DTM@GPU: Explorando redundância de traços em GPU

Saulo T. Oliveira[1], Leandro Santiago[1], Brunno F. Goldstein[1], Felipe M. G. França[1], Maria Clicia S. de Castro[2], Alexandre S. Nery[2], Alexandre C. Sena[2], Igor M. Coelho[2],Tiago A. O. Alves[2], Leandro A. J. Marzulo[2], Cristiana Bentes[3]

[1]Programa de Engenharia de Sistemas e Computação - COPPE
Universidade Federal do Rio de Janeiro (UFRJ), Rio de Janeiro, Brasil

[2]Instituto de Matemática e Estatística
Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, Brasil

[3]Faculdade de Engenharia
Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, Brasil

{sto, lsantiago, bfgoldstein, felipe}@cos.ufrj.br,
{clicia, anery, asena, igor.machado, tiago, leandro}@ime.uerj.br,
cris@eng.uerj.br

Abstract. *During a program execution, usually identical instructions, that have the same input parameters and produce the same output, execute repeatedly. This redundancy allows the reuse of previous computations, reducing the program execution time. The DTM (Dynamic Trace Memoization) technique exploits trace reuse, where a trace comprises a sequence of redundant instructions, enabling the trace to be completely skipped when its input set repeats over time. DTM showed promising speedups with trace reuse, however, it was originally proposed for superscalar CPUs. Recently, GPUs have been used to accelerate computationally intensive applications with an excellent cost/performance ratio. This paper proposes the implementation of the DTM technique in the GPU architecture. The proposal includes architectural modifications and the identification of reuse characteristics in multithreaded environments. Trace reusing was evaluated in the GPGPU-sim simulator and promising results were found for a set of 9 real-world applications: approximately 35.3% of the instructions are reused, given an estimated 10% of performance gain.*

Resumo. *Durante a execução de um programa, usualmente instruções idênticas, que recebem os mesmos dados de entrada e que produzem as mesmas saídas, são executadas repetidamente. Esta redundância permite a reutilização de cálculos anteriores, reduzindo o tempo de execução do programa. A técnica DTM (Dynamic Trace Memoization) explora o reúso de traços, compostos por uma sequência de instruções redundantes, possibilitando evitar o processamento de todas as suas instruções uma só vez. DTM mostrou acelerações promissoras com o reúso de traços, porém, foi originalmente proposta para CPUs superescalares. Recentemente, GPUs (Graphics Processing Units) têm sido usadas para acelerar aplicações computacionalmente intensivas com uma excelente relação custo/desempenho. Este trabalho propõe a implementação da técnica DTM na arquitetura da GPU. A proposta inclui mudanças arquiteturais*

e a identificação de características de reúso em ambientes multithread. O reúso de traços foi avaliado no simulador GPGPU-sim e resultados promissores foram encontrados para um conjunto de 9 aplicações reais: aproximadamente $35,3\%$ *das instruções são reutilizadas apontando para uma estimativa de* 10% *de ganho de desempenho.*

1. Introdução

Um fenômeno interessante observado em estudos anteriores [Sodani and Sohi 1998, Lipasti et al. 1996, Sazeides and Smith 1997] é que durante a execução de um programa há uma quantidade significativa de instruções que executam repetidamente com as mesmas entradas, produzindo, dessa forma, os mesmos resultados. Mais especificamente, se uma instrução i recebe como operandos o_1 e o_2 e gera como resultado r_i, durante a execução do programa, se i for executada outras vezes com o_1 e o_2 produzirá r_i novamente.

Uma abordagem para explorar esta redundância é o reúso de processamento, que permite armazenar resultados de computações redundantes e evitar o reprocessamento de instruções, blocos básicos, traços de instruções ou até mesmo funções. A técnica de reúso de traços DTM *(Dynamic Trace Memoization)*, proposta em [da Costa et al. 2000], é capaz de explorar o reúso em sequências (traços) de instruções redundantes, assim como em instruções redundantes individuais. DTM permite identificar oportunidades de reúso de forma dinâmica, obtendo taxas de reúso maiores que as estratégias anteriormente propostas.

A estratégia DTM, porém, explora o reúso em programas que executam em uma CPU. Atualmente, há uma inegável tendência de se utilizar computação heterogênea com auxílio de aceleradores. Aceleradores como as GPUs (*Graphic Processing Units*) têm sido amplamente adotados nas arquiteturas modernas, provendo ganhos de desempenho significativos. Programas escritos para a GPU, contudo, possuem características distintas de programas escritos para CPU e suas execuções seguem o paradigma SIMT *(Single Instruction Multiple Threads)*.

Este trabalho propõe o uso da técnica DTM em GPUs. O objetivo principal é avaliar o potencial de ganho de desempenho com a exploração de reúso de processamento em GPUs. A técnica DTM foi adaptada para o modelo arquitetural da GPU da NVIDIA e implementada no simulador GPGPU-sim [Bakhoda et al. 2009]. Além disso, com foco em maximizar o desempenho em ambientes *multithread* foram elaboradas diferentes formas de reúso: *intra-thread*, *inter-thread* e traços redundantes.

Os experimentos foram realizados em diferentes classes de aplicações e os resultados obtidos mostram, em média harmônica, até $35,3\%$ de reúso, no qual $66,4\%$ foi proveniente do reúso *inter-thread*, $23,1\%$ do reúso de traços redundantes e o restante foi obtido do reúso *inter-thread*. A estimativa de *speedup* alcançou a ordem de $10,7\%$.

O restante deste trabalho está organizado da seguinte forma: a técnica DTM aplicada em CPUs é apresentada na Seção 2. A Seção 3 explica a implementação da técnica DTM em GPU. A Seção 4 apresenta os experimentos e resultados. A Seção 5 apresenta as conclusões e os trabalhos futuros.

2. *Dynamic Trace Memoization* (DTM)

O *Dynamic Trace Memoization* (DTM) é uma técnica de reúso de traços [da Costa et al. 2000] que evita a reexecução das operações contidas em traços redundantes. Traços são formados por uma sequência de instruções cujos operandos de entrada se repetem ao longo da execução do programa. Tabelas de reúso que armazenam o contexto das instruções e traços são usadas para atualizar o estado do processador quando as entradas se repetem em execuções futuras, evitando assim execuções redundantes.

O DTM utiliza duas tabelas de reúso: a MTG (*Memo Table G*) para armazenar dados de instruções redundantes e a MTT (*Memo Table T*) usada na construção dinâmica e reúso de traços. A MTG armazena o contador de programa, valores de entrada e saída, além do estado do preditor de desvio para cada instrução. Já a MTT armazena o contador de programa do início e fim do traço, as identificações dos registradores de entrada e saída e seus valores, além do estado do preditor de desvio. O mecanismo define um conjunto de instruções válidas (que podem ser candidatas para o reúso) que normalmente exclui operações de ponto flutuante (cujo valor tem menor probabilidade de repetição) e de acesso à memória (por possuírem efeitos colaterais).

Durante a decodificação de uma instrução válida, o DTM verifica se existe uma linha correspondente na MTG cujo conjunto de entrada coincida com as entradas correntes. Caso as entradas sejam inéditas elas são armazenadas na MTG junto com o resultado produzido. Caso as entradas sejam redundantes, o resultado é reusado. Além disso, quando uma instrução armazenada na MTG é reusada pela primeira vez é iniciada a construção de um traço, que será composto por múltiplas instruções redundantes. A construção do traço é terminada quando uma instrução inválida for encontrada ou quando ocorrer a execução de uma instrução sem reúso. É importante ressaltar que a busca por reúso ocorre primeiro na MTT e depois na MTG, ou seja, primeiro tenta-se buscar um traço redundante e depois uma instrução redundante.

Com base no DTM, em [Pilla et al. 2004], [Pilla et al. 2003] e [Pilla et al. 2006], foi proposto e estudado um mecanismo de execução especulativa que alcança um melhor reúso dos traços em CPUs superescalares. Paralelamente, em [Silva et al. 2005], foi aplicado a técnica DTM na arquitetura da máquina virtual Java.

3. *Dynamic Trace Memoization* em GPU

A fim de beneficiar-se do poder de paralelismo maciço oferecido pelas GPUs e do potencial de reúso de instruções da técnica DTM, este trabalho propõe o modelo DTM@GPU. Neste modelo, traços presentes em ambientes *multithread* são identificados e reutilizados por meio da implementação do mecanismo DTM, como mostra a Figura 1.

Considerando que os traços dinâmicos gerados pela execução de kernels são registrados, propõe-se que cada linha de um traço possua um conjunto de informações sobre as instruções executadas em um determinado *core* da GPU. Tais informações, listadas na Tabela 1, alimentam o DTM@GPU que, por sua vez, realiza a execução e análise de reúso de traços considerando a arquitetura da GPU.

A técnica DTM foi criada originalmente para CPUs. Aplicar simplesmente seu modelo às GPUs, sem qualquer alteração, seria inviável. Tais arquiteturas possuem um controle de fluxo simplificado quando comparado aos da CPUs. A ausência de preditores

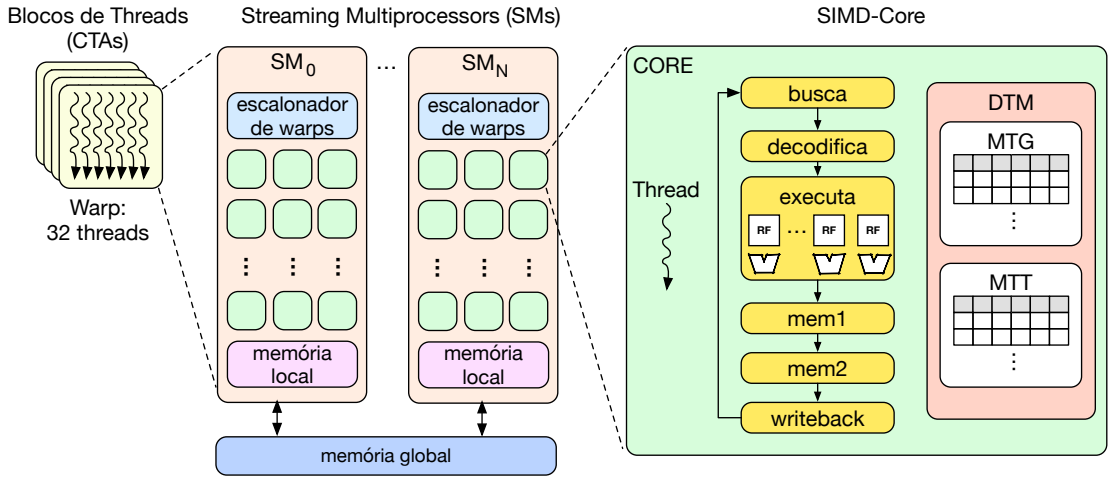

Figura 1. Visão geral da arquitetura da GPU com o mecanismo DTM.

Tabela 1. Conjunto de informações de cada instrução executada em um determinado *core* da GPU.

Item	Descrição
laneid	Posição da *thread* no *warp*, indica qual core da GPU executou a *thread*
num1	Contagem de instruções executadas globalmente
uid	Identificador único da *thread*
CTA	Coordenada x,y,z do CTA (*Cooperative Thread Array*) no Grid
thread	Coordenada x,y,z do *thread* no CTA
num2	Contagem de instrução executadas por *thread*
pc	Contador de programa da instrução
op	Mnemônico da operação
regs	Vetor de registradores
values	Vetor de valores

de desvios em GPUs caracteriza bem essa diferença. Dentre as alterações arquiteturais propostas em DTM@GPU, destacam-se as realizadas nas tabelas MTG e MTT. Cada entrada na tabela MTG e MTT passa a ser composta pelos campos descritos na Figura 2.

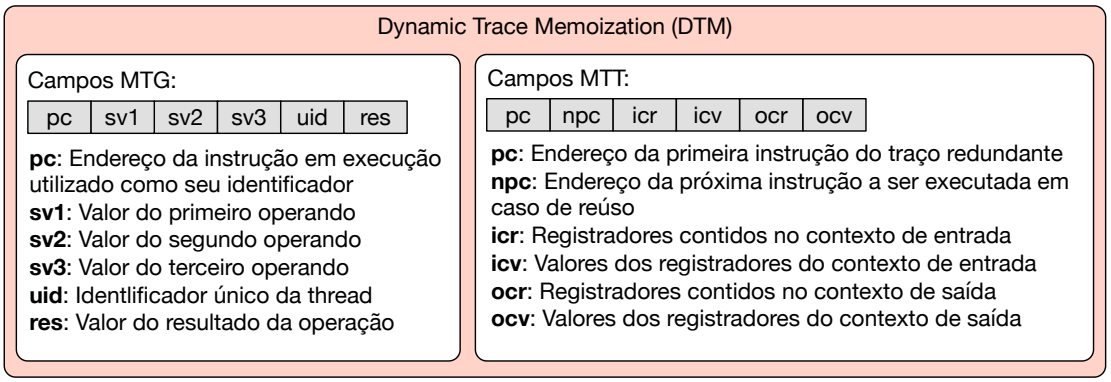

Figura 2. Campos das tabelas MTG e MTT.

Como não existe predição de desvios em GPU, os campos que armazenam o estado do preditor de desvios foram removidos da MTG e MTT originais. Por outro lado, foram adicionados os campos *uid* e *sv3*. O campo *uid* é responsável pela identificação da *thread* que executou a tarefa. Tal informação é importante para garantir que um determinado traço seja criado apenas por instruções válidas de uma mesma *thread*. O campo *sv3* possibilita o reúso de instruções com até três operandos explícitos, ou instruções com dois operandos e um predicativo. Da mesma forma que na MTG, os campos referentes à predição de desvios da MTT original foram removidos. Nenhum novo campo é adicionado à MTT, permitindo um melhor aproveitamento do *buffer* disponível para formação de traços.

3.1. Conjunto de Instruções

DTM@GPU analisa uma gama maior de instruções quando comparada à versão DTM para CPUs. Este fato ocorre devido ao conjunto específico de instruções da arquitetura de GPUs. Com isso, os conjuntos de instruções válidas e inválidas devem ser reestruturados. Dado que o novo conjunto de instruções válidas resulta do complemento do conjunto de instruções inválidas no universo de instruções existentes, o conjunto de instruções inválidas é definido como:

1. **Instruções de sincronização** de *thread* como *bar* e *exit*. Utilizadas para forçar a convergência das *threads*, elas definem barreiras onde as *threads* devem aguardar as outras *threads* do mesmo CTA antes de prosseguir com sua execução. A instrução *exit*, diferente da instrução *bar*, marca o término da execução da *thread*. O reúso deste tipo de instrução poderia acarretar a perda do ponto de sincronização levando algumas *threads* a acessarem posições de memória antes do tempo correto;
2. **Instruções de acesso à memória** como *ld* e *st*. Assim como no DTM, as instruções de *load* e *store* não são consideradas válidas devido aos custos associados à manutenção das tabelas de memorização;
3. **Instruções de textura e superfície** como *tex*, *txq*, *tld4*, *suld*, *sust*, *sured* e *suq* possuem as mesmas penalidades presentes nas instruções de acesso à memória;
4. **Instruções de ponto flutuante** como as marcadas com os modificadores *f16*, *f32* e *f64*. Seguem o mesmo conceito utilizado no DTM original, onde sua baixa localidade dos valores de ponto flutuante inviabiliza a implementação da memorização.

3.2. Distribuição das MTGs e MTTs

Diferente do mecanismo DTM original, onde existe apenas um par de tabelas e um fluxo de execução, no DTM@GPU cada *core* armazena um par de tabelas e existem centenas de fluxos de execução paralelos distribuídos pelos *cores* . As *threads*, além de salvarem seu contexto, possuem também um *buffer* temporário, utilizado para construção dos traços antes deles serem adicionados às MTTs. O mesmo algoritmo DTM@GPU é executado em cada *core*. As várias *threads* que compartilham o mesmo *core*, trocam informações por intermédio das tabelas de memorização. O conjunto de suas interações resulta no reúso de instruções no sistema paralelo. Este algoritmo permite, sempre que possível, aproveitar instruções já executadas, mesmo que em fluxos diferentes.

A opção de distribuir as tabelas por *cores* resultou do fato de que *warps* executam concorrentemente dentro dos *cores*. Com isso, um número maior de *threads* poderia se

beneficiar do reúso de instruções e traços previamente identificados no *core*. A utilização de apenas duas tabelas para toda a GPU, como ocorre na versão original DTM, seria inviável, pois implicaria em uma grande quantidade de sincronismo para o acesso a elas. Já o oposto, onde cada *thread* possui suas tabelas, anularia o reúso entre as *threads* e aumentaria o consumo de recursos disponíveis. O número de tabelas MTG e MTT utilizadas no DTM@GPU equivale à quantidade total de *cores* da GPU.

3.3. Tipos de Reúso

Da mesma forma que a técnica DTM original, o DTM@GPU identifica instruções dinâmicas redundantes ao longo da execução de uma determinada aplicação. Porém, devido ao ambiente paralelo das GPUs, a versão DTM@GPU é capaz de identificar e reutilizar instruções redundantes dentro de uma *thread* específica ou entre *threads* que executam no mesmo *core*. Tais instruções são rotuladas como *redundante intra-thread* e *redundante inter-thread* respectivamente.

A Figura 3 mostra um exemplo de formação de traços. As instruções *não redundantes* são representadas por círculos brancos, as instruções *redundantes intra-thread* são representadas por círculos cinzas com borda simples e as instruções *redundantes inter-thread* são representadas por círculos cinzas com borda dupla. A Tabela 2 lista as condições necessárias para que ocorra um determinado tipo de rotulação.

Tabela 2. Classificação das instruções para criação da MTG no DTM@GPU.

Instr. Válida	Presente MTG	Op. Iguais	UID Igual	Ação
falso	não avaliado	não avaliado	não avaliado	Marcar instrução como não redundante
verdadeiro	falso	não avaliado	não avaliado	Marcar instrução como não redundante e inserir na MTG
verdadeiro	verdadeiro	falso	não avaliado	Marcar instrução como não redundante e inserir na MTG
verdadeiro	verdadeiro	verdadeiro	falso	Marcar instrução como redundante *inter-thread* e inserir na MTG
verdadeiro	verdadeiro	verdadeiro	verdadeiro	Marcar instrução como redundante *intra-thread* e inserir na MTG

3.4. Construção de Traços Redundantes

Em DTM@GPU traços redundantes são formados, exclusivamente, por duas ou mais instruções, rotuladas como *intra-thread*, consecutivas no fluxo de execução. Estes traços são delimitados por instruções do tipo *não redundante* ou *inter-thread*. A Figura 3 exibe cinco combinações onde traços redundantes podem ser ou não formados. Nos itens *(a)*, *(b)* e *(c)*, ocorrem a criação de traços redundantes delimitados por instruções *não redundantes* e *inter-thread*. No caso dos itens *(d)* e *(e)*, apesar do surgimento de sequências de instruções redundantes válidas, nenhum traço pode ser formado, pois não há sequências de duas ou mais instruções *intra-thread*.

Instruções *inter-thread* são excluídas da formação dos traços. Desta forma, é possível garantir que toda a sequência de instruções em um determinado traço provém da mesma *thread*. Com isso, a consistência do estado do contexto da *thread* que utilizará um determinado traço é preservada.

Traços formados com instruções *inter-thread* são considerados falsos e não são permitidos no modelo DTM@GPU. A Figura 3, item (d), exemplifica o surgimento de um traço falso, composto pelas instruções 2, 3, 4 e 5. Caso tal sequência ocorra, cada instrução redundante é reutilizada separadamente.

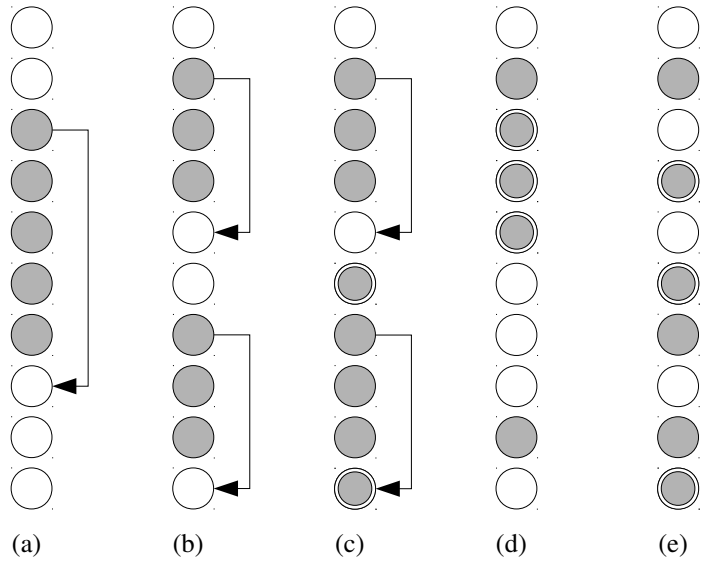

Figura 3. Exemplo de situações onde traços podem ou não ser formados. (a) Traço de tamanho cinco, (b) dois traços de tamanho três, (c) dois traços de tamanho três, o primeiro terminado em instrução *não redundante* e o segundo terminado em instrução *redundante inter-thread*, (d) e (e) nenhum traço formado.

4. Análise Experimental

A técnica DTM@GPU foi implementada no simulador GPGPU-sim e avaliada com um conjunto de 9 aplicações reais. Durante cada experimento, as seguintes métricas foram analisadas: percentual de reúso obtido, estimativa de aceleração com o reúso de instruções e a quantidade média de traços reusados.

4.1. Simulador GPGPU-sim

O GPGPU-sim é um simulador de GPU capaz de executar o conjunto de instruções de *threads* paralelas (PTX) da NVIDIA. O modelo de simulação usado contempla a arquitetura Fermi da NVIDIA. Embora a Fermi seja uma arquitetura mais antiga da NVIDIA, as inovações propostas nas arquiteturas seguintes como a Kepler, Maxwell e recentemente a Pascal, não afetam a avaliação sobre o potencial de reúso das aplicações. A arquitetura do simulador consiste de uma coleção de *cores* paralelos, chamados de *shader cores*, que estão conectados a múltiplos módulos de memória através de uma rede de interconexão. Cada *shader core* é um processador SIMD de largura 8 e equivale a um *Streaming Multiprocessor* (SM) da NVIDIA. O *shader core* possui um *pipeline* de 24 estágios com execução em ordem sem adiantamento de instruções. O *pipeline* é dividido em 6 estágios lógicos: despacho, decodificação, execução, memória1, memória2 e *writeback*.

As *threads* são escalonadas no *pipeline* em um grupo fixo de 32 *threads* (*warp*) e distribuídas nos *shader cores* proporcionalmente com um CTA inteiro. Os recursos são liberados somente quando todas as *threads* de um CTA finalizarem seu processamento.

Como a largura de cada *shader core* é de 8 unidades funcionais, são necessários 4 ciclos para que uma instrução seja executada nas 32 *threads* do *warp*. A cada 4 ciclos, *warps* prontos para execução entram no *pipeline*. Um *warp* está pronto se todas suas *threads* também estiverem.

O simulador GPGPU-sim foi modificado para coletar informações sobre a execução das aplicações CUDA e avaliar o grau de instruções redundantes presente nelas. A política utilizada para substituição de linhas na MTG e MTT foi a LRU.

4.2. Métricas

A Tabela 3 mostra as medidas realizadas. A medição de reúso total do sistema é feita através da utilização das tabelas MTG e MTT e ocorre através de duas situações distintas:

1. Contagem dos acertos na MTG, o que caracteriza reúso simples de uma única instrução por vez;
2. Contagem das instruções pertencentes aos traços reusados. Neste caso, várias instruções foram reusadas de uma única vez.

Tabela 3. Medidas realizadas.

Medidas de Reuso		Medidas de Aceleração	
$intra$	Quantidade de instruções reusadas dentro da *thread*	$w0$	Quantidades de ciclos que o *warp* deixou de executar por dependência estrutural
$inter$	Quantidade de instruções reusadas entre *threads*	$w0_{dtm}$	Quantidades de ciclos que o *warp* deixou de executar por causa do DTM
$trace$	Quantidade de instruções reaproveitadas por reúso	$ciclos_{sem}$	Quantidade de ciclos executados sem utilização do DTM
$itot$	Quantidade total de instruções executadas	$ciclos_{dtm}$	Quantidade de ciclos executados com DTM, $ciclos_{dtm} = ciclos_{sem} - w0_{dtm}$

De acordo com os valores coletados, o percentual de reuso é dado por: $reuso = (intra + inter + trace)/itot$ e a aceleração é dada por: $speedup = ciclos_{sem}/ciclos_{dtm}$.

4.3. Aplicações

As aplicações utilizadas em nossos experimentos são as mesmas propostas pelos autores do simulador GPGPU-sim. Com isso, os resultados obtidos durante esta fase podem ser comparados com os resultados obtidos na versão original do simulador. Possibilitando também, a comparação de trabalhos futuros na mesma plataforma. As aplicações utilizadas nos experimentos sofreram intensa otimização por parte dos autores. Desta forma, os ganhos apresentados representam acréscimos a códigos que possuem pouca margem para aperfeiçoamento.

1. **AES Cryptography (AES)**: Implementação do algoritmo de criptografia *Advanced Encryption Standard* em CUDA. Otimizada para fazer uso da memória de constantes e textura. No experimento um arquivo de 256KB foi criptografado com chave de 128 bits. O AES possui 65792 *threads* e 28M instruções.
2. **Breadth First Search (BFS)**: Busca em largura em um grafo onde cada nó é mapeado em uma *thread*. Desta forma, o paralelismo escala com o tamanho da entrada. Neste experimento, a busca em largura foi realizada em um grafo aleatório com 65536 nós. O BFS possui 65536 *threads* e 17M instruções.

3. **Coulombic Potential (CP)**: Simulação física integrante do "Parboil Benchmark suit". Também faz uso da memória constante em suas otimizações. Foram simulados 200 átomos em uma *grid* de tamanho 256×256. O CP possui 32768 *threads* e 126M instruções.
4. **3D Laplace Solver (LPS)**: Aplicação financeira paralela otimizada para uso de memória compartilhada e acessos à memória agrupados (Coalesced). O LPS possui 12800 *threads* e 82M instruções.
5. **MUMmerGPU (MUM)**: Ferramenta de sequenciamento de DNA. A aplicação foi otimizada para utilizar memória de textura. Nos experimentos, foram utilizados os primeiros 140000 caracteres do genoma Bacilo anthracis str. Ames. O MUM possui 50000 *threads* e 77M instruções.
6. **Neural Network (NN)**: Rede neural para reconhecimento de escrita. Durante os experimentos foram reconhecidos 28 dígitos do "Modified National Institute of Standards Technology database of handwrite digits". O NN possui 35000 *threads* e 68M instruções.
7. **N-Queens Solver (NQU)**: Problema das N rainhas no Xadrez. Consiste em tentar colocar N rainhas em um tabuleiro $N \times N$ sem que nenhuma ataque a outra. Durante os experimentos, um tabuleiro 10×10 foi utilizado. O NQU possui 21408 *threads* e 2M instruções.
8. **Ray Tracing (RAY)**: Renderização de imagem foto-realística. Foram executados 5 níveis de reflexão com sombra para renderizar uma imagem de 256×256. O RAY possui 65536 *threads* e 71M instruções.
9. **StoreGPU (STO)**: Implementação do algoritmo MD5 com um arquivo de entrada de 192KB. Aplicação também otimizada para fazer uso da memória compartilhada e evitar tráfego fora da GPU. O STO possui 49152 *threads* e 134M instruções.

4.4. Percentual de Reúso

A Figura 4 exibe os percentuais de reúso obtidos para cada aplicação do conjunto de experimentos com diferentes tamanhos de entradas das tabelas. A média harmônica cresceu de 13,6% para 35,3% utilizando tabelas com tamanho variando de 16 até 8192 entradas.

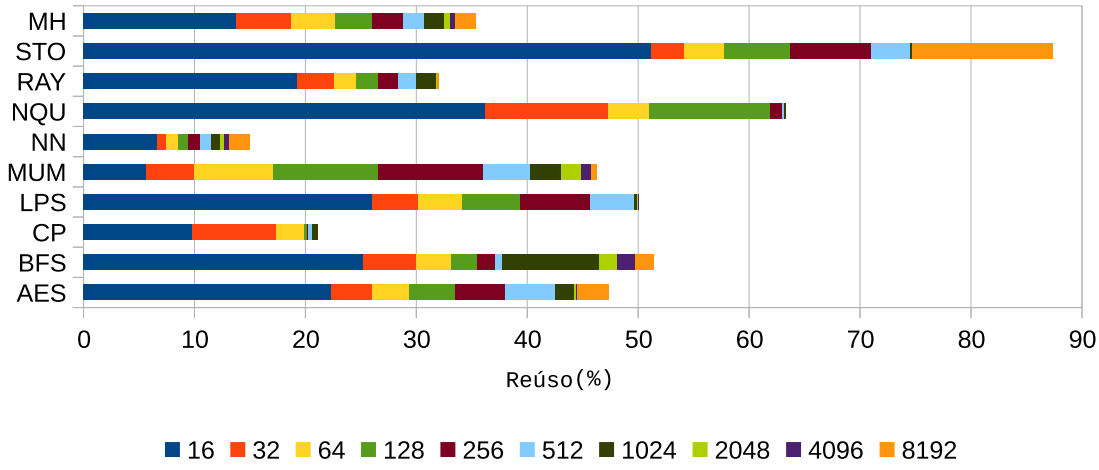

Figura 4. Variação do percentual de reúso no DTM@GPU para diferentes tamanhos de entradas das tabelas.

Nas tabelas com 16 e 32 entradas, a variação de reúso chega a 5% no pior caso, observado na aplicação MUM. Conforme o número de entradas nas tabelas aumenta, o nível de reúso cresce, chegando a alcançar o patamar de 30,7% com 512 entradas. Neste platô, a melhor diferença ocorre quando o tamanho das tabelas passa de 4096 para 8192, resultando em 1,83% a mais de reúso.

Considerando cada aplicação isoladamente, a que obteve um maior destaque foi a STO, chegando a 87,35% de instruções reusadas. As demais aplicações também apresentaram resultados interessantes, demonstrando patamares bem característicos no nível de instruções reusadas. Para as aplicações AES, CP, LPS, NN, NQU e RAY, foram alcançados os patamares de reúso de 44%, 21%, 50%, 12%, 63% e 31%, respectivamente.

O experimento demonstra que, para várias classes de aplicações, mesmo com um número reduzido de entradas nas tabelas de memorização, o modelo DTM@GPU é capaz de sustentar um bom nível de reúso de instruções. Também é possível notar que, para estes cenários, um aumento significativo no tamanho das tabelas trariam poucos benefícios.

4.5. Estimativa de Aceleração

A medida de aceleração representa uma estimativa, pois não considera os impactos que a suspensão da execução dos *warps* podem ocasionar no *pipeline* . Os dados extraídos durante a execução foram analisados e a provável aceleração de desempenho foi inferida a partir do padrão de reúso observado.

Como as instruções reusadas dispensam reexecução, as *threads* que contêm esse tipo de instrução podem ser desabilitadas. Ao conjunto de *threads* desabilitadas no *warp* soma-se o conjunto de *threads* desabilitadas pelo controle de fluxo. A quantidade de ciclos economizados corresponde ao total de ciclos em que *warps* deixaram de executar, por não possuírem *threads* que necessitem de execução naquele momento.

A Figura 5 apresenta os percentuais de aceleração obtidos variando-se as tabelas de memorização de 16 até 8192 entradas. A média harmônica da aceleração do desempenho variou entre 1,26% e 10,76%. Os melhores percentuais ocorrem na transição de 64 para 128 entradas, alcançando um incremento de 1,57% na aceleração.

A partir de 1024 entradas nas tabelas, o patamar de 10% de aceleração é alcançado. Aumentos adicionais nas entradas passam a não ser tão expressivos, causando pouco impacto no desempenho das aplicações.

Analisando as aplicações de forma independente, é possível notar a baixa aceleração por nível de reúso das aplicações BFS, MUM e NQU. Apesar de possuírem percentual de reúso máximo de 50%, 45% e 60%, respectivamente, obtiveram aceleração de apenas 4%, 5% e 13%. Já as aplicações CP, LPS, NN, RAY e STO acompanharam os níveis de reúso máximo, chegando a uma estimativa de aceleração próxima a 70% (LPS).

4.6. Distribuição de reúso por tipo

O último experimento realizado foi a análise da contribuição de cada tipo de reúso por aplicação. Estes tipos são compostos por instruções *intra-thread*, *inter-thread* e instruções que formam traços. A Figura 6 detalha os percentuais obtidos para cada aplicação, onde destacam-se a STO e a AES com 98% e 100% de reúso *inter-thread*, respectivamente.

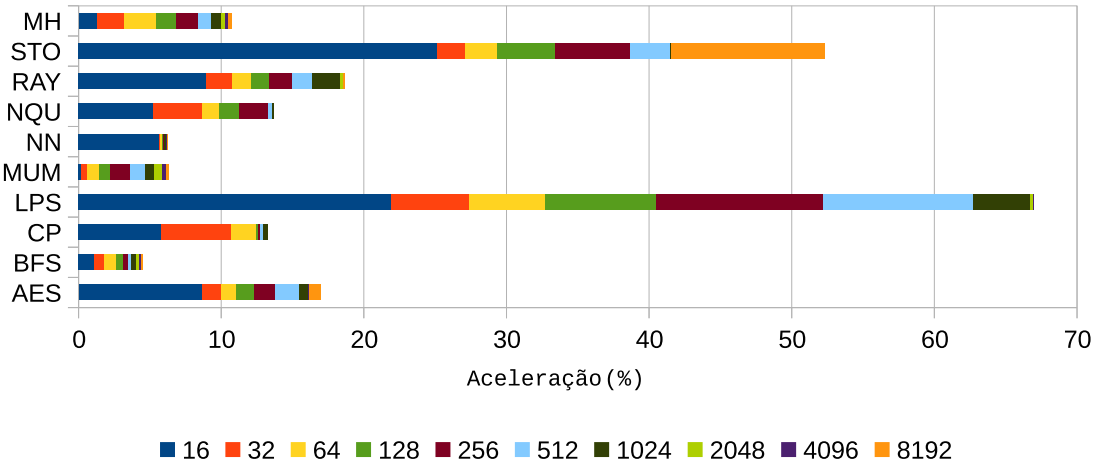

Figura 5. Variação da estimativa de aceleração de desempenho no DTM@GPU para diferentes tamanhos de entradas das tabelas.

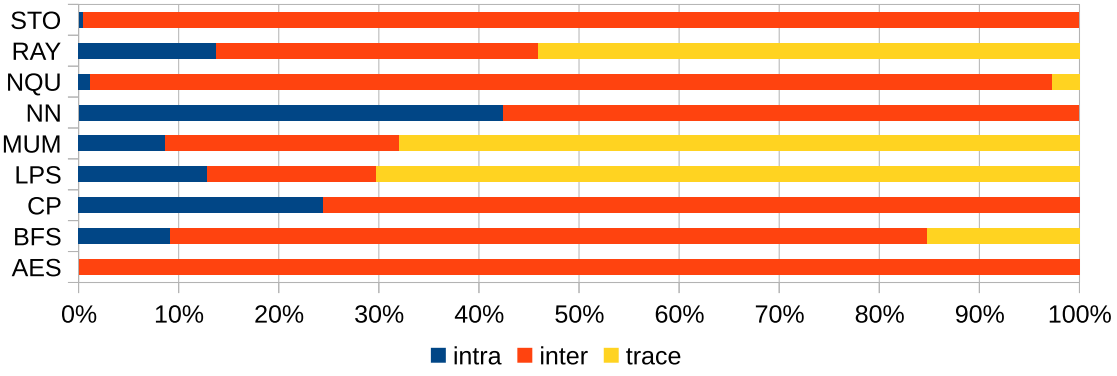

Figura 6. Contribuição percentual de cada tipo de reúso para tabelas com 8192 linhas.

5. Conclusões e Trabalhos Futuros

Este trabalho apresentou o uso da técnica DTM para explorar o reúso de instruções em GPUs. Foram indentificados três tipos de reúsos em ambientes *multithread* como a GPU: reúso *intra-thread*, reúso *inter-thread* e reúso de traços redundantes. Também foi proposta a distribuição de pares de tabelas de memorização MTG e MTT para cada *core*, de forma que os *warps* possam aproveitar o reúso em suas *threads*, e eliminar os campos de entradas das tabelas de memorização correspondentes aos mecanismos de previsão de desvio, já que GPUs não fazem uso desses mecanismos.

Resultados experimentais com 9 aplicações reais mostraram que o reúso de instruções em GPU consegue melhorar o desempenho das aplicações mesmo quando intensamente otimizadas. Os resultados mostraram que $35,3\%$ (média harmônica) das instruções são reutilizadas em relação ao total de instruções executadas. O reúso *intra-thread* foi aplicado em $10,4\%$ das instruções redundantes, sendo que esse resultado não foi maior, porque a redundância de traços exerce maior prioridade de reúso. O reúso de traços foi aplicado em $23,1\%$ das instruções redundantes e o reúso *inter-thread* obteve a maior fatia das instruções redundantes, cerca de $66,4\%$ das instruções. As estimativas de

aceleração alcançaram a média harmônica de $9,99\%$, $10,32\%$, $10,53\%$ e $10,76\%$ para as tabelas com 1024, 2048, 4096 e 8192 entradas, respectivamente.

Como trabalhos futuros, pretende-se avaliar o impacto energético com o desligamento físico das faixas de execução não utilizadas. Pretende-se também adotar um método de formação dinâmica de *warps* que reagrupa *threads* para maximizar sua ocupação. Por fim, pretende-se a introdução de um mecanismo para o programador configurar o tamanho das tabelas de memorização no processador.

Agradecimentos

À FAPERJ (processo E-26/203.537/2015), ao CNPq e à CAPES pelo apoio dado aos autores deste trabalho.

Referências

Bakhoda, A., Yuan, G., Fung, W., Wong, H., and Aamodt, T. (2009). Analyzing cuda workloads using a detailed gpu simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*, pages 163–174.

da Costa, A., Franca, F., and Filho, E. (2000). The dynamic trace memoization reuse technique. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 92–99.

Lipasti, M. H., Wilkerson, C. B., and Shen, J. P. (1996). Value locality and load value prediction. *ACM SIGPLAN Notices*, 31(9):138–147.

Pilla, M., Childers, B., da Costa, A., Franca, F., and Navaux, P. (2006). A speculative trace reuse architecture with reduced hardware requirements. In *18th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 47–54.

Pilla, M., Navaux, P., Childers, B., da Costa, A., and Franca, F. (2004). Value predictors for reuse through speculation on traces. In *16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 48–55.

Pilla, M., Navaux, P., da Costa, A., Franca, F., Childers, B., and Soffa, M. (2003). The limits of speculative trace reuse on deeply pipelined processors. In *15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 36–44.

Sazeides, Y. and Smith, J. E. (1997). The predictability of data values. In *Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–258.

Silva, B., Abreu, E., and Franca, F. (2005). JDTM memorização e reuso dinâmico de traços em uma arquitetura de processador java. In *VI Workshop em Sistemas Computacionais de Alto Desempenho, 2005*, pages 57–64.

Sodani, A. and Sohi, G. S. (1998). An empirical analysis of instruction repetition. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 35–45.