os registradores).

CUDA	PTX	Descrição	A
		Nenhuma Operação	-
R0=R0 & Xj ;	and.b32 R0, R0, Xj ;	$R(0) \leftarrow R(0) \wedge X(j)$	j
R0=R0 & Ri ;	and.b32 R0, R0, Ri ;	$R(0) \leftarrow R(0) \wedge R(i)$	i
Ri=Ri & R0 ;	and.b32 Ri, Ri, R0 ;	$R(i) \leftarrow R(i) \wedge R(0)$	i
R0=R0 — Xj ;	or.b32 R0, R0, Xj ;	$R(0) \leftarrow R(0) \vee X(j)$	j
R0=R0 — Ri ;	or.b32 R0, R0, Ri ;	$R(0) \leftarrow R(0) \vee R(i)$	i
Ri=Ri — R0 ;	or.b32 Ri, Ri, R0 ;	$R(i) \leftarrow R(i) \vee R(0)$	i
R0= $\sim$ (R0 & Xj) ;	and.b32 R0, R0, Xj ; not.b32 R0, R0 ;	$R(0) \leftarrow \overline{R(0) \wedge X(j)}$	j
R0= $\sim$ (R0 & Ri) ;	and.b32 R0, R0, Ri ; not.b32 R0, R0 ;	$R(0) \leftarrow \overline{R(0) \wedge R(i)}$	i
Ri= $\sim$ (Ri & R0) ;	and.b32 Ri, Ri, R0 ; not.b32 Ri, Ri ;	$R(i) \leftarrow \overline{R(i) \wedge R(0)}$	i
R0= $\sim$ (R0 — Xj) ;	or.b32 R0, R0, Xj ; not.b32 R0, R0 ;	$R(0) \leftarrow \overline{R(0) \vee X(j)}$	j
R0= $\sim$ (R0 — Ri) ;	or.b32 R0, R0, Ri ; not.b32 R0, R0 ;	$R(0) \leftarrow \overline{R(0) \vee R(i)}$	i
Ri= $\sim$ (Ri — R0) ;	or.b32 Ri, Ri, R0 ; not.b32 Ri, Ri ;	$R(i) \leftarrow \overline{R(i) \vee R(0)}$	i
R0= $\sim$ R0 ;	not.b32 R0, R0 ;	$R(0) \leftarrow \overline{R(0)}$	-

O próximo passo consiste em testar o valor hexadecimal que representa cada uma das instruções, usando o cabeçalho e o rodapé. Para cada instrução a ser testada, é gerado um programa que contém o cabeçalho, a instrução e o rodapé. O programa é executado e o resultado é comparado com o resultado esperado que foi previamente calculado na CPU.

7.2 GMGP

O modelo básico de GMGP segue a inspiração quântica ilustrada na Figura 6.2 do capítulo anterior. Depois que os indivíduos quânticos de uma população são observados, são gerados os indivíduos clássicos (ou somente indivíduos). Em GMGP, os indivíduos clássicos de uma geração são criados pela observação dos indivíduos quânticos, tal qual em QILGP. Cada indivíduo clássico é composto por *tokens*, os quais representam as instruções e os seus

Linguagem PTX

```
                ...
        ld.global.f32  %f1, [%rd8+0];
                ...
        ld.global.f32  %f2, [%rd10+0];
                ...
        ld.global.f32  %f3, [%rd12+0];
                ...
$Lt_0_1060:
        add.f32        %f1,  %f1,  %f2;
        setp.le.f32    %p1,  %f1,  %f3;
        @%p1 bra       $Lt_0_1060;
                ...
```

Figura 7.1: Laço utilizado para evitar a remoção da instrução PTX `add`. As variáveis utilizadas para controlar o laço possuem seu conteúdo inicializado a partir da leitura da memória global. Assim, não é possível prever, em tempo de compilação, a condição de parada deste laço, evitando que a instrução `add` seja removida pelo compilador *ptxas*, com nível de otimização -O4.

argumentos. Para cada indivíduo, GMGP varre todos os *tokens* de função e todos os *tokens* de terminal deste indivíduo, realizando a substituição pelo código de máquina correspondente de cada instrução. Cada população é composta de um conjunto de programas em linguagem de máquina que não requerem nenhuma etapa de compilação para sua avaliação. De qualquer forma, a avaliação destes programas continua sendo a etapa mais custosa em termos computacionais.

GMGP explora o paralelismo de um ambiente heterogêneo composto de CPU e GPU de duas formas diferentes: híbrida e GPU. A solução híbrida é chamada de **GMGP-h**. Ela executa as etapas de observação, ordenação dos indivíduos para avaliação e aplicação do operador P na CPU e a etapa mais custosa, a avaliação dos indivíduos, em paralelo na GPU. A solução GPU é chamada de **GMGP-gpu**. Ela implementa todas as etapas da PG na GPU. Conforme mostra a Figura 7.2. Ambas as soluções implementam a avaliação dos indivíduos na GPU. O paralelismo é explorado em dois níveis conforme na metodologia Pseudo-Assembly: no nível dos indivíduos e no nível das amostras de dados, conforme apresentado na Figura 6.4 do capítulo anterior. Buscamos também evitar a divergência de código, uma

vez que todas as *threads* de um bloco executam a mesma instrução sobre amostras de dados diferentes e indivíduos diferentes são executados por blocos de *threads* diferentes. Empregamos o máximo de paralelismo possível ao avaliar os indivíduos de uma população. Assim, esperamos que a nossa metodologia esteja pronta para explorar o paralelismo das gerações futuras de GPUs que deverão ter mais elementos de processamento do que as atuais.

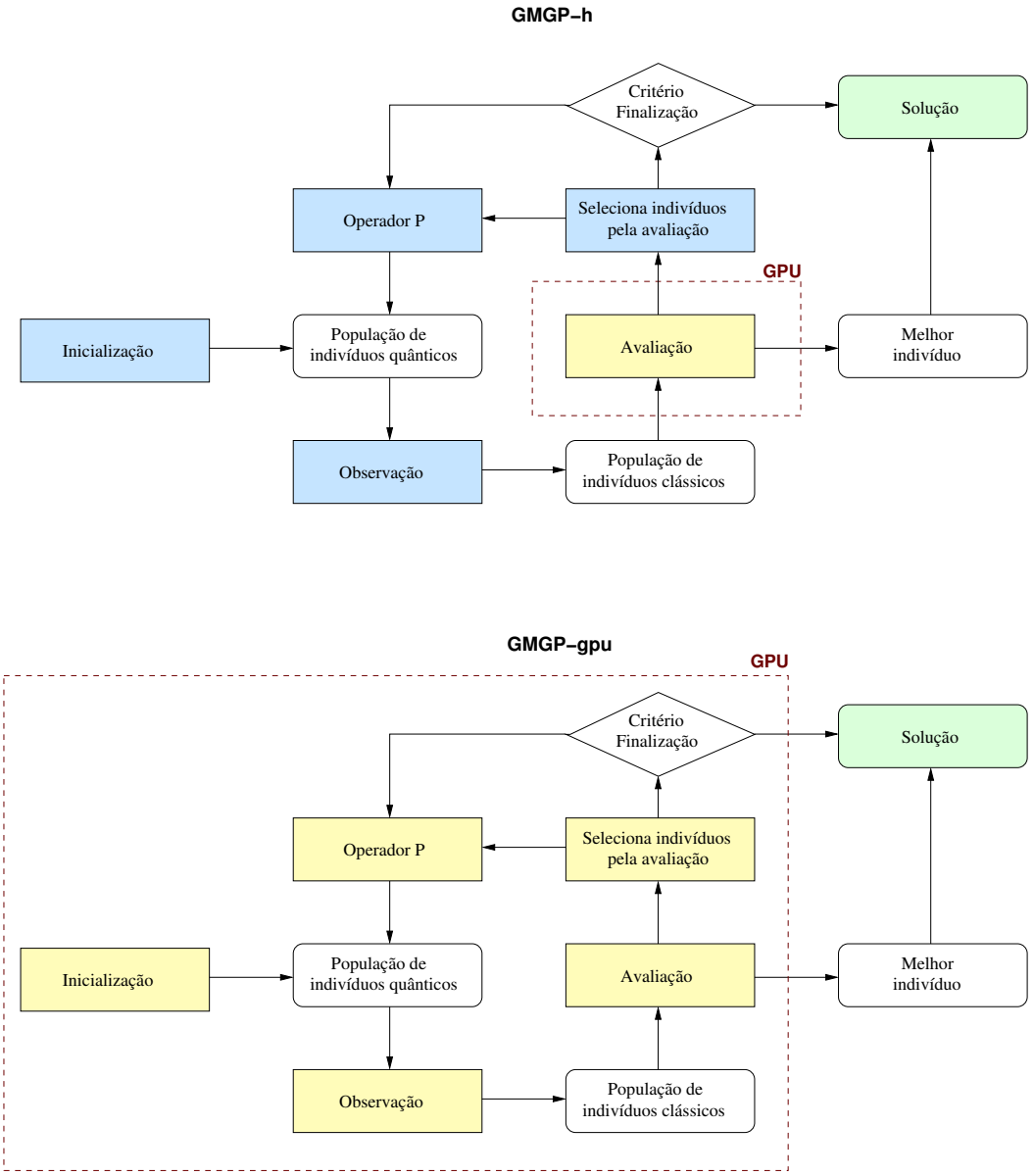

Figura 7.2: As duas soluções de GMGP para explorar o paralelismo de um ambiente heterogêneo composto de CPU e GPU: GMGP-h e GMGP-gpu. Em azul estão representadas as operações realizadas pela CPU e em amarelo as operações realizadas em paralelo na GPU.

7.3 Instruções

GMGP é capaz de evoluir sequências lineares de instruções em precisão simples de ponto-flutuante ou também, sequências lineares de instruções booleanas. O conjunto de funções para evolução de operações de ponto-flutuante é composto por adição, subtração, multiplicação, divisão, transferência de dados, instruções trigonométricas e aritméticas, máximo, mínimo, comparação e salto condicional da próxima instrução. As instruções máximo, mínimo, comparação e execução condicional da próxima instrução conferem ao algoritmo evolutivo um alto grau de não linearidade. Isto contribui para melhorar o desempenho em problemas de classificação. O conjunto de funções para evolução de operações booleanas é composto por AND, OR, NAND, NOR e NOT. A Tabela 7.1 apresenta o conjunto de instruções para operações de ponto-flutuante e a Tabela 7.2 apresenta o conjunto de instruções para operações booleanas. Cada uma destas instruções possui um número em hexadecimal que a representa e um ou dois argumentos. O argumento pode ser um registrador ou uma posição de memória. Quando o argumento é um registrador, o seu valor varia de $R0$ até $R7$. Quando é uma posição de memória, o argumento pode ser utilizado para carregar os valores das variáveis de entrada ou os valores das contantes. As instruções em hexadecimal obtidas por engenharia reversa até o momento permitem trabalhar, na metodologia de criação de indivíduos em código de máquina, com um número máximo de 512 variáveis de entrada e com um número máximo de 128 constantes pré-definidas.

Para exemplificar o formato hexadecimal utilizado para representar as instruções em código de máquina de GPUs, apresentamos na Tabela 7.3 o código CUBIN da instrução de adição com algumas de suas variações necessárias para representar as variáveis de entrada nas posições de memória (X) e as variações necessárias para representar os registradores $(Ri \mid i \in [0\,..\,7])$. Cada variação de uma instrução CUBIN, necessária para representar os seus argumentos (constantes ou registradores), tem um valor em hexadecimal diferente.

A metodologia de criação de indivíduos em código de máquina trabalha apenas com instruções em precisão simples de ponto-flutuante e desvio ou execução condicionada da próxima instrução, ou com instruções booleanas. Laços e desvios longos no código, contendo muitas instruções, não são utilizados uma vez que não são empregados nos problemas tratados neste trabalho. Entretanto, esta metodologia poderia ser estendida para considerar problemas com laços e desvios longos, incluindo mecanismos para restringir desvios para posições inválidas de memória e também para evitar laços infinitos.

Tabela 7.3: Representação hexadecimal do código de máquina de GPU da instrução add.

CUBIN (Representação hexadecimal)	Descrição	A
0x7e, 0x7c, 0x1c, **0x9**, 0x0, **0x80**, 0xc0, **0xe2**, **0x7e**, 0x7c, 0x1c, **0xa**, 0x0, **0x80**, 0xc0, **0xe2**, 0x7d, 0x7c, 0x1c, 0x0, **0xfc, 0x81**, 0xc0, 0xc2, 0x7d, 0x7c, 0x1c, 0x0, **0x0**, 0x82, 0xc0, 0xc2, 0x7d, 0x7c, 0x1c, 0x0, **0x2**, 0x82, 0xc0, 0xc2, 0x7d, 0x7c, 0x1c, 0x0, **0x4**, 0x82, 0xc0, 0xc2, 0x7d, 0x7c, 0x1c, 0x0, **0x5**, 0x82, 0xc0, 0xc2, 0x7d, 0x7c, 0x1c, 0x0, **0x6**, 0x82, 0xc0, 0xc2, 0x7d, 0x7c, 0x1c, 0x0, **0x7**, 0x82, 0xc0, 0xc2, 0x7d, 0x7c, 0x1c, 0x0, **0x8**, 0x82, 0xc0, 0xc2, 0x7d, 0x7c, 0x1c, **0x80, 0x8**, 0x82, 0xc0, 0xc2,	$R(0) \leftarrow R(0) + X(j)$	j
0x7e, 0x7c, **0x9c, 0x0f**, 0x0, 0x80, 0xc0, 0xe2, 0x7e, 0x7c, **0x1c, 0x0**, 0x0, 0x80, 0xc0, 0xe2, 0x7e, 0x7c, **0x1c, 0x3**, 0x0, 0x80, 0xc0, 0xe2, 0x7e, 0x7c, **0x9c, 0x3**, 0x0, 0x80, 0xc0, 0xe2, 0x7e, 0x7c, **0x1c, 0x4**, 0x0, 0x80, 0xc0, 0xe2, 0x7e, 0x7c, **0x9c, 0x4**, 0x0, 0x80, 0xc0, 0xe2, 0x7e, 0x7c, **0x1c, 0x5**, 0x0, 0x80, 0xc0, 0xe2, 0x7e, 0x7c, **0x9c, 0x5**, 0x0, 0x80, 0xc0, 0xe2,	$R(0) \leftarrow R(0) + R(i)$	i
0x7e, 0x7c, **0x9c, 0x0f**, 0x0, 0x80, 0xc0, 0xe2, **0x2**, 0x7c, **0x1c, 0x0**, 0x0, 0x80, 0xc0, 0xe2, **0x1a**, 0x7c, **0x1c, 0x3**, 0x0, 0x80, 0xc0, 0xe2, **0x1e**, 0x7c, **0x9c, 0x3**, 0x0, 0x80, 0xc0, 0xe2, **0x22**, 0x7c, **0x1c, 0x4**, 0x0, 0x80, 0xc0, 0xe2, **0x26**, 0x7c, **0x9c, 0x4**, 0x0, 0x80, 0xc0, 0xe2, **0x2a**, 0x7c, **0x1c, 0x5**, 0x0, 0x80, 0xc0, 0xe2, **0x2e**, 0x7c, **0x9c, 0x5**, 0x0, 0x80, 0xc0, 0xe2,	$R(i) \leftarrow R(i) + R(0)$	i

7.4 Indivíduo

Um programa em código de máquina de GPUs, evoluído pela metodologia GMGP representa uma solução para um indivíduo. Este programa consiste, tal qual em Pseudo-assembly, de três segmentos: cabeçalho, corpo e rodapé. O cabeçalho e o rodapé são os mesmos para todos os indivíduos e durante todo o processo evolucionário, sendo otimizados da mesma forma que no compilador da nVidia. Cada um dos segmentos é descrito por:

– Cabeçalho – contém instruções em código de máquina ou CUBIN 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 código de máquina ou CUBIN do programa evoluído.
– Rodapé – contém instruções em código de máquina ou CUBIN 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 de PG.

Para cada indivíduo, o programa em código de máquina é montado através do empilhamento das instruções em código hexadecimal na mesma ordem que os *tokens* da PG são lidos. Não há a necessidade de comparações e desvios dentro do programa de um indivíduo porque as instruções são executadas sequencialmente. Evitar comparações e desvios é uma característica evolutiva importante da nossa metodologia. Conforme foi explicado anteriormente, as GPUs são particularmente sensíveis aos desvios condicionais.

Além disso, agrupamos o corpo de todos os programas de uma mesma população em um único *kernel* de GPU. Este *kernel* possui um único cabeçalho e único rodapé, reduzindo o volume total de bytes de uma população em código de máquina e assim reduzindo o tempo necessário para transferir o programa para a memória da GPU através do barramento PCIe.

7.5 GMGP-h

Na abordagem GMGP-h, a função de avaliação é paralelizada na GPU. O restante da PG é executado na CPU utilizando QILGP. Antes de iniciar o processo evolucionário, os conjuntos de dados de treinamento, validação e teste são transferidos para a memória global da GPU. Os indivíduos clássicos gerados por QILGP em CUBIN são carregados para a memória de programas da GPU e são executados em paralelo.

A população inteira de indivíduos é avaliada em uma única chamada do *kernel*. Quando o número de amostras de dados é menor do que o número de *threads* em um bloco, mapeamos um indivíduo por bloco. Para os problemas que possuem um número de amostras de dados maior do que o número de *threads* de um bloco, um *grid* bidimensional é utilizado e cada indivíduo é mapeado em múltiplos blocos de *threads*. Cada indivíduo é identificado pela variável interna `blockIdx.y` e cada amostra de dados é identificada por `(blockIdx.x * blockDim.x + threadIdx.x)`. Para manter o código de todos os indivíduos dentro de um único *kernel* de GPU, usamos instruções em código de máquina com funcionalidade equivalente ao **if** da linguagem C. Com isto,

é possível fazer a distinção entre o código de cada indivíduo através do valor de `blockIdx.y`. Entretanto, a utilização destas instruções para distinguir os indivíduos não introduz divergência dentro de um bloco de *threads* porque todas as *threads* de um bloco seguem o mesmo caminho de execução. Esta metodologia possibilita a execução eficiente da avaliação dos indivíduos.

O processo de avaliação trata os problemas causados pelos erros de execução, tais como divisão por zero ou raiz quadrada de números negativos, afetando diretamente a avaliação de um programa evoluído. Em ambos os casos, o valor atribuído para o resultado é zero ($Ri \leftarrow 0$), e esta é a mesma abordagem empregada por QILGP [62].

7.6 GMGP-gpu

Na abordagem GMGP-gpu, a execução em código de máquina da função de avaliação na GPU é realizada da mesma forma que em GMGP-h. Porém, as demais etapas do modelo evolutivo também são paralelizadas para a GPU. A principal motivação desta metodologia é tornar o tempo de execução do modelo de PG menos dependente do número de instruções utilizadas para o tamanho máximo do programa.

GMGP-gpu possui 5 *kernels*, conforme apresentado na Figura 7.3. O primeiro kernel é responsável pela inicialização dos indivíduos quânticos. Cada um dos indivíduos quânticos é representado por três matrizes de valores com precisão simples de ponto flutuante (32 bits): S_f que representa a superposição dos estados para os *qudits* de função (QFs); S_{tm} que representa a superposição dos estados para os *qudits* de terminal (QTs) de memória; e S_{tr} que representa a superposição dos estados para os *qudits* de terminal (QTs) de registradores. As dimensões das três matrizes são dadas por: dimensão(S_f) = comprimento máximo do programa × número de funções; dimensão(S_{tm}) = comprimento máximo do programa × (número de variáveis de entrada + número de constantes); e dimensão(S_{tr}) = comprimento máximo do programa × número de registradores auxiliares.

O primeiro kernel carrega para cada indivíduo i, as três matrizes $S_{f\ i}$, $S_{tm\ i}$, e $S_{tr\ i}$ agrupadas em uma única matriz S_i cuja dimensão é dada por: dimensão(S_i) = (comprimento máximo do programa × número de indivíduos) × (número de funções + número de variáveis de entrada + número de constantes + número de registradores auxiliares).

Assim, em uma única chamada de *kernel*, todos os QFs e QTs de memória e registradores de todos os indivíduos da população são inicializados em paralelo. Neste kernel, o bloco de *threads* é unidimensional com o número de

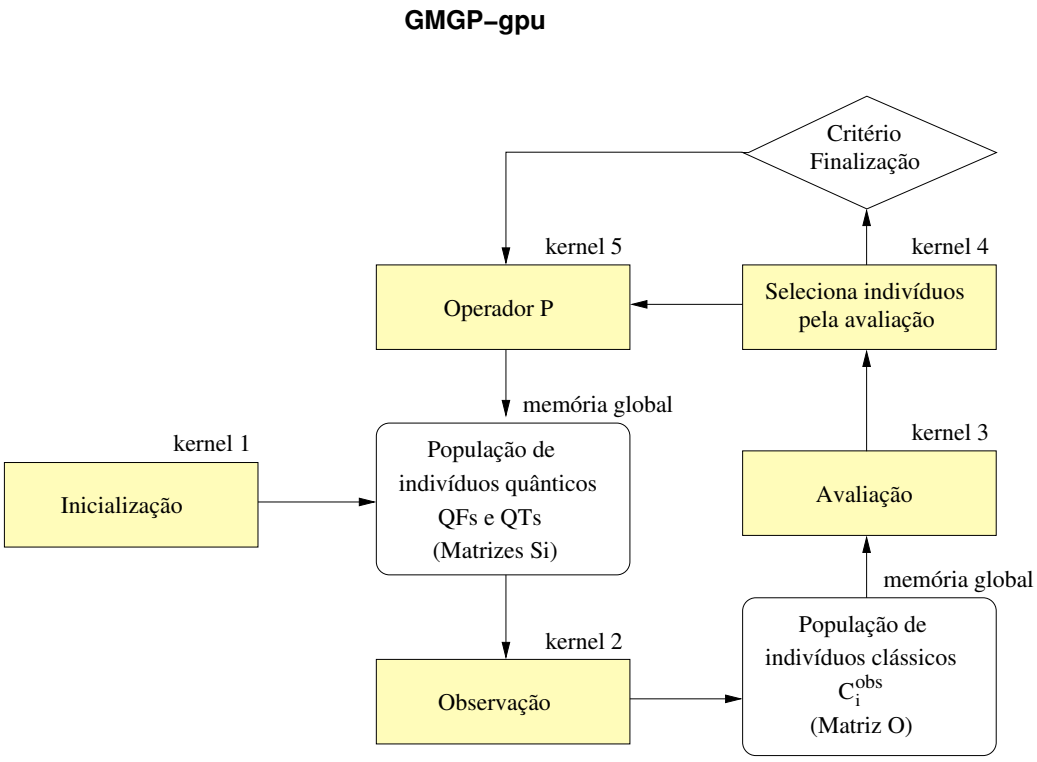

Figura 7.3: Metodologia GMGP-gpu: toda a PG é executada dentro da GPU por diferentes *kernels*.

threads igual ao comprimento máximo do programa. O *grid* é bidimensional e seu tamanho é dado por: número de indivíduos × (número de funções + número de variáveis de entrada + número de constantes + número de registradores auxiliares)).

O segundo *kernel* realiza a observação de M indivíduos quânticos para gerar M indivíduos clássicos observados C_i^{obs}. Cada um dos M indivíduos clássicos observados C_i^{obs} é representado por uma matriz O_i de caracteres (8 bits), cuja dimensão é dada por: dimensão(O_i) = comprimento máximo do programa × 2. A primeira linha desta matriz armazena os **Tokens de Função** (TFs) e segunda linha armazena os **Tokens de Terminal** (TTs). Na memória da GPU, uma matriz maior O é criada para representar todos os M indivíduos clássicos observados C_i^{obs}. A matriz O possui dimensão dada por: dimensão(O) = (comprimento máximo do programa × número de indivíduos) × 2. Desta forma, em uma única chamada de *kernel* são observados todos os TFs e TTs de todos os novos indivíduos clássicos gerados. O bloco de *threads* é unidimensional com tamanho igual ao comprimento máximo do programa. O *grid* também é unidimensional com tamanho igual ao número de indivíduos. Cada *thread* realiza a observação de um TF e em seguida, com o resultado deste TF, realiza a observação de um TT. Não é possível realizar a observação

do TF e do TT em paralelo porque o valor do TF é necessário para selecionar o QT, entre QT de memória ou QT de registrador.

O terceiro *kernel* realiza a avaliação dos indivíduos em paralelo da mesma forma que foi descrito para GMGP-h. Após a avaliação, o quarto *kernel* realiza a seleção dos indivíduos para a próxima geração.

O quarto *kernel* seleciona os indivíduos através da ordenação das suas avaliações. As avaliações dos M indivíduos clássicos da população atual C_i, juntamente com as avaliações dos novos M indivíduos clássicos observados C_i^{obs}, são ordenadas do menor para o maior. O *grid* possui um único bloco, o qual possui um número de *threads* igual ao número de indivíduos da população. Cada *thread* realiza a comparação de dois valores de avaliações, realizando a substituição da ordem dos indivíduos sempre que encontrar um indivíduo com um valor menor para a avaliação. Este processo é repetido iterativamente até que todas as avaliações estejam ordenadas. Após a ordenação, são mantidos na população de indivíduos clássicos, os M indivíduos que apresentarem os menores valores para as avaliações, sendo os demais descartados. A substituição dos indivíduos é realizada através da substituição de seus valores para os *tokens* de função e para os *tokens* de terminal.

Após a atualização da população de indivíduos clássicos, o quinto *kernel* atualiza os indivíduos quânticos. Esta atualização é realizada para os *qudits* de função (QFs) e os *qudits* de terminal (QTs) de memória e de registradores para todos os indivíduos quânticos da população. O *kernel* efetua a atualização aplicando o operador P da equação 4-5. É utilizado um bloco de *threads* unidimensional com tamanho igual ao comprimento máximo do programa. A *grid* é bidimensional com tamanho (número de indivíduos $\times$ 2); Cada *thread* é responsável por atualizar um único *qudit* de função QF ou um único *qudit* de terminal (QT). O valor de cada TF do indivíduo clássico é utilizado para selecionar a probabilidade p_i de cada QF do indivíduos quântico que deverá ser atualizada de acordo com o operador P. Da mesma forma, o valor de cada TT do indivíduo clássico é utilizado para selecionar a probabilidade p_i de cada QT do indivíduo quântico, para a qual será realizada a atualização de acordo com o operador P.

Em GMGP-h, quando o número de instruções do indivíduo aumenta de, por exemplo, 128 para 512, o tempo de execução do modelo de PG na CPU aumenta na mesma proporção. Para GMGP-gpu, a execução do modelo de PG com 128 instruções não possui um volume de dados suficientemente grande para ocupar todos os SPs da GPU. Nem mesmo a execução com 512 instruções. Assim, espera-se que ao aumentar o número de instruções de 128 para 512, o efeito seja o de aumentar o número de SPs utilizados e não o de

aumentar o tempo de execução.

7.7 GMGP-gpu+

GMGP-gpu+ é uma versão otimizada de GMGP-gpu. A otimização proposta foi implementada no *kernel* que seleciona os indivíduos para a próxima geração. Neste *kernel*, propomos para GMGP-gpu+ um novo algoritmo de seleção. Conforme mostra a Figura 7.4.

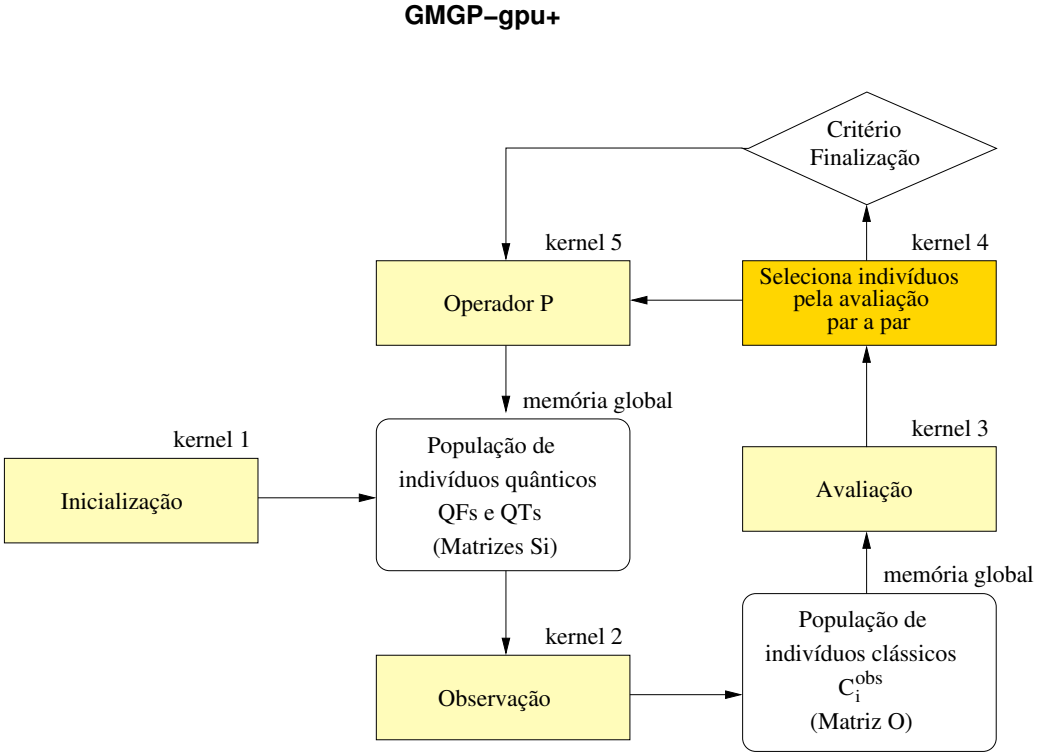

Figura 7.4: Metodologia GMGP-gpu+: toda a PG é executada dentro da GPU por diferentes *kernels*.

Neste algoritmo, a seleção dos indivíduos para a população de indivíduos clássicos é feita através da comparação par a par. Assim, cada um dos M indivíduos clássicos da população atual C_i é comparado com o seu respectivo indivíduo dentre os M indivíduos clássicos observados C_i^{obs}. Se o indivíduo observado C_i^{obs} possuir um menor valor para a avaliação, ou se as avaliações forem idênticas e com um menor comprimento do programa, então ele é utilizado para substituir o indivíduos clássico da população atual C_i. O *grid* possui um único bloco, o qual possui um número de *threads* igual ao número de indivíduos da população. Cada *thread* é utilizada para realizar a comparação dos valores das avaliações de dois indivíduos, um da população atual e um observado. Quando for necessária a substituição dos indivíduos, esta é realizada

pela substituição de seus valores para os *tokens* de função e para os *tokens* de terminal.

7.8 Detalhes de Implementação

A ferramenta `cuobjdump` fornecida pela nVidia foi utilizada para desmontar arquivos CUBIN, com o objetivo de adquirir o código de máquina de cada instrução. Na saída desta ferramenta é fornecido um mnemônico e um valor em hexadecimal para cada instrução. Por exemplo, o mnemônico `FADD.FTZ R0, R0, R4;`, de uma instrução que adiciona para o conteúdo do registrador `R0` o valor do registrador `R4`, é acompanhada do seguinte código hexadecimal `0xe2c08000021c0002`. Este código hexadecimal é composto por 8 bytes que representam uma instrução em código de máquina para GPUs da nVidia. Porém, a utilização deste código na ordem em que é fornecido pela ferramenta `cuobjdump` para montar um programa CUBIN resulta num código que não pode ser executado pala GPU. Após vários testes, chegamos a conclusão de que é necessário dividir cada instrução ao meio, inverter a ordem das metades e inverter a ordem de cada byte, em cada uma das metades, resultando no seguinte código hexadecimal `0x2, 0x0, 0x1c, 0x2, 0x0, 0x80, 0xc0, 0xe2`, o qual pode ser utilizado para montar programas CUBIN que serão executados pela GPU.

Após cada programa em código de máquina ter sido montado, ele precisa ser carregado para a memória de programas da GPU para ser executado em paralelo. A única opção fornecida pela nVidia para carregar um programa em código de máquina para a memória de programas é através da utilização da função `cuModuleLoadDataEx`. Porém esta função realiza a leitura de um programa em código de máquina que se encontra na memória RAM da CPU. Assim, nas versões GMGP-gpu e GMGP-gpu+ é necessário transferir os *tokens* da GPU para CPU, realizar a montagem dos indivíduos em código de máquina (transformar *tokens* em programa CUBIN), e em seguida chamar a função `cuModuleLoadDataEx`. É realizada a transferência de *tokens* da GPU para a CPU, ao invés de montar o indivíduo em código de máquina na GPU, porque o tempo de transferência de *tokens* é menor. Cada instrução é representada por dois *tokens*, um de função e um de terminal, que ocupam dois bytes. Em código de máquina, cada instrução é representada por oito bytes.