O POTENCIAL DE REUSO DE TRAÇOS EM GPUS

Saulo Tavares Oliveira

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Felipe Maia Galvão França

Rio de Janeiro
Setembro de 2014

O POTENCIAL DE REUSO DE TRAÇOS EM GPUS

Saulo Tavares Oliveira

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Leandro Augusto Justen Marzulo, D.Sc.

Prof. Maurício Lima Pilla, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2014

Oliveira, Saulo Tavares

O Potencial de Reuso de Traços em GPUs/Saulo Tavares Oliveira. – Rio de Janeiro: UFRJ/COPPE, 2014.

XIII, 103 p.: il.; 29, 7cm.

Orientador: Felipe Maia Galvão França

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2014.

Referências Bibliográficas: p. 89 – 93.

1. Reuso dinâmico de traços. 2. GPU. 3. Computação Heterogênea. 4. CUDA. 5. OpenCL. 6. Multi-thread. 7. Paralelismo. 8. Arquitetura de Processador. I. França, Felipe Maia Galvão. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Agradecimentos

À minha família, por ter acreditado em mim e dado todo apoio que eu precisava. Meus pais, por terem me ensinado a importância de estudar, sempre me incentivando e dando os exemplos para que eu pudesse me desenvolver. Meu irmão e minha companheira, por terem tido paciência comigo e compreendido minhas faltas e defeitos. Eles são únicos e insubstituíveis em minha vida.

Ao professor Felipe França, por ter me aceitado como seu orientando, me dado liberdade de ideias e identificado em mim o potencial para desenvolver este trabalho. O jeito cativante sempre fez sua presença ser um momento de alegria, característica fundamental para manter unidos todos aqueles sob sua orientação. Obrigado por ter me posto em contato com as pessoas certas para que eu pudesse avançar com a pesquisa. Ao professor Amarildo da Costa, pela atenção dada em responder minhas perguntas, suas respostas muito me ajudaram a entender o DTM.

Ao pessoal do LAM, Alexandre Nery, Leandro Marzulo, Tiago Alves, Rafael Lima, Fabrício Barros, Luneque Silva, Alexandre Sardinha e Lúcio Paiva, obrigado por todas as dúvidas tiradas. Especialmente Sardinha e Lúcio, eles influenciaram minha vinda para o LAM enquanto eu ainda era candidato à aluno na COPPE. Ao pessoal no LabIA Douglas Cardoso, João Abrantes, Paulo Felipe, Daniel Alves, Daniel Nunes, Danilo Carvalho, Fábio Jimenez, Hugo Cesar, Kleber Aguiar, Diego Souza e Eduardo Ribeiro, agradeço a companhia e aos bons momentos colonizando ilhas, derrotando orcs, limpando zumbis e memoráveis partidas de xadrez. Aos colegas da Fundação COPPETEC Pieter Veldman, Luís Fernando, Francinei Gomes e Pedro Rougemont, por também terem me incentivado a entrar no mestrado. Aos meus contemporâneo na COPPE Nelson Perez, Bruno Steckelberg, Fellipe Duarte, Marcos Vieira e Vanius Farias pelas inúmeras trocas de boas ideias durante o curso.

Ao Programa de Engenharia de Sistemas e Computação (PESC/COPPE/UFRJ) e seus funcionários e professores pela estrutura, física e intelectual, necessária para o desenvolvimento deste trabalho. Aos professores do DCC/UFRJ por terem me ensinado as bases da computação, especialmente aos professores Gabriel Silva e Austeclynio Pereira por terem sido meus grandes orientadores durante a graduação. Aos pagantes de impostos por terem financiado as instituições de ensino que tenho sido aluno desde adolescente.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

O POTENCIAL DE REUSO DE TRAÇOS EM GPUS

Saulo Tavares Oliveira

Setembro/2014

Orientador: Felipe Maia Galvão França

Programa: Engenharia de Sistemas e Computação

O poder de processamento dos microprocessadores cresce rapidamente seguindo a lei de Moore, porém devido limitações físicas migrou-se para processadores paralelos. Neste cenário as GPUs mostraram-se capazes de alta vazão no processamento de dados com diversas aplicações não gráficas portadas para esta arquitetura. Mesmo assim algumas aplicações permanecem não se beneficiando das grandes acelerações no desempenho normalmente associadas a ela.

Propomos a inédita aplicação do DTM (*Dynamic Trace Memorization*) utilizando-se a GPU (*Graphics Processing Unit*) como arquitetura substrato com o objetivo de medir o potencial de reuso e ganhos de desempenho associados. Em trabalhos anteriores esta técnica já foi aplicada com êxito em processadores superescalares e em um processador Java. O reuso de traços identifica redundância e reaproveita a computação de sequências de instruções já executadas.

Este trabalho contribui principalmente (i) identificando as alterações necessárias para o DTM funcionar em ambientes massivamente paralelos e (ii) definindo um método para compartilhamento de redundância oriundas de diversas *threads*. Em nossos experimentos medimos reusos de até 35,3% na média harmônica, ele se dividiu em 66,4% proveniente do compartilhamento *inter-thread*, 23,1% proveu do reuso de traços redundantes e o restante originou do reuso *intra-thread*. Estimamos uma aceleração de desempenho da ordem de 10,7%. Com isto, mostramos como um conjunto diverso de aplicações se beneficia deste aprimoramento conceitual através da exposição do grau de redundância presente.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

THE POTENTIAL OF TRACE REUSE ON GPUS

Saulo Tavares Oliveira

September/2014

Advisor: Felipe Maia Galvão França

Department: Systems Engineering and Computer Science

The processing power of microprocessors grows rapidly according with Moore's law, however because of physical limitations they migrated to parallel processors. In this scenario GPUs are capable of high throughput processing power with various non-graphical applications ported to this architecture. Nevertheless some applications remain not taking advantage of the rapid acceleration performance normally associated with it.

We propose the novel application of DTM using the GPU architecture as substrate in order to measure the potential reuse and performance gains associated. In previous work this technique has already been applied successfully in superscalar processors and in Java processor. The trace reuse identifies redundancy and use again the values of instructions traces already executed.

This research mainly contribute (i) identifying the necessary modifications on DTM in order to work on a massively parallel environments and (ii) defining a method for sharing redundancy from different threads. In our experiments we measured reuses up to 35.3% (harmonic mean), it was divided into 66.4% from the *inter-thread* share, 23.1% provided from trace reuse and the rest of the reuse was *intra-thread*. We estimate an speedup of 10.7%. With this, we show how a very different set of applications can benefit from this conceptual enhancement by exposing the degree of redundancy present in them.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este trabalho explora o conceito de Memorização e Reuso Dinâmico de Traços aplicado em GPUs (*Graphics Processing Unit*). Usando o simulador GPGPU-SIM [1] implementamos o mecanismo de *Dynamic Trace Memoization* (DTM) proposto por DaCosta[2] de forma a identificar e medir as sequências de instruções redundantes em aplicações portadas para GPU. Batizamos esta variante de DTM@GPU.

Nas últimas décadas o poder de processamento dos microprocessadores cresceu rapidamente seguindo a lei de Moore. Ela prediz que a capacidade de processamento dobra a cada 18 meses. Sabendo como limitações físicas impedem o aumento indiscriminado da frequência de operação, os arquitetos tomaram como alternativa o aumento do número de núcleos por integrado na tentativa de manter válido o ritmo de expansão. Somado a isto, processadores antes dedicados à gráficos, agora incluem mecanismos que permitem utilizá-los para propósito geral.

Estas características colocam-nos na era dos computadores massivamente paralelos. Neste patamar tecnológico um único processador pode possuir centenas de núcleos idênticos replicados. Problemas de escalabilidade que antes eram desprezíveis, devido aos poucos núcleos, agora passam a ter impactos relevantes. Entre os problemas, a dificuldade de manter os núcleos alimentados com um fluxo constante de dados devido contenção no barramento de comunicação.

Outras propostas de processadores tentaram superar as limitações das arquiteturas dominantes. Mudanças agressivas no paradigma de programação já foram tentadas antes, porém, elas esbarraram em diversos problemas incluindo os custos de desenvolvimento de *software*. Como exemplo recente de abandono de arquitetura temos o processador CELL da IBM [3] [4]. Muito a frente de seu tempo e com grande poder de processamento bruto ele logo tomou posição de destaque na época de seu lançamento. Entretanto atingir o pico de desempenho teórico deste processador mostrou-se uma tarefa de difícil realização devido a dificuldade de se programa-lo. Atentos a demanda por *teraflops*, uma outra empresa antes invisível no cenário de computação de alto desempenho introduziu um *hardware* multi-propósito capaz de

grande volume de processamento.

Em 2007 a NVIDIA apresentou a CUDA, uma plataforma de programação paralela inventada para aproveitar o poder de processamento disponível nas placas de vídeo. Ela expõe, através de uma API (*Application Programming Interface*) própria, o *hardware* que antes era utilizado apenas para processamento gráfico. Rapidamente a comunidade científica adotou este paradigma, no processo surgiu a plataforma de código aberto e independente de fabricante: a OpenCL [5].

Processadores de vídeo possuem uma grande base de ocupação de mercado, parte de seus investimentos provem da indústria do entretenimento como jogos e cinematográfica. A popularidade destes processadores, somados ao baixo custo por *gigaflop* tornaram aceleradores baseados em GPU atraentes em diversos campos científicos como simulação de partículas, processamento de imagens, criptografia, entre outros.

O crescente número de projetos que usufruíram de grande aumento de desempenho quando portados para GPU levou ao aumento da popularidade desta forma de computação. Diversos trabalhos foram publicados apresentando propostas para aperfeiçoar esta arquitetura. Cada um baseou-se em algum dos simuladores de GPUs disponíveis, como por exemplo, o GPGPU-sim, Ocelot e o Barra.

Neste trabalho utilizamos o GPGPU-sim, um simulador de computador paralelo desenvolvido na University of British Columbia pela equipe de Tor M. Aamodt. Eles alteraram seu código para torná-lo compatível com o código PTX (Parallel Thread Execution). A migração de arquitetura deste simulador viabilizou diversas investigações necessárias para o aprimoramento da GPU como co-processador [6] [7] [8].

A técnica de reuso de traços descreve um aprimoramento conceitual de um processador. Pela utilização de tabelas de memorização, armazenam-se os resultados das instruções executadas para posteriormente reutiliza-los na próxima ocorrência delas, evitando assim sua reexecução. Esta técnica baseia-se na observação de que a quantidade de instruções redundantes de um programa representa um percentual elevado do total de instruções executadas.

Outras pesquisas já exploraram redundância de processamento em diversos níveis, por exemplo, em nível de função, trecho estático, traços, blocos básicos ou mesmo instruções. DaCosta [2] mostrou-nos que o reaproveitamento em nível de traços supera outras formas de reuso por conseguir, dinamicamente, identificar mais oportunidades de reuso. Tendo como base o trabalho dele, Pilla [9], [10], [11] propõe aperfeiçoamento do mecanismo adicionando capacidade de execução especulativa. Ortogonalmente também já foi estudada a aplicação do DTM tendo como alvo a arquitetura da máquina virtual Java [12].

O reuso de traço reaproveita uma sequência de instruções redundantes dispensando a reexecução delas. Este depende das instruções terem sido rotuladas ante-

riormente, de acordo com o resultado da consulta a uma tabela de memorização. Durante a construção do traço redundante informações sobre as instruções são colhidas e posteriormente utilizadas para identificar oportunidades de reuso e atualizar o estado do processador.

Este trabalho propõe medir o potencial de reuso de traços aplicado à arquitetura *multithread* da GPU. Ele contribui com uma implementação capaz de (i) identificar construtivamente, memorizar e reusar traços, (ii) identificar e mitigar a armadilha associada ao DTM quando aplicado a sistema *multithread* e (iii) compartilhar as tabelas de memorização entre diversos fluxo de execução.

1.1 Motivação

A introdução de novas técnicas em nível de arquitetura visando aprimoramento exige minuciosa pesquisa, afim de, antever se as modificações projetadas trarão benéficos fortes o suficientes que garantam a viabilidade da implementação física da ideia proposta. Para isto, exaustivas simulações e análises dos dados gerados suportam as escolhas realizadas, diminuindo os riscos de falha no projeto.

Neste trabalho estudamos duas tecnologias já testadas e estabelecidas e propomos avaliar se a aplicação do DTM possuindo a GPU como arquitetura substrato poderia proporcionar ganhos de *performance*. Como parte do estudos de viabilidade inerentes ao complexo desenvolvimento de um processador, fomos motivados pelas seguintes questões:

- Nas cargas de trabalho características de uma GPU, em particular nos programas não gráficos executados em GPU, existe nível de redundância suficiente que justifique a aplicação do DTM neste tipo de processadores?
- Dada a natureza SIMD(*Single Instruction*, *Multiple Data*) das GPUs, a presença de redundância consegue alcançar as faixas de execução do processador de forma a este vetor de processamento poder ser desabilitado devido múltiplas redundâncias (economizando em processamento)?

Para isso, através do emprego de um simulador, adaptamos o DTM ao modelo arquitetural da GPU de modo a conseguir avaliar o grau de redundância deste tipo de sistemas. Como programa de teste, utilizamos aplicações portadas que enfrentam dificuldades de *performance* nas GPUs atuais. Tais aplicações podem se beneficiar de avanços no *hardware* como forma de superar os gargalos presentes.

1.2 Objetivo

Como objetivo desta pesquisa queremos medir o nível de reuso presente nas cargas de processamento tipicamente submetidas a execução em GPU. Através do reaproveitamento de instrução em nível de traço, queremos provar o potencial de ganho de desempenho neste sistema. Para tal, propusemos as alterações necessárias no DTM de modo a funcionar nesta distinta arquitetura.

Durante o estudo de viabilidade do reuso em nível de traço em GPUs, procedemos as seguintes investigações:

1. Determinar os parâmetros arquiteturais adequados para o DTM@GPU tendo em vista se tratar de um processador com paradigma de programação e conjunto de instruções bem distintos da CPU;

2. Verificar os níveis de redundância na execução das instruções do *kernel* instanciado por uma *thread*;

3. Verificar os níveis de redundância entre diferentes *threads* instanciadas do mesmos *kernel*;

4. Determinar como o DTM pode afetar a ocupação das faixas de execução nos vetores de processamentos tendo em mente a natureza SIMD destes processadores;

5. Determinar se o reuso ocorrerá em nível suficiente a ponto de atingir todas as faixas de execução, levando a ocupação ao nível zero e por sua vez dispensando a execução da instrução. Exclusivamente a ocorrência deste evento caracterizaria a suspensão da execução desta instrução. Em contraste, na CPU qualquer instrução reusada não precisa ser reexecutada;

6. Determinar se os traços formados podem ser reusado em fluxos de execução diferentes daqueles onde foram construídos;

7. Determinar se podemos usar o DTM para diminuir a ocorrência da serialização por divergência de código, causadora de um dos maiores impactos sofridos pela computação em GPU.

Durante os experimentos preliminares, realizados logo no início da pesquisa, percebemos que o reuso de traços não funcionaria com a GPU sem a inserção de novos conceitos ao DTM. Isto ocorreu devido identificação de uma armadilha relacionada ao reuso de traços em ambientes *multithread*. Desta forma estendemos o objetivo do trabalho para incluir a correta caracterização e o detalhamento sobre como evitar esta armadilha.

Mostraremos neste trabalho como um aprimoramento conceitual tem potencial de resolver gargalos já identificados neste paradigma paralelo através da exposição do grau de redundância presente nas aplicações estudadas.

1.3 Estrutura do texto

Esta dissertação segue a seguinte organização: No Capítulo 2, explicamos o funcionamento do mecanismo de reuso DTM e apresentamos os detalhes de sua implementação. No Capítulo 3, falamos sobre o novo paradigma de programação que utiliza GPUs como coprocessadores de uso geral e apresentamos a arquitetura CUDA detalhando suas peculiaridades, neste capítulo apresentamos ainda o simulador GPGPU-sim utilizado como base neste trabalho. No Capítulo 4, tratamos da técnica DTM aplicada ao simulador, esclarecemos as alterações realizadas tanto no mecanismo DTM quanto na arquitetura da GPU. No Capítulo 5, apresentamos o método experimental e os resultados além de analisarmos os dados obtidos. No Capítulo 6, deixamos nossas conclusões e apontamentos para trabalhos futuros.

Capítulo 2

Reusos de Traços

O aprimoramento de um processador pode ocorrer tanto física quanto conceitualmente. O aprimoramento físico diz respeito a tecnologia de semi-condutores empregada, enquanto aprimoramento conceitual refere-se a arquitetura na qual seus componentes são organizados.

Pode-se alcançar o aprimoramento físico, por exemplo, pelo aumento da frequência de operação ou alteração na técnica de fabricação. Já no aprimoramento conceitual, pode-se buscar ter mais unidades funcionais ou aumentar a quantidade de cache ou mesmo propor uma organização dos componentes que evite deixar estruturas ociosas.

A técnica de reuso de traços trata de um aprimoramento conceitual. Ela utiliza uma tabela de memorização para guardar os resultados processados dispensando a reexecução de sequencias de instruções repetidas.

Este capítulo está organizado da seguinte forma: Começamos apresentando o reuso de computação e relatando trabalhos anteriores na área, na sequência detalhamos as etapas necessárias para identificação das instruções redundantes, construção dos traços redundantes e identificação das oportunidades de reuso dos traços redundantes. Por fim, trataremos da arquitetura substrato apontando as diferenças entre processadores super-escalares e GPU sob a ótica do DTM.

2.1 Visão Geral

Em um programa a quantidade de instruções redundantes representa uma fatia expressiva do total de instruções executadas.

Instruções redundantes são instâncias de instruções estáticas sendo executadas com os mesmos valores de operandos e por conta disto produzindo os mesmo resultado[2]. Quando elas aparecem em sequência no fluxo de instruções dinâmicas damos o nome de traços redundantes. Eles compreendem apenas instruções redundantes e seu reuso significa reaproveitamento de todas as instruções de uma vez.

Em referências anteriores, a técnica DTM (*Dynamic Trace Memoization*) obteve ganhos de desempenho devido reduções no número de instruções executadas e diminuição das penalidades por desvios preditos incorretamente. A construção de um traço requer a identificação de redundância com granularidade em nível de instrução dinâmica.

Listamos os principais conceitos envolvidos na técnica de reuso de traço. Eles compreendem os seguintes itens:

Instrução dinâmica: São instâncias dinâmicas das instruções estáticas do código executando. Tem associados a ela os valores dos operandos de entrada e saída;

Traço de execução: Sequência de instrução dinâmicas, redundantes ou não, originada da execução de um programa. O caminho seguido por um traço de execução correspondentes a um programa varia de acordo com o dados sendo processado. Uma pequeno trecho de código pode, por exemplo, gerar um longo traço de execução contendo muitas instruções dinâmicas devido a presença de laços.

Instruções válidas: Conjunto de instruções selecionadas como candidatas ao reuso. Por exemplo, instruções aritméticas e lógicas.

Instruções inválidas: Conjunto de instruções não pertencentes às candidatas ao reuso. Por exemplo, instruções de memória, por possuírem efeitos colaterais. Diversos fatores externos ao processador podem alterar o valor da posição de memória tornando custoso manter seu estado consistente;

Instrução redundante: São instruções dinâmicas e válidas executadas com os mesmos valores de operandos e em consequência produzindo o mesmo resultado. Em outras palavras, o resultado de uma operação redundante dispensa reprocessamento, pois já é conhecido e encontra-se armazenado na tabela de memorização;

Traço redundante: Sequência de instruções redundantes identificadas nos traços de execução. Ele expande a ideia de reuso de instruções. Ao invés de reaproveitar uma única instrução isoladamente, ele reaproveita uma sequência de instruções atualizando todos os registradores de destino envolvidos;

Contexto de entrada: Conjunto de operandos e seus respectivos valores que foram alterados por instruções de fora do traço redundante. Para um traço redundante ser reutilizado ele deve corresponder aos mesmos valores do estado do processador;

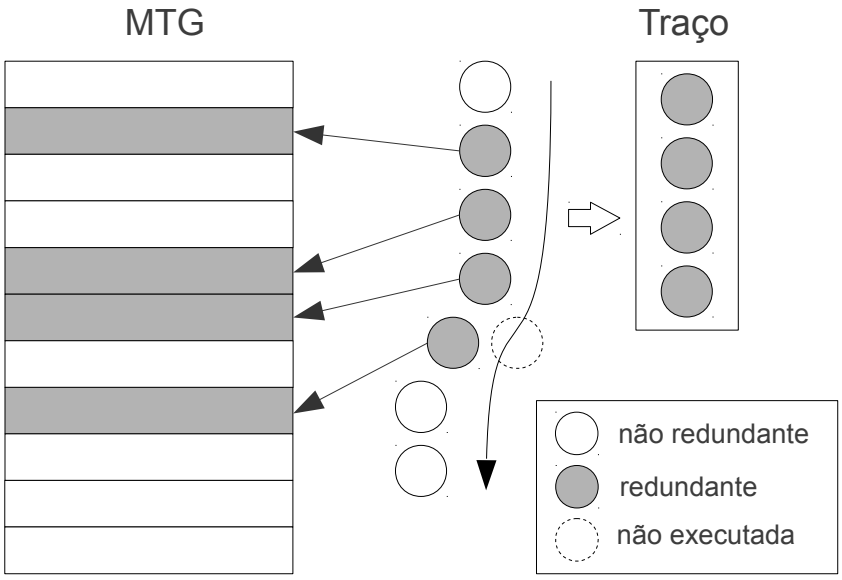


Figura 2.1.1: Formação de traço redundante a partir de sequência de instruções dinâmica redundantes.

> **Contexto de saída**: Conjunto de operadores e seus respectivos valores que foram alterados por instruções de dentro do traço redundante. Os valores contidos nele correspondem ao resultado da computação realizada pelas instruções que compõe o traço redundante.

Um traço redundante é delimitado por instruções não redundantes e pode incluir instruções de desvio, como exemplificado na Figura 2.1.1. Nela pode-se observar como a MTG (Memo Table G) é utilizada para armazenar as instruções já executadas e rotular as instruções como redundantes ou não. Observa-se ainda como a sequência de instruções redundantes formam um único traço redundante capaz de reaproveitar todas as instruções contidas nele de uma única vez.

Trabalhos Anteriores

Antes do DTM outros trabalhos propuseram reuso de computação. Eles podem ser classificados como estático ou dinâmico quanto ao modo de identificação da computação redundante; e classificados como Software ou Hardware quanto ao método de memorização da computação e quanto a forma de identificação de reuso.

Como exemplo podemos citar o *Value Cache*, *Instruction Reuse*, *BlockHistory Buffer*, *ALU Lookup Table*, *Trace Level Reuse* e Reuso assistido pelo compilador.

Outros trabalhos já foram realizados tendo como base o DTM. As propostas englobavam tanto aperfeiçoamento do mecanismo quanto aplicação da técnica em

outra arquitetura alvo.

Pilla [9], [10], [11] observou que através de predição de valores diversos traços contidos na tabela de memorização poderiam ser reusados mesmo quando os valores de entrada não estivessem completamente disponíveis. Este método especulativo recebeu o nome de *Reuse through Speculation on Traces* (RST). Na sequência de trabalhos ele levou o mecanismo ao seu limite obtendo ganhos de desempenho até 57% superior ao DTM não especulativo.

Já a proposta apresentada por Silva [12] de implementar o mecanismo DTM em outro processador alvo obteve aceleração de 11%. Segundo o autor, aplicando o DTM em um processador Java obteve-se ganho de *performance* devido a redução do número de instruções executadas, a redução dos caminhos críticos determinados pelas dependências verdadeiras e pela redução das penalidades associadas às instruções de desvios.

2.2 Identificação de instruções redundantes

Chamamos de instrução redundante quando uma instrução válida executa mais de uma vez com os mesmos parâmetros de entrada. Podemos antecipar o resultado desta operação, evitando reexecuções desnecessárias, deste que os valores tenham sido armazenado em execução anterior.

Através da utilização da MTG(Memo Table G) podemos identificar e armazenar dinamicamente suas ocorrências e resultados. O uso desta tabela de memorização entrega o valor anteriormente computado dispensando uma nova execução da instrução.

Como veremos adiante, o processo de construção dos traços redundantes exige primeiro a correta detecção da ocorrência de instruções redundantes. A Tabela 2.2.1 lista as ações necessárias de acordo com a situação. Para cada instrução decodificada consulta-se a MTG em busca de uma entrada correspondente. De acordo com o resultado da busca a instrução receberá um rótulo apropriado.

A busca na tabela ocorre pela comparação dos valores do pc, sv1, sv2 da instrução com as entradas presente na MTG. Em caso de acerto nesta tabela rotulamos a instrução como *redundante*. Quando não encontramos uma entrada nesta tabela rotulamos a instrução como *não redundante* e adicionamos ela na tabela.

Cada item na MTG guarda informações a respeito da instrução dinâmica executada. Esta tabela funciona como uma cache armazenando dos dados das instruções executadas. Ao todo armazena-se cinco campos como pode se visto na Figura 2.2.1.

A seguir listamos os campos que compõe a MTG e suas respectivas descrições:

- **pc**: Armazena o endereço da instrução em execução, ele é suficiente para

Instrução válida	Memo Table G	Operandos iguais	Ação
falso	não avaliado	não avaliado	Marcar instrução como não redundante
verdadeiro	falso	não avaliado	Marcar instrução como não redundante e inserir na Memo table G
verdadeiro	verdadeiro	falso	Marcar instrução como não redundante e inserir na Memo Table G
verdadeiro	verdadeiro	verdadeiro	Marcar instrução como redundante

Tabela 2.2.1: Lista de ações para classificar as instruções de acordo com resultado da busca na MTG. Instruções inválidas e instruções que não estejam na MTG com seus operandos recebem o rótulo 'não redundante'; e instruções presentes na MTG com todos os seus operandos são rotuladas como 'redundante'.

Figura 2.2.1: Uma entrada da Memo Table G

identificar qual instrução esta sendo executada dispensando o conhecimento de seu *opcode*;

- **sv1**: Armazena o valor do primeiro operando;
- **sv2**: Armazena o valor do segundo operando;
- **res**: Armazena o valor do resultado da operação;
- **jmp**: bit indicador de desvio incondicional;
- **brc**: bit indicador de desvio condicional;
- **btaken**: bit indicador de desvio tomado.

2.2.1 Reusando instruções redundantes

Podemos reutilizar instruções redundantes apenas com as informações da MTG. Para isso, durante a fase de decodificação, é feita uma busca nesta tabela pelos campos pc, sv1 e sv2. Em caso de acerto, o resultado da operação é recuperado pulando-se o estágio de execução e indo direto para escrita do resultado no registrador de destino.

O reaproveitamento das instruções já apresenta ganhos pela não reexecução de instruções repetidas. Em [2] é mostrado que muitos destes valores alimentam instruções subsequentes que também são redundantes.

pc	npc	icr(N)	icv(N)	ocr(N)	ocv(N)	bmask(N)	btaken(N)

Figura 2.3.1: Uma entrada da MTT

Em sua pesquisa ele identificou sequencias de instruções redundantes que apresentam dependência verdadeira entre si. Ou seja, várias cadeias se formam onde a saída de uma instrução é utilizada na entrada de outra logo a frente formando um traço redundante. Sabendo disso, ele propõe um mecanismo de reaproveitar todo um traço composto por instruções redundantes a partir da detecção da redundância da primeira instrução do traço e seu contexto de entrada associado.

2.3 Construção de traços redundantes

Construímos traços redundantes na medida que identificamos instruções redundantes. Durante o processo registramos o contexto de entrada para posteriormente podermos identificar oportunidades de reuso e o contexto de saída que servirá para atualizar o estado do processador.

Dependendo da implementação campos adicionais podem ser empregados afim de manter o estado da máquina consistente na ocasião do reuso. Por exemplo, na implementação original do DTM apesentada por DaCosta [2] guarda-se também informações sobre saltos e tomadas de desvio afim de atualizar adequadamente o mecanismo de predição de desvio presente na arquitetura que serviu de base para seu trabalho.

Algumas instruções são consideradas inválidas no mecanismo de reuso de traços. Por exemplo, as instruções de acesso a memória (load e store) e as instruções de entrada e saída que lidam com dispositivos externos ao processador. Destacamos que instruções de saltos são válidas ao contrário do que ocorre com o mecanismo de reuso de blocos básicos.

No DTM os traços redundantes são armazenados em uma tabela de memorização própria denominada MTT(Memo Table T). Cada entrada desta tabela, exibida na Figura 2.3.1, representa uma traço redundante.

A seguir listamos os campos que compõe a MTT e suas respectivas descrições:

- **pc**: Armazena o endereço da primeira instrução do traço redundante;
- **npc**: Endereço da próxima instrução a ser executada em caso de reuso;
- **icr**: Indica os registradores contidos o contexto de entrada;
- **icv**: Armazena os valores dos registradores do contexto de entrada;

- **ocr**: Indica os registradores contidos o contexto de saída;
- **ocv**: Armazena os valores dos registradores do contexto de saída;
- **bmask**: Indica a presença de desvio do trace;
- **btaken**: Indica a presença de desvio tomados no trace;

A construção de um traço começa com a identificação de uma instrução redundante. Os dados colhidos durante esta fase são armazenados numa área temporária utilizada como *buffer*. Os contexto de entrada e saída do traço redundante em construção recebem dados de cada instrução redundante analisada durante o processo.

Utilizamos os seguintes procedimentos na formação dos contextos de entrada e saída:

1. O registrador fonte de uma instrução adicionada entra no contexto de entrada do traço redundante em formação apenas se este operando não estiver contido no contexto de saída presente;
2. O registrador destino de uma instrução adicionada entra no contexto de saída do traço em construção.

A restrição imposta no preenchimento do contexto de entrada indica que apenas registradores alterados fora do traço devem pertencer ao contexto, dispensando o armazenamento de valores gerados pelas instruções internas. Os campos *icr* e *icv* representam mapas de bits que indicam o registrador contido em cada contexto enquanto os campos *ocr* e *ocv* indicam os valores atribuidos.

As informações de desvio são atualizadas nos campos *bmask* e *btaken*, assim o preditor de desvios poderá ser atualizado quando o traço for reusados. O processo de construção termina com uma instrução não redundante. Quando isto ocorre, utiliza-se o *pc* da instrução não redundante para preencher o campo *npc* do traço em construção. Após isto, utilizamos o *buffer* temporário como entrada para MTT.

2.4 Identificação e reuso de traços redundantes

A técnica de Reuso de traço atualiza o processador para um estado futuro como se ele de fato tivesse executado as instruções presente nele. Quando o DTM identifica um traço redundante, ele reusa todas as instruções dele de uma única vez pulando para instrução apontada no campo *npc*.

Para reusar o traço, ele precisa identificar uma instrução redundante e verificar se o contexto de entrada coincide com o estado atual do processador. A cada instrução emitida realiza-se uma busca em ambas as tabelas de memorização. A Tabela 2.4.1

Memo Table G	Memo Table T	Ação
falso	falso	Nada para reusar
falso	verdadeiro	Reusar traço
verdadeiro	falso	Reusar instrução
verdadeiro	verdadeiro	Reusar traço

Tabela 2.4.1: Prioridade de reuso. Traços redundantes recebem prioridade de reuso frente ao reuso simples pois representam o reaproveitamento de várias instruções simultaneamente

resume as ações tomadas de acordo com o resultado das buscas tanto na MTG quanto na MTT.

Encontrando na MTT um registro correspondente ao *pc* ele verifica se o contexto de entrada confere com o estado atual do processador. Se os valores conferirem, então um traço redundante acaba de ser identificado e será reusado.

Na sequência os dados do traço redundante armazenado serão utilizados para atualizar o estado do processador. Para isso, altera-se os registradores do processador com os valores indicados no contexto de saída. O registrador *pc* recebe o valor indicado no campo *npc* do traço e atualiza-se os valores do preditor de desvio. Estas etapas colocam o processador no estado que ele estaria caso tivesse executado as instruções do traço reusado.

2.5 Arquitetura substrato

Em [2], [9], [10] e [11] aplica-se a técnica DTM numa arquitetura super-escalar. Ela representava na época o estado da arte de arquiteturas computacionais. Este tipo de processador possui muitas unidades funcionais e um pipeline extremamente profundo. Desta forma, um desvio de execução representava grande penalidade na *performance* dele.

Esta arquitetura utiliza um mecanismo de preditor de desvio para aliviar as penalidades de esvaziar um longo pipeline. O impacto na *performance* ocorre devido mudança do fluxo de execução do programa. Com isto em mente o desenvolvimento do DTM contemplou o comportamento deste componente, para tanto incluiu-se informações a respeito de desvios tanto na MTT quando na MTG, afim de que em caso de reuso os mecanismo preditores pudessem ser corretamente atualizados.

Nesta dissertação também trazemos o DTM aplicado ao atual estado da arte em arquiteturas de processadores. Introduzimos, pela primeira vez, a ideia de reuso em GPUs. Por possuírem peculiaridades em relação aos processadores super-escalares, adaptações serão feitas na técnica.

Agora as instruções de desvio continuam sendo as causadoras de problemas,

mas de uma forma diferente de antes. Nas GPUs o esvaziamento do pipeline não representa mais o ponto crítico. Comparado aos processadores super-escalares os núcleos da GPU são muito mais simples e possuem muito menos estágios.

Enquanto arquiteturas super-escalares executam instrução fora de ordem, possuem adiantamento de instrução, preditores de desvio e generosas quantidades de cache, os núcleos da GPU não possuem nenhuma destas estruturas e quando tem cache é em quantidade modesta. Na GPU a pouca quantidade de cache é compensada pela grande quantidade de threads ativas simultaneamente para reduzir os efeitos da latência da memória.

Na GPU, por ser uma arquitetura SIMT (*Single instruction, multiple thread*), as threads prioritariamente executam emparelhadas para se alcançar o máximo de desempenho. Em outras palavras, na situação ideal, todas as threads da GPU devem estar executando a mesma instrução simultaneamente.

O desemparelhamento das threads ocorre quando em uma determinada instrução de desvio condicional resulta em threads seguindo diferentes ramos de execução. Os arquitetos e programadores de GPU conhecem bem este fenômeno que recebe o nome de divergência por desvio (branch divergence).

Devemos incluir no DTM meios para que o reaproveitamento possa diminuir a ocorrência de divergência nos caminhos de execução e adaptá-lo ao contexto de execução massivamente *multithread*.

Capítulo 3

Graphics Processing Unit

A função primária das unidades de processamento de vídeo consiste em preparar imagens na sua memória interna para serem exibidas na tela de sistemas computacionais. Elas são processadores especializados nas funções da computação gráfica e responsáveis pela visualização de informações. Dada a abundância do poder computacional delas e por serem um computador massivamente paralelo, atualmente também são utilizadas para outros fins de processamento como simulações físicas, processamento de sinais, redes neurais, visão computacional, criptografia entre outras aplicações de alta *performance*.

A GPU(Graphics Processing Unit), originalmente pensada para desenhar e escrever na tela de computadores, aos poucos expôs os processadores paralelos para computação de propósito geral.

A natureza dos algoritmos de computação gráfica influenciou uma arquitetura com elevado grau de unidades funcionais replicadas. Tanto paralelismo torna esta classe de dispositivos capaz de alta vazão no processamento de dados.

De início várias destas unidades funcionais eram especializadas em executar determinados algoritmos gráficos de forma que uma única GPU possuía várias classes de unidades diferentes. Era comum ocorrer um desbalanceamento causando sobrecarga de trabalho em algumas classes e ociosidade em outras. Sua organização evoluiu para uma estrutura mais genérica, programável, contendo unidades funcionais capazes de assumir a responsabilidade por qualquer atividade. Mudar a arquitetura permitiu utilizá-las para outros fins diferentes de processamento gráfico.

Este capítulo esta dividido em sete seções. Começamos comparando CPU e GPU seguido de uma breve introdução às formas de se explorar paralelismo, depois recapitulamos a evolução dos processadores de vídeo, ela revela escolhas arquiteturais ao longo dos anos que resultaram nas GPU de hoje. Apresentamos uma nova vertente de programação, representada pelo OpenCL, que utiliza processadores de vídeo e sistemas heterogêneos como aceleradores de computação. Na sequência detalhamos a arquitetura CUDA, uma instância da computação em GPU, utilizada como

referência arquitetural neste trabalho. Mostramos também a linguagem PTX utilizada na máquina virtual CUDA e pelo simulador GPGPU-sim adotado em nossa implementação, por fim apresentamos o próprio simulador dando detalhes do seu funcionamento.

3.1 CPU vs GPU

Nesta seção descrevemos o funcionamento da CPU, relatando sua diversa gama de aplicações, e fazemos paralelos com as diferenças em relação a GPU. Outros detalhes sobre GPU daremos nas seções seguintes.

A unidade central de processamento é o *hardware* responsável pela execução das instruções dos programas. Em máquinas modernas existem vários co-processadores distribuídos pelo sistema cuidando de funcionalidades muitas vezes de forma invisível para o utilizador do sistema. Por exemplo, os cartões de memória, os discos rígidos e o SSD possuem processador e memória próprios para cuidar da manutenção de setores defeituosos e executar algoritmos responsáveis pela confiabilidade e eficiência destes dispositivos.

O ISA da CPU, muita das vezes, representa a interface externa exposta de um aparato mais sofisticado com diversos elementos de processamento menores coordenados por um micro-programa com objetivo de executar os programas do usuário e do sistema operacional corretamente. Esta divisão em camadas expande-se deste o nível do *hardware* até os níveis mais altos distribuindo responsabilidades da computação entre micro-arquitetura, ISA, sistema operacional, VMs, compiladores e aplicações.

Em alguns casos elementos destas camadas se repetem devido ao uso de máquinas virtuais tornando possível o uso de sistemas operacionais ou mesmo ISA sobrepostos, como por exemplo, nos processadores virtuais como JVM, Python e Chip8 que possuem seu próprio conjunto de instruções (também referidos como bytecodes). Como veremos na seção 3.6, a GPU utiliza abordagem similar para definir o modelo de execução ao adotar uma linguagem intermediária que será traduzida de acordo com o *hardware* instalado.

Costumamos usar o termo CPU para nos referir ao processador principal instalado, enquanto apenas utilizamos os serviços providos pelas outras peças ignorando que ali também reside um ou mais processadores. No decorrer da evolução as GPUs desenvolveram-se tanto a ponto de equipararmos sua importância às da CPU.

Normalmente, dentro da divisão de tarefas entre as camadas de computação, cabe ao sistema operacional a atividade de escalonamento, alocando processos aos núcleos do processador para execução.

Para prover uma melhor utilização dos recursos um sistema pode ser preemptivo

ou não preemptivo. No primeiro caso uma tarefa será executada de um só vez sem interrupção, independente do tempo de execução ou dos recursos solicitados. Esta característica pode levar a desperdícios e ociosidade do processador. Para evitar isto existem os sistemas preemptivos, que possuem um escalonador com poder de interromper um processo em execução e por outro em seu lugar.

Quando um processo ou *thread* precisam ser interrompido o SO deve salvar todos os dados relativos a ela numa ação chamada troca de contexto. Apesar de, em alguns casos, o SO poder contar com a ajuda do processador para realizar esta tarefa com mais eficiência ela continua sendo considerada uma atividade custosa, enquanto na GPU o suporte à troca de contexto ocorre de modo mais eficiente pois é realizado pelo *hardware*.

Processadores diferentes exigem modelos de execução diferenciados, por exemplo, na arquitetura SIMD a mesma instrução executa em diferentes pontos de dados beneficiando, principalmente, aplicações multimídia. Uma das dificuldades de programa-lo é manter ele ocupado preenchendo adequadamente suas faixas de execução. Para isso o compilador/programador devem estar cientes da largura da unidade para utilizá-la adequadamente. Como veremos na subseção 3.5.3 a GPU assemelha-se a este vetor de processamento e compartilha a mesma dificuldade de manter suas faixas de execução ocupadas, porém possuindo suporte de *hardware* para mitigá-la.

Como o objetivo de dividir a carga de trabalho pelos diversos tipos de sistemas paralelos desenvolveram-se técnicas, que aplicadas ao problema, auxiliam na exploração de paralelismo. Estes métodos podem ser combinados para melhor adequar-se as característica arquiteturais do processador alvo. Na próxima seção apresentamos algumas das técnicas de paralelismo mais comuns e utilizadas na exploração de sistemas heterogêneo CPU-GPU.

3.2 Paralelismo

Limitações tecnológicas levaram a indústria de processadores a mudar de processadores com um único núcleo para processadores com vários núcleos. Fisicamente tornou-se difícil a confecção de processadores superescalares ainda mais rápidos, pois as altas frequências utilizadas são acompanhadas de aumento na dissipação térmica e consumo de energia.

Diversos aparelhos de nosso uso cotidiano hoje utilizam processadores paralelos. Por exemplo, celulares, tablet, roteadores, video games, leitores digitais, tocadores de mídia e computadores. Dispositivos específicos como a GPU, responsável por prover avançadas capacidades gráficas, possuem centenas de núcleos. Por conta disto, a consideramos um processador massivamente paralelo.

O desperdício de energia que ocorre quando se tenta aprimorar o processador pelo

aumento da frequência de operação alcançou patamares impraticáveis inviabilizando prosseguir por este caminho. Dito isto, a tendência atual é pelo aumento do número de núcleos por circuito integrado, assim podemos esperar processadores com centenas deles num futuro próximo mesmo fora do nicho da GPU. Esta mudança implica na alteração do paradigma de programação de serial para paralelo.

Para atingir alto desempenho, extraindo o máximo possível, dos sistemas paralelos precisamos decompor o problema em pedaços cada vez menores até que tenhamos tarefas de tamanho e em quantidade suficiente para se executar paralelamente. Uma abordagem ingênua frequentemente leva à subutilização do processador.

As subdivisões podem ter diferentes tamanhos e a independência entre elas é desejada mas não obrigatória. Porém, quanto mais independentes forem, mais facilmente pode ser atingido o paralelismo.

Chamamos decomposição em grão fino quando as subdivisões do problema geram muitas tarefa. Quando elas geram poucas chamamos de grão grosso. Enquanto no primeiro caso favorece a concorrência e escalabilidade no segundo tende a exigir menos comunicação entre as tarefas e uma menor necessidade de sincronismo entre elas. A escolha se por grão fina ou grão grosso, dependerá muito do problema a ser resolvido e das características do sistema onde ele será executado.

3.2.1 Técnicas de Decomposição

Algumas técnicas podem ser empregadas para decomposição do problema. As mais comuns são decomposição recursiva, de dados, funcional, exploratória e especulativa.

Decomposição Recursiva

Utiliza a estratégia de dividir-e-conquistar na subdivisão do trabalho. Nesta técnica o problema é dividido em sub-problemas que por sua vez são divididos em pedaços menores, prosseguindo recursivamente até que a tarefa a ser executada seja de solução trivial.

Em seguida os resultados encontrados são juntados para formar a solução do problema inicial. Frequentemente esta técnica consegue dividir o problema em tarefas independentes o que favorece o paralelismo. Problemas de ordenação costumam se aproveitar da recursão.

Decomposição de Dados

Para operar em grande quantidade de dados convém particiona-los de forma a dividir o processamento entre as partes executando o mesmo algoritmo em cada uma das partições. Quanto mais independentes elas forem umas das outras mais favore-

cerão a escalabilidade, aumentando o número de tarefas junto com a quantidade de informação a ser processada.

Podemos particionar tanto a entrada quanto a saída de dados. Utilizamos o primeiro caso quando o resultado não ocupa uma posição fixa, por exemplo, algoritmo de ordenação e função resumo da entrada como o cálculo do máximo, mínimo ou soma.

Já o particionamento dos dados de saída ocorre quando o resultado de uma tarefa pode ser calculado independente do resultado das outras tarefas e exclusivamente em função dos dados de entrada. Por exemplo, no algoritmo de multiplicação de matrizes o cálculo de cada cédula do resultado ocorre de forma independente dos demais e em função apenas dos dados de entrada tornando possível o particionamento de dados de saída neste problema.

Existe ainda a possibilidade do particionamento híbrido e do particionamento de dados intermediário. No híbrido combina-se a divisão tanto da entrada quanto da saída. No intermediário divide-se a computação em passos intermediários necessitando de um estágio adicional de processamento para alcançar o resultado correto. Apesar da etapa adicional esta técnica pode expor um paralelismo outrora não aparente.

A decomposição de dados é muito utilizada quando portamos um programa para GPU. A escalabilidade desta técnica permite um melhor aproveitamento das centenas de núcleos disponíveis neste sistema.

Decomposição por exploração

Utilizamos decomposição por exploração em algorítimos associados a problemas de busca dentro de um espaço amplo. Dividimos a busca em pedaços independentes para executarem concorrentemente. O primeiro a encontrar a solução sinaliza aos demais o termino da busca. Por exemplo, problema das oito rainhas e do passeio do cavalo.

Decomposição especulativa

Na decomposição especulativa o paralelismo decorre da execução dos diversos caminhos de execução possíveis em paralelo antes de possuir as informações suficientes para definir o caminho correto.

Quando o valor do elemento especulado torna-se conhecido guarda-se o resultado das computações realizadas no caminho correto e descartam-se os demais. Apesar de dispendiosa esta técnica pode diminuir o tempo de resposta geral através do aproveitamento de sistemas paralelos.

Decomposição Funcional

Na decomposição funcional, diferente da decomposição de dados, divide-se o programa em tarefas menores. O pipeline, por exemplo, é um exemplo deste tipo de decomposição. Nele as etapas necessárias para se executar uma instrução são divididas entre vários estágios. No caso ideal, para cada instante de tempo um estágio executará um pedaço de uma instrução diferente.

Este método de divisão possui problemas de escalabilidade e de balanceamento. Ele não escala muito bem, pois o tamanho do programa limitará o máximo de divisões possíveis. O outro problema está em dividir as tarefas igualmente entre os elementos de processamento, evitando o desperdício causado pelo termino de uma antes das outras. Na seção seguinte veremos, como na história da GPU, a questão do desbalanceamento foi resolvida apesar das limitações da decomposição funcional.

3.3 Elemento de Processamento

Uma GPU é composta por centenas de elementos de processamento organizados em um multiprocessador vetorial. Cada elemento de processamento é capaz de executar tanto funções gráficas quanto servir como um processador de propósito geral. Seu processo de desenvolvimento começou influenciado pelas demandas de aplicações gráficas e atualmente consegue atender os requisitos gráficos e de *GPU Computing*.

Antes de receber o nome de GPU a unidade de vídeo era muito simples e chamada apenas de VGA(Video Graphics Array). Ela não possuía um processador e era composta apenas da memória(*buffer*) para armazenar a imagem a ser exibida na tela e circuitos específicos responsáveis por transformar aquela informação binária em um sinal analógico de vídeo.

Com o tempo, o *hardware* da GPU incorporou funções anteriormente executadas pela CPU aumentando o desempenho destas tarefas. Nesta época, surgiram APIs capazes de avançadas funções gráficas permitindo abstrair a utilização de recursos visuais com chamadas de alto nível. Por exemplo, OpenGL [13] e Direct3D [14]. Estas interfaces são bem documentadas e facilitam o desenvolvimento de aplicações visuais sofisticadas, garantindo desta forma grande aceitação do mercado, sobretudo dos Jogos [15].

O estabelecimento de padrões permitiu aos produtores de *hardware* implementarem em circuito certas funções da API que antes eram executadas via software na CPU, servindo como aceleradores. Em 2000 o desenvolvimento das VGAs agregou tantas funções chegando ao nível de poder executar pequenos trechos de código. A partir deste ponto este dispositivo se tornou um processador e por isso foi batizado de GPU pela indústria.

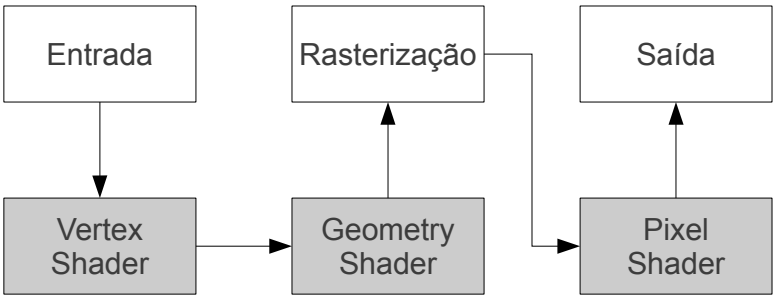


Figura 3.3.1: Pipeline da GPU com unidades distintas.

Um ciclo virtuoso surgiu e a liberação das especificações das APIs procuravam sincronizar com os lançamentos de novas GPUs e vice versa. Esta sintonia, alimentada por um mercado sedento por mais poder de processamento, permitiu um rápido avanço em suas capacidades.

Nos seus primeiros anos de vida a GPU não era um processador completo e dentro das etapas do processamento das imagens misturava funções fixas(definidas em circuitos) e programáveis representada por pequenos programas(*shader*) que podiam ser redefinidos pelo programador.

Para ajudar a entender o que estava prestes a acontecer com este dispositivo é necessário compreender como eles funcionavam. Na Figura 3.3.1 pode ser visto como eram organizados os principais estágios do pipeline de uma GPU em suas primeiras gerações. Os elementos programáveis estão destacado na figura são eles: *Vertex Shader*, *Geometry Shader* e *Pixel Shader*.

O pipeline da GPU divide-se em estágios podendo ser programáveis ou de funções fixas. Cada um fica responsável por uma determinada atividade e possui entradas e saídas interconectadas [13]. O resultado de um estágio serve de entrada para o seguinte. Os principais estágios desta arquitetura executam as seguintes atividades:

- **Entrada**: Fornece os dados que deverão ser processados (triângulos, linhas e pontos);
- **Vertex Shader**: Estágio programável responsável pelo processamento dos vértices. Ele recebe um único vértice na entrada e produz um único vértice na saída. Nele normalmente são executadas transformações, iluminação ou pré-processamento para outros estágios do pipeline.
- **Geometry Shader**: Um estágio também programável encarregado pelo processamento de primitivas (triângulos, linhas e pontos). Ele recebe uma primitiva como entrada e pode produzir zero ou mais primitivas na saída. Ele também permite mudar suas topologias.

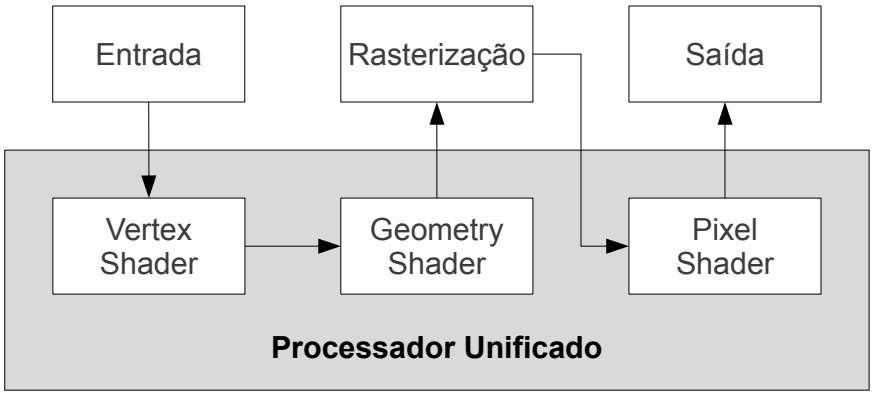


Figura 3.3.2: Pipeline da GPU com unidades unificadas.

- **Rasterização**: Neste estágio as primitivas são aparadas e preparadas para o Pixel Shader. Rasterizar significa converter informação vetorial em um mapa de bits (bitmaps) para compor uma imagem que contém a descrição de cada pixel.

- **Pixel Shader**: Recebe o bitmap gerado na rasterização e fez um processamento em cada pixel. Neste estágio filtros e efeitos podem ser aplicados como tratamento da imagem rasterizada.

Algumas das linguagens disponíveis para serem utilizadas na programação dos shader são: HLSL(High Level Shading Language) [16] incluso no Direct3D, GLSL (OpenGL Shading Language) [17] utilizada no OpenGL e Cg [18] desenvolvida pela NVIDIA.

Em sua trajetória as GPUs evoluíram de um pipeline engessado com função definida em *hardware* para um processador paralelo programável. Aos poucos as unidades funcionais especializadas em executar algoritmos específicos foram sendo substituídos por unidades mais genéricas e programáveis. Em um determinado momento fez sentido unificar estas unidades distintas em um único processador. As várias classes de unidades diferentes foram substituídas por um conjunto de processadores programáveis também conhecidos como elementos de processamento.

O pipeline da GPU moderna possui um elemento de processamento capaz de executar diversas funções. Sua versatilidade permitiu rearranjá-lo de forma que o fluxo de dados passe várias vezes por ele como exibido na Figura 3.3.2. Esta organização viabiliza uma melhor distribuição da carga de trabalho entre as unidades funcionais disponíveis permitindo um melhor balanceamento.

A substituição de várias unidades distintas por uma única unidade mais robusta permitiu um melhor aproveitamento do espaço e uma economia na quantidade de

transistores utilizados, favorecendo a inclusão de mais unidades funcionais na área disponível.

3.4 GPU Computing

Computação em GPU ou GPU Computing é a capacidade de utilizar a GPU junto com a CPU para acelerar o desempenho de aplicações em diversas áreas do conhecimento. Ela tem sido utilizada por instituições que demandam alto poder de processamento em seus datacenters [19] por terem custo reduzido e ocuparem menos espaço que um grande servidor convencional com a mesma capacidade de processamento.

As características de custo/benefício e eficiência energética das GPUs têm permitido que grupos menores com orçamentos limitados tenham acesso ao poder de processamento dos super-computadores de apenas alguns anos atrás. A aceleração de aplicações em GPU permite liberar significativa carga de trabalho das CPUs permitindo uma divisão de trabalho entre elas de forma a explorar as particularidades de cada um. Distribuir corretamente partes do problema entre CPU e GPU permite alcançar uma aceleração maior do que se fosse utilizado apenas CPUs ou apenas GPUs isoladamente. Para tanto, as técnicas de decomposição dos problemas apresentadas na seção 3.2 podem ser utilizadas.

A computação em GPU começou quando os processadores de vídeo acumularam grande capacidade de processamento para atender o aumento na qualidade gráfica de computadores domésticos tornando-se interessante utilizá-los para outros fim diferentes daqueles previstos originalmente.

Surgiram iniciativas para liberar o potencial das GPUs para processamento de aplicações científicas que outrora exigiam um super-computador. Alguns exemplos desta empreitada podem ser vistos em [20], [21], [22], [23] e [24]. A possibilidade de ter ganhos de desempenho com um baixo investimento em *hardware* se encarregou de popularizar a técnica chamada de GPGPU(General Purpose Computing on Graphics Processing Units).

De início a única forma de acessar o pipeline da GPU era por meio das APIs gráficas disponibilizadas. Tanto OpenGL [13] quando Direct3D [14] serviam à este propósito. Esta abordagem era extremamente trabalhosa e exigia a engenhosidade dos programadores para mapear os problemas a serem resolvidos (não gráficos) em chamadas das APIs gráficas. Com tanto empecilho, apenas especialistas com profundo conhecimento do interior das GPUs e dos problemas a serem resolvidos conseguiam domar este potencial computacional.

Felizmente, este movimento não passou despercebido pelos fabricantes e nas atuais gerações incluíram capacidade de executar programas mais complexos. De fato,

alguns de seus pioneiros envolveram-se no desenvolvimento de novas GPUs e ajudaram a conduzir as melhorias. Com o intuito de permitir às unidades unificadas se comportarem de modo mais parecido com um processador de propósito geral adicionou-se ao pipeline novos elementos arquiteturais. Desta forma não se necessita mais de improvisos com APIs gráficas para fazer computação de propósito geral na GPU.

3.4.1 Computação Heterogênea

A disponibilidade, em um único sistema, de diversos processadores com características arquiteturais bem distintas recebe o nome de Computação Heterogênea. O potencial deste arranjo está em aproveitar de cada arquitetura o melhor que ela pode oferecer.

Por exemplo, utilizamos a Decomposição Funcional para dividir um programa em tarefas menores. Deliberadamente agrupamos as atividades com características computacionais semelhantes e identificamos qual arquitetura teria maior eficiência em executá-la.

Para fins de ilustração, poderíamos alocar na GPU a parte do programa responsável por intensos cálculos de ponto flutuante enquanto poderíamos deixar para CPU a tarefa que tivesse um uso de memória mais intenso. Neste caso as centenas de unidades funcionais de ponto flutuante da GPU proporcionaria alta vazão de processamento enquanto a grande quantidade de cache da CPU diminuiria a latência no acesso aos dados guardados na memória.

Cada uma destas tarefas são distintas em suas características computacionais e padrões de acesso a memória. Uma análise cuidadosa antes de decidir como distribuir as atividade é determinante para se alcançar maior *performance*. Como exemplo de ferramentas para Computação Heterogênea temos CUDA [25], OpenCL [5], OpenMP [26] e OpenACC [27].

Atualmente, dois destes frameworks são dominantes. Um é open source e se chama OpenCL [5] o outro pertence a NVIDIA e se chama CUDA [25]. Ambos fornecem ferramentas para se programar as GPUs em linguagens variantes do 'C' e ajudaram a disseminar o conceito de Computação em GPU e Computação Heterogênea. Seus desenvolvimentos continuam ativos. Eles incluem ainda a capacidade de programar multiplas CPUs consolidando-se como um ambiente de programação paralela para um sistema heterogêneo.

3.4.2 OpenCL

OpenCL é um framework para programação paralela fornecedor de linguagem, bibliotecas e API necessárias para programação de sistemas heterogêneos compostos

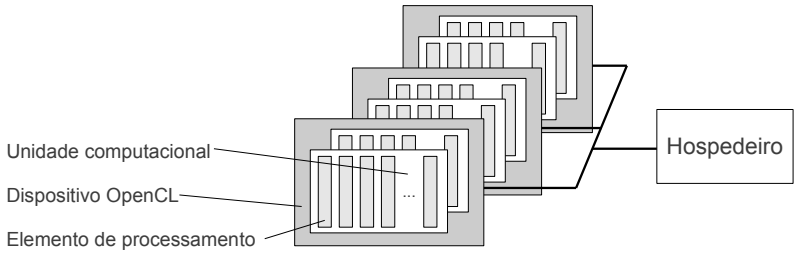


Figura 3.4.1: Organização física do OpenCL mostrando a localização do hospedeiro, dos dispositivos OpenCL, das unidades computacionais e dos elementos de processamento.

por várias CPUs, GPUs e outros tipos de processadores. Sua linguagem baseada em 'C' permite a escrita de programas portáteis sem perder eficiência. Detalhes do *hardware* também são expostos pela API permitindo otimizações em baixo nível, se desejável.

O framework representa o sistema como um hospedeiro com um ou mais dispositivos OpenCL conectados. Um dispositivo pode conter uma ou mais unidades computacionais (compute unit) que por sua vez pode conter um ou mais elementos de processamento (PE - processing elements), como pode ser visto na Figura 3.4.1.

Um programa escrito em OpenCL divide-se em hospedeiro e *kernel*. O primeiro executa na CPU enquanto o segundo na GPU. Através de chamadas específicas da API o hospedeiro pode comandar a execução do *kernel* em um ou mais dispositivos OpenCL. Antes de ser submetido, o código do *kernel* precisa ser compilado na arquitetura alvo do dispositivo. Normalmente o hospedeiro e o dispositivo OpenCL possuem compiladores diferentes por possuírem arquiteturas distintas.

A etapa de compilação pode ser on-line ou off-line. A escolha do tipo de processo deve levar em consideração o ambiente de execução e o objetivo a ser alcançado. Sistemas de tempo real e plataformas heterogêneas demandam diferentes características que estes modos podem oferecer.

Na compilação on-line, ou JIT(Just in time compilation), o programa hospedeiro possui o código fonte do *kernel*. Neste modo a compilação ocorre no momento da execução. O programa hospedeiro invoca o compilador e em seguida envia o binário para o dispositivo OpenCL.

Este tipo de abordagem adiciona um atraso, prejudicando o tempo de resposta. Entretanto, em sistemas heterogêneos com uma grande diversidade de dispositivos distintos, a geração do binário sob demanda pode diminuir significativamente o tamanho do executável.

Já na compilação off-line o código do *kernel* é compilado antes. O hospedeiro carrega o binário e o transmite para o dispositivo. Sistemas com vários dispositivos exigem o carregamento de diversos binários. Em sistemas de tempo real, o ganho de tempo por ter o código previamente compilado pode ser crítico para o sucesso. Em contrapartida, se o sistema possuir muitos dispositivos diferentes manter o binário para múltiplos alvos pode ser custoso demais [28].

OpenCL oferece flexibilidade no mapeamento da computação nos elementos de processamento. Ele permite tanto um controle de fluxo convergente como também divergente. O fluxo convergente ocorre em execuções onde uma instrução executa simultaneamente em vários elementos de processamento. Dispositivos SIMD se beneficiam deste tipo de controle. Quando os elementos de processamento executam instruções diferentes trata-se de um fluxo divergente que ocorre sempre quando o resultado de um desvio toma caminhos diferentes entre os PEs.

Cada *kernel* executa com um contexto associado gerenciado pelo hospedeiro. O contexto de execução inclui quatro recursos: dispositivos, objetos de *kernel*, objetos de programa e objetos de memória. Os recursos são responsáveis por executar, passar argumentos, armazenar fontes e binários e enviar dados para o dispositivo, respectivamente.

O hospedeiro utiliza a API para se comunicar com os dispositivos através do envio de comandos. Para organizá-los cada dispositivo possui dois tipos de filas, uma interna e outra externa. Existem três tipos de comandos possíveis: Enfileiramento de *kernel* para execução, transferência de memória entre hospedeiro/dispositivo e comandos de sincronismo.

A fila externa, chamada de command-queue, recebe comandos do hospedeiro enquanto a fila interna, chamada de *device-side command queue*, recebe comandos dos próprio *kernel* tornando possível a existência de *child kernel*.

Os comandos nas filas de execução passam por seis estados durante sua existência:

- **Enfileirado** (Queued)
- **Submetido** (Submitted)
- **Pronto** (Ready)
- **Executando** (Running)
- **Terminado** (Ended)
- **Completo** (Complete)

Um comando entra na fila no estado enfileirado e passa para o estado submetido quando enviado para o dispositivo. Satisfeitos os requisitos para sua execução ele

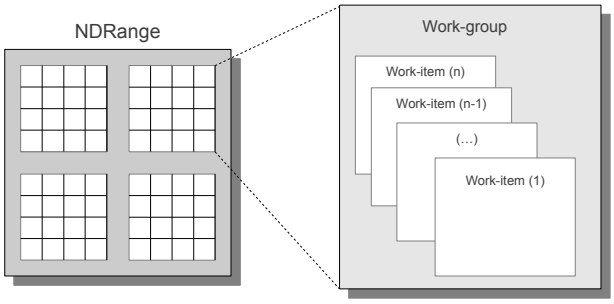


Figura 3.4.2: Espaço de indexação do OpenCL. Os work-items são agrupados em work-groups que por sua vez são agrupados em um NDRange.

passa para o estado de pronto e fica aguardando o escalonamento. Os estados executando e terminado marcam o início e o final da execução do comando que passa para o estado completo apenas quando todos os seus comandos filhos terminarem de executar.

A execução do *kernel* ocorre dentro de um espaço de indexação especial chamado NDRange, mostrado na Figura 3.4.2. Nele são criados a quantidade de linhas de execução definidas no seu lançamento as quais recebem o nome de work-item e funciona como se fosse uma *thread*. Os work-items são agrupados em work-groups que por sua vez são agrupados em um NDRange. Os work-groups existem para permitir comunicação e cooperação entre os work-items.

Alguns problemas paralelos necessitam de cooperação entre os fluxos de execução para serem resolvidos. No OpenCL, os work-items comunicantes devem necessariamente esta no mesmo work-groups, pois visto que são de um sistema de memória compartilhada precisam utilizar a memória sincronizadamente para comunicação. Para tanto, utiliza-se primitivas de sincronismos para restringir a ordem de execução entre duas ou mais unidades de execução protegendo a região crítica e evitando uma condição de corrida.

Paralelismo de dados é o modelo de paralelismo que melhor se adapta ao modelo de execução do framework. A divisão das tarefas acontece pela divisão dos dados entre os work-items. Cada work-item e work-group possuem IDs que servem como coordenadas para localizar em qual parte da memória estão os dados de entrada e os locais onde serão armazenados os resultados.

O **modelo de memória** presente no OpenCL divide a memória do dispositivo em quatro regiões:

- **Global**: Permite leitura e escrita por todos os work-items contidos no mesmo contexto de execução, mesmo que estejam presentes em work-group diferentes;

- **Constante**: Mantêm seu valor durante toda execução do *kernel*;
- **Local**: Funciona como uma área comum para comunicação entre work-items no mesmo work-group;
- **Privativa**: Pertence apenas a um work-item e por mais nenhum outro.

As regiões de memória são disjuntas, ou seja, não se sobrepõe. O OpenCL expõe uma divisão lógica das regiões de memória. Contudo, a implementação das divisões podem ser partições de uma mesma memória física. Além disso ele suporta o uso de memória cache. Entretanto, isto depende do suporte do dispositivo instalado.

OpenCL fornece as abstrações necessárias para programação de diversos tipos de dispositivos diferentes. Nitidamente sua organização foi influenciada pelas características presentes nas GPUs, porém sua estrutura não é tão amarrada a nenhum *hardware* específico. Isto ajuda a ter uma visão mais ampla de como operar um sistema heterogêneo. Sua característica de open source e a participação ativa de diversos fabricantes relevantes o tornam uma base de conhecimento com potencial de sobreviver por várias gerações de *hardware*.

3.5 Arquitetura CUDA

Compute Unified Device Architecture é o framework de programação paralela criado pela NVIDIA capaz de utilizar GPUs para computação de propósito geral. A versão mais recente ampliou sua capacidade para permitir a programação de CPUs de múltiplos núcleos. Em decorrência disso, assim como OpenCL, CUDA também pode ser utilizado para programação heterogênea. Para esta seção nos baseamos nas informações e adaptamos as imagens contidas em [15], [29], [30], [25].

Seguindo a popularidade do GPU Computing incluiu-se no *hardware* das GPUs os mecanismos necessários e o suporte de drivers para que elas pudessem ser programadas com mais facilidade. Com esta melhoria, o paradigma computação evoluiu culminando em um modelo de programação e uma plataforma de software capaz de fornece a interface necessária para acessar os recursos computacionais da GPU sem a necessidade de utilizar API gráficas.

Uma aplicação gráfica moderna demanda muito poder de processamento para funcionar e para atender estas exigências as GPUs precisam de alta vazão de processamento. Características específicas de problemas gráficos influenciaram o desenvolvimento da arquitetura da GPU para que ela pudesse oferece alta vazão e escalabilidade.

Como os algoritmos de processamento de imagem presentes no pipeline da GPU apresentam pouca dependência de dados e regularidade nas estruturas de dados,

elas evoluíram para uma densa matriz de processadores operando sob uma única lógica de controle.

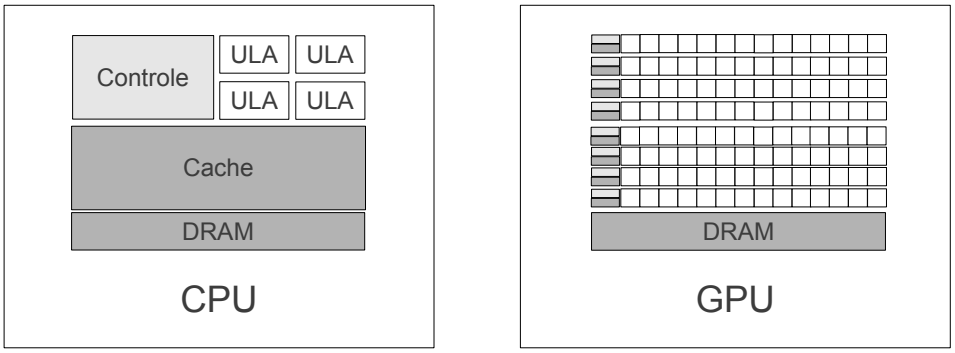


Figura 3.5.1: Comparativo entre a distribuição de transistores na CPU e na GPU. As GPUs possuem mais transistores voltados para operações lógicas/aritméticas enquanto as CPUs devotam mais para memória cache.

Decisões arquiteturais optaram por ter mais transistores dedicados a ULAs do que memória cache. Comparado com as CPUs muito mais área na GPU dedica-se a elementos de processamentos do que memória, como mostra a Figura 3.5.1. Largos barramentos de dados permitem o fluxo constante de informação a ser processada na matriz de núcleos garantindo uma vazão expressiva. A alta independência dos dados propicia que sempre terá trabalho para ser executado sem que seja necessário aguardar a resposta de operações anteriores.

CUDA expõe um *hardware* concebido para produzir gráfico onde cada núcleo (shader) executa um *shader program* em uma vasta quantidade de dados. Para traduzir esta característica para programação de propósito geral ela utiliza um modelo SPMD (single program multiple data), onde o programador cria um programa que será instanciado em milhares de *threads* e executado em centenas de núcleos. Neste modelo, alta *performance* resulta da farta quantidade de *threads* presente em número muito superior ao de núcleos instalados.

Nesta arquitetura o suporte a *threads* ocorre em nível de *hardware* garantido alta eficiência nas constantes trocas de contexto envolvidas. A medida que as trocas ocorrem com mais frequência melhor acontece a ocultação da alta latência dos acessos à memória.

Para melhor explorar o paralelismo oferecido pelo *hardware* o programador deve decompor os dados do problema procurando minimizar a dependência entre as divisões geradas. Aplicações que manipulam terabytes de informação, como bioinformática [31], financeiras [32], dinâmica de fluidos [33], visão computacional [34], análise numérica [35], entre outras científicas tem potencial para gerar divisões suficientes do trabalho para manter uma grande quantidade de núcleos ocupadas.

3.5.1 Hierarquia de *threads*

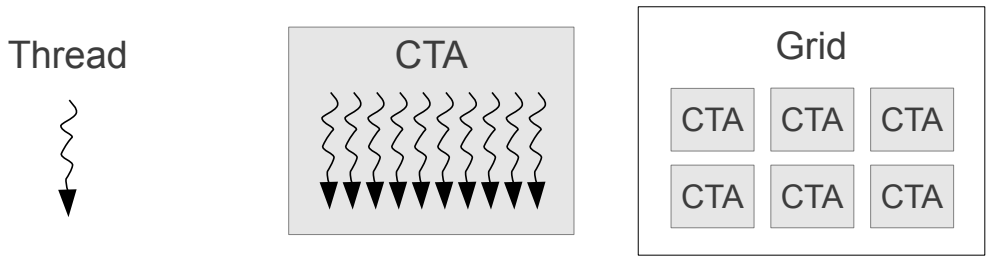
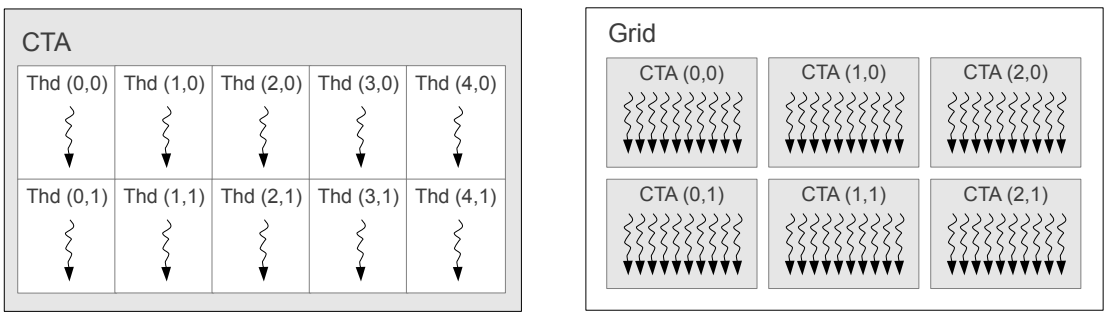


Figura 3.5.2: Hierarquia de *threads*.

A maneira como as *threads* executam e acessam a memória obedecem à hierarquia mostrada na Figura 3.5.2. Isoladas elas possuem seu próprio contador de programa, registradores de dados e memória privativa. Em grupo, chamam-se CTA (Cooperative Thread Array) e cooperam na solução do problema, nele as *threads* compartilham e se sincronizam no acesso aos diversos tipos de memórias. A Figura 3.5.3a expõe as *threads* no interior do CTA [29].

Threads em CTAs diferentes não podem se comunicar, pois sujeitam-se a ocorrência de condição de corrida podendo tornar o dado lido da memória indeterminado. Um conjunto de CTAs recebe o nome de Grid, dentro dela todas as *threads* executam o mesmo programa. A Figura 3.5.3b exibe um Grid e os CTAs em seu interior.

A hierarquia de *threads* possui um sistema de coordenadas utilizado para mapear a divisão do problema. Elas podem possuir uma, duas ou três dimensões escolhidas pelo programador de acordo com o domínio dos dados. Através deste sistema podemos, por exemplo, determinar que a *thread* (2,0) do CTA (1,1) irá acessar dados diferentes dos da *threads* (2,0) do CTA (1,0) desta forma os identificadores ajudam a localizar a parte dos dados que elas operarão.



(a) Threads são agrupadas em CTAs (b) CTAs são agrupados em GRIDs

Figura 3.5.3: Agrupamentos na hierarquia de *threads*

3.5.2 Microarquitetura

Fisicamente as *threads* executam em processadores simples chamados SPs (*Scalar Processor*) herdados do *pipeline* gráfico. Eles não possuem preditor de desvio, nem contam com execução fora de ordem. Em contrapartida as GPUs possuem centenas destas unidades replicadas.

Um conjunto de SP formam uma unidade maior chamada SM (*Streaming Multiprocessor*). A quantidade de SP dentro do SM varia de acordo com o modelo de GPU e tem aumentado a cada nova geração de arquitetura elevando ainda mais o paralelismo. A Tabela 3.5.1 mostra a evolução da quantidade de SPs ao longo das gerações.

Modelo	Arquitetura	SM	SP
GeForce 8800 GTX	G80	16	128
GeForce GTX 280	G200	30	240
GeForce GTX 580	Fermi	16	512
GeForce GTX 680	Kepler	8	1536

Tabela 3.5.1: Quantidade de Scalar Processors (SP) e Streaming Multiprocessors (SM) em diversos modelos de diferentes gerações.

O escalonador distribui os CTAs dentro da Grid pelos SM disponíveis. Outro escalonador distribui as *threads* no interior do CTA pelos SP. Uma GPU pode possuir vários SM. Como pode ser visto na Figura 3.5.4, *thread* mapeia no SP, CTA no SM e Grid na GPU.

A divisão do trabalho pelo uso das *threads*, CTAs e Grid permitem uma abordagem do problema independente da geração do *hardware* utilizado. Este sistema prove escalabilidade, pois obriga o programador a dividir o problema em partes independentes que podem ser executados sem nenhuma restrição de ordem.

Devido a esta característica, aumentar o número de SM a cada nova geração de GPU permite maior número de CTAs executando simultaneamente sem a necessidade de reescrever o código. Em outras palavras, o mesmo programa irá executar mais rápido em uma GPU com 16 SM do que uma com 8 SM. Desta forma CUDA provê uma camada de abstração para sistemas paralelos capaz de escalar em *performance* com o aumento do número de núcleos.

3.5.3 Modelo de Execução

O programa executado na GPU pode ser escrito em CUDA C, uma extensão da linguagem C criada para expor o paralelismo presente nos processadores gráficos. Uma função especial denominada *kernel*, quando chamada, executa N vezes em paralelo no dispositivo. Neste modelo a GPU opera como um co-processador aliviando

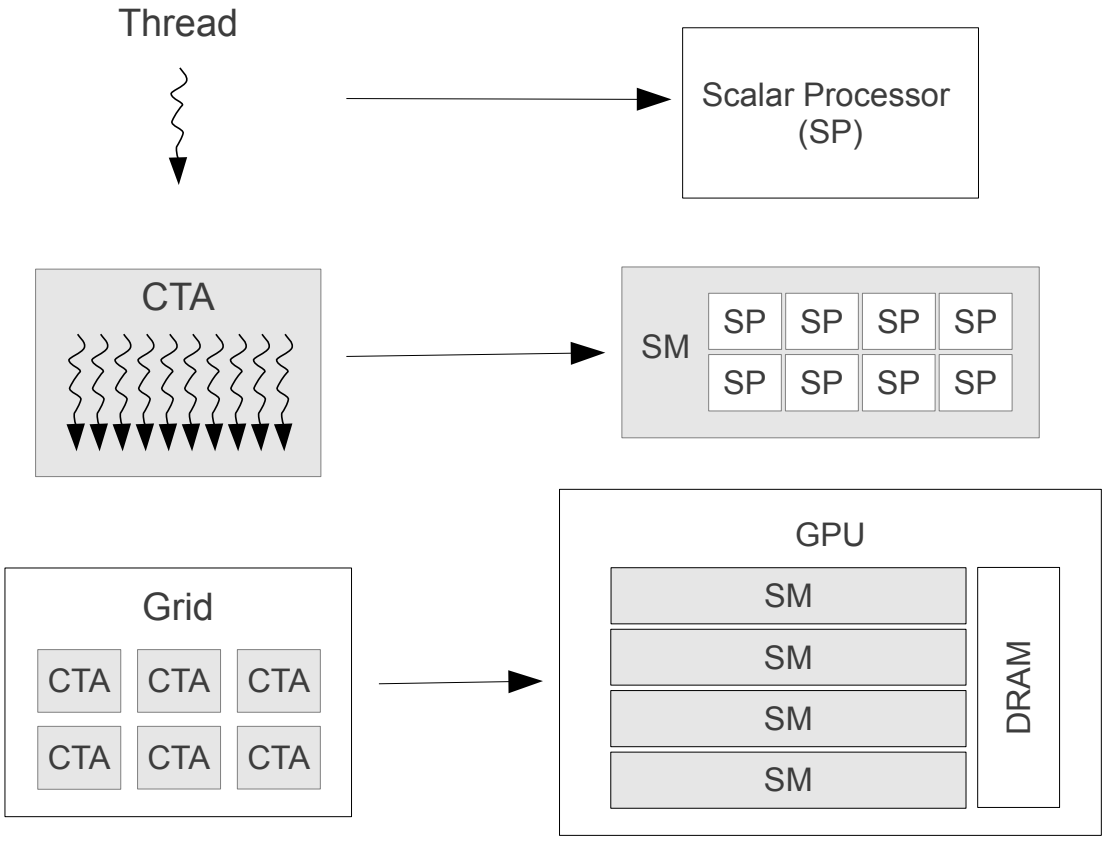


Figura 3.5.4: Mapeamento da Hierarquia de *threads* na GPU.

a carga de trabalho da CPU.

```
//Função Kernel executada na GPU
__global__ void somaVetorial( int n, float *A, float *B, float *C ) {

        int i = threadIdx.x;
        C[ i ] = A[ i ] + B[ i ];
}

int main(){
        // código executado na CPU

        // Lançamento do Kernel com n threads
        somaVetorial<<< 1, n >>>(n, A, B, C);

        // código executado na CPU
}
```

Figura 3.5.5: Exemplo de código CUDA.

A Figura 3.5.5 mostra um exemplo de código em CUDA C cuja função *kernel* chama-se *somaVetorial()*. Identifica-se ela pela palavra reservada *__global__* presente antes da declaração de seu nome. Em sua linha de chamada, além dos parâmetros

definidos no padrão C, estão os parâmetros indicando a quantidade de *threads* para execução. Estes parâmetros adicionais aparecem compreendidos entre <<<>>> e determinam o número de *threads* instanciadas na GPU, neste exemplo *n*. A função *main()* inicia o programa executado na CPU, quando atingido determinado ponto ela chama a função *kernel somaVetorial()* despachando o trabalho para GPU.

A NVIDIA classifica seu sistema como SIMT (Single Instruction Multiple Threads) e segue o modelo de decomposição de dados, nele cada *thread* executa o mesmo programa em pedaços diferentes dos dados. Para identificar onde ela deve atuar utiliza-se um identificador exclusivo para cada uma. No exemplo da Figura 3.5.5 o comando *threadIdx.x* identifica a *thread* e determina qual parte do vetor somar. Neste exemplo, são criadas n *threads* para somar dois vetores de n elementos de forma que cada uma fica responsável pela soma de uma única posição do vetor. Alguns autores usam o termo SPMD em vez de SIMT em uma referencia a similaridades como o modelo SIMD [15]

No fluxo de execução de um aplicativo CUDA, exibido pela Figura 3.5.6, uma *thread* de CPU fica responsável por copiar os dados para GPU, configurar os parâmetros de execução, invocar o *kernel* e copiar o resultado do dispositivo para o hospedeiro. Tanto a quantidade de *threads* quanto a de CTAs são parâmetros definidos no lançamento do *kernel*. Durante a execução do programa um mesmo *kernel* ou diferentes *kernel*s podem ser invocados diversas vezes com diferentes quantidades de *threads* e CTAs.

Os CTAs podem ser executados concorrentemente ou em paralelo nos SM dependendo do dispositivo. Como a ordem de execução dos CTAs não é relevante, o escalonador é livre para fazer distribuição deles pelos SM disponíveis.

As *threads* dentro do CTA concorrem pelos SP, elas executam em grupos chamados warps. As *threads* dentro de um warp executam a mesma instrução simultaneamente. Sempre que alguma *thread* acessar a memória todo warp é bloqueado e dá lugar ao warp seguinte, tornando a ser executada quando seus dados vindos da memória estiverem prontos.

Com uma quantidade de *threads* suficiente dentro do CTA a latência de acesso à memória pode ser ocultada. O tamanho do warp depende do dispositivo e seu valor pode ser consultado em tempo de execução pela constante definida WARP_SZ.

3.5.4 Modelo de Memória

A arquitetura CUDA divide o sistema em memória do hospedeiro e memória do dispositivo. A primeira é a RAM do sistema, nela reside o Sistema Operacional e seus processos enquanto a segunda refere-se a memória contida na placa de vídeo.

A cópia entre estas duas memória acontecem através do barramento ponto-a-

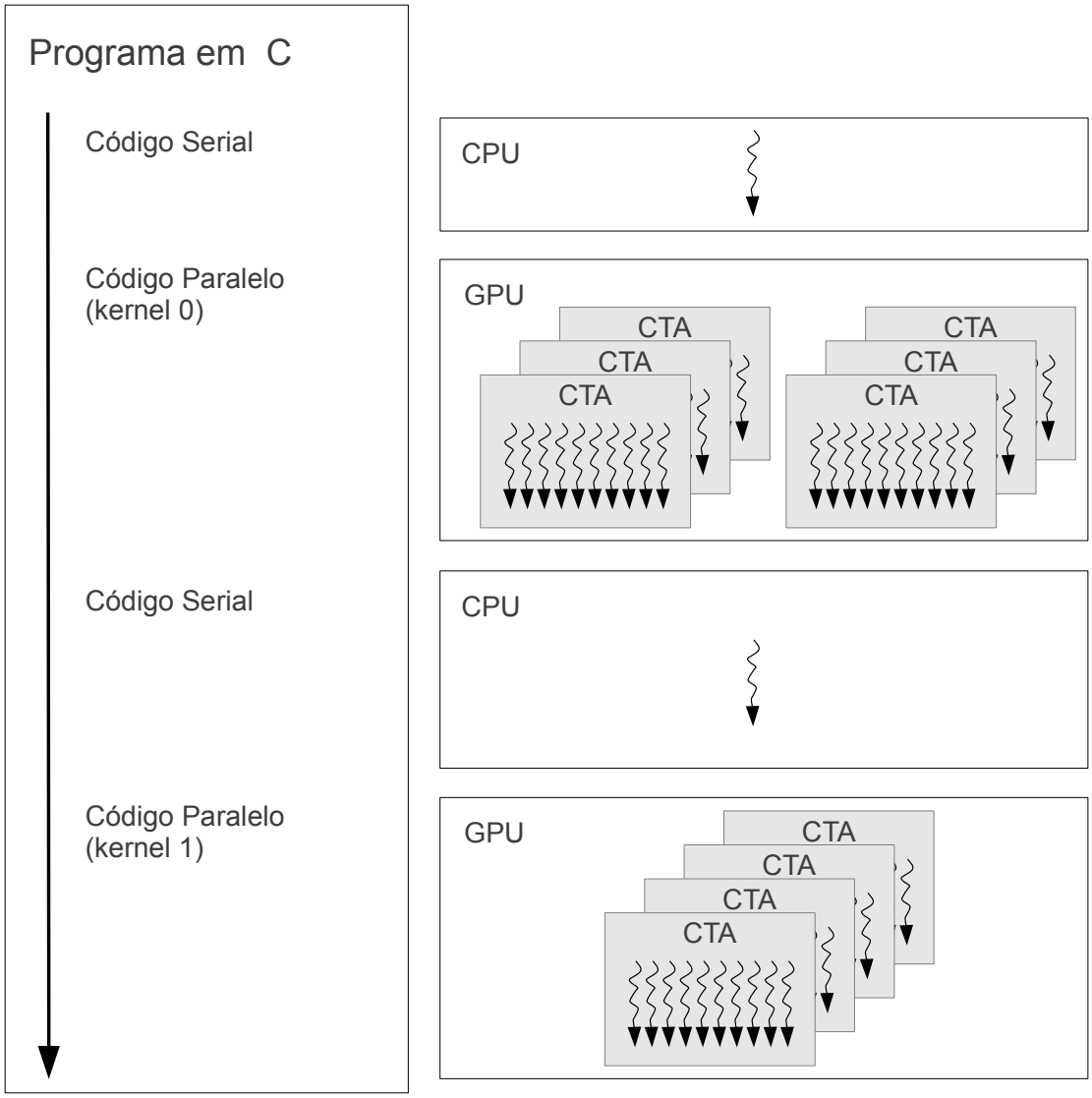


Figura 3.5.6: Fluxo de execução para programação heterogênea.

ponto PCI-e pelo uso de funções fornecidas pela API. A CPU e a memória do hospedeiro ocupa uma extremidade deste barramento enquanto a GPU e a memória do dispositivo a outra. Esta transação sujeita-se a latência inerente à utilização do barramento e cabe ao programador decidir como lidar com ela.

Em uma das abordagens copia-se os dados da aplicação para GPU antes da invocação do *kernel*, em outra utiliza-se o mecanismo que permite cópia de dados simultaneamente a sua execução e uma terceira forma permite acessar a memória do hospedeiro diretamente a partir da GPU. Cada uma das estratégias para alcançar alta *performance* exige uma divisão do trabalho que consiga ocultar as latências envolvidas.

A memória do dispositivo possui seis tipos distintos, cada uma com suas próprias características e utilizações. A cuidadosa distribuição dos dados entre elas permite

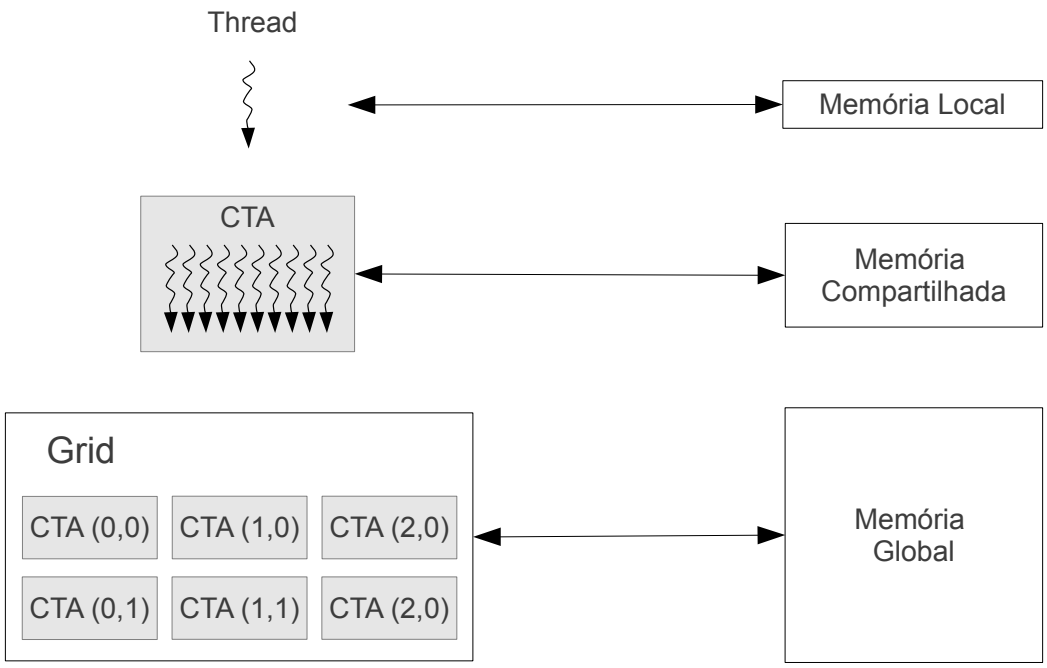


Figura 3.5.7: Hierarquia de memória da GPU (Adaptado).

melhor aproveitar o que tem a oferecer. Dentro do circuito integrado da GPU existem: os registradores e a memória compartilhada. Fora dele a memória global, constante, textura e local.

Cada região da memória do dispositivo possui as seguintes características:

- **Memória Local** (Local memory): A memória local ao contrário do que o nome sugere não está próxima do processador, o termo local se refere ao escopo, por ser privativa de cada *thread*. Ela é lenta porque está situada fora do circuito integrado da GPU, por isso seu uso deve ser evitado.

- **Memória Global** (Global memory): A memória global é o escopo mais externo de memória existente. Este espaço pode ser utilizado para compartilhamento de informação entre as *threads* e entre os dispositivos e o hospedeiro. Entretanto, para comunicação entre *threads* existe uma memória mais apropriada para isso.

- **Memória Compartilhada** (Shared memory): A memória compartilhada, uma das mais velozes mas também de menor capacidade da GPU, dedica-se ao compartilhamento de dados entre *threads* do mesmo CTA. O espaço disponível é particionado entre os CTAs, desta forma alocar mais desta memória significa que menos CTAs poderão coexistir na GPU, diminuindo o desempenho da aplicação.

- **Registradores** (Registers): Os registradores, que têm o mesmo tempo

de acesso da memória compartilhada, são também em número limitado e diferenciam-se da memória compartilhada por serem privativos de cada *thread*. Eles devem ser usados com parcimônia.

- **Textura** (Texture): Não se trata exatamente de um tipo de memória, mas de um método de acesso herdado do pipeline gráfico. Esta categoria contempla acesso a textura e a superfícies. A primeira é somente leitura enquanto a segunda permite leitura e escrita. Ambas possuem cache e mapeiam para o mesmo espaço físico da memória global.

- **Constante** (Constant): A memória constante também é somente leitura e com cache.

Tipo	Tempo de acesso em ciclos
Registrador	0
Compatilhada	0
Constante	0
Local	> 100
Global	> 100
Textura	> 100

Tabela 3.5.2: Custo estimado para acessar os diferentes tipos de memória [30]

Algumas áreas de memória podem ser muito lentas, na Tabela 3.5.2 pode ser vista uma estimativa dos tempos de acesso. As regiões mais rápidas aparecem dentro do circuito integrado e são em menor quantidade enquanto as mais lentas são mais abundantes e encontram-se em chips de memórias externos fixados na placa de vídeo.

Existem duas formas possíveis para se ocultar a latências das memórias mais distantes. Uma utiliza múltiplas *threads* para ocultar o tempo de acesso a memória e a outra adianta o máximo possível as instruções de carregamento, uma vez que elas não são bloqueantes [30], permitindo que o valor esteja disponível mais a frente no código quando for utilizado.

Uma instrução bloqueante trava a execução do código até que o resultado se torne disponível enquanto uma não bloqueante permite o código continuar executando as instruções seguintes que não possuam dependência de dados com ela.

Durante a execução uma *thread* pode necessitar de dados de diferentes espaços de memória, o programador decide entre velocidade e quantidade quando escolhe a declaração dos tipos de dados na GPU. Os tipos de memória com cache favorecem o desempenho, mas inspiram cuidados. Por exemplo, tanto o cache da memória de textura quanto a de superfície não são mantidos coerentes em relação a memória global. Isto implica que quando um dado é gravado na memória global ou na memória de

superfície uma posterior leitura pela memória de textura ou de superfície ao mesmo endereço, dentro da mesma chamada de *kernel*, retornará um valor indefinido.

O modelo de memória fornecido pelo CUDA oferece grande diversidade na maneira como a aplicação poderá acessar os dados. Várias combinações de abordagem entregam diferentes desempenhos tornando difícil o domínio de todo ferramental disponibilizado de forma a extrair o máximo do *hardware*.

3.6 PTX

PTX é uma linguagem intermediaria usada pela máquina virtual da NVIDIA. Ela adiciona uma camada entre a aplicação e o ISA do *hardware* permitindo que a evolução da arquitetura e da micro-arquitetura da GPU ocorra sem perturbar as aplicações já desenvolvidas. Além disso, ela oferece a abstração necessária para se criar um programa paralelo que funcione em um *hardware* com quantidade arbitrária de núcleos. Esta habilidade proporciona escalabilidade ao código permitindo aumento de *performance* pela adição de mais núcleos.

Esta linguagem utiliza predicados para execução condicional. Um predicado posto antes da instrução determina se ela executará ou não. De acordo com o manual o PTX [30] toda instrução pode ser precedida de um predicado. Entretanto, pudemos observar que o compilador, dentro das aplicações analisadas, só utilizou o predicado para diferenciar desvios condicionais de incondicionais.

Entre outras características do PTX podemos citar:

- ISA independente do *hardware* favorece que seja usado como alvo para outros compiladores;
- Oferece a conveniência de abstrações do *hardware* junto com a habilidade de programar em baixo nível permitindo que se escreva códigos manualmente em PTX;
- Desempenho comparável à códigos nativos e permanece estável entre as várias gerações de GPU que já existiram.

A Figura 3.6.1 mostra o caminho que o programa percorre até chegar a GPU e destaca o momento de geração do .ptx.

Uma aplicação inclui vários arquivos entre fontes e cabeçalhos. Os arquivos em C/C++ puro podem ser encaminhados diretamente para o compilador padrão, normalmente gcc. Já os arquivos que incluem também o fonte do *kernel* devem passar pelo nvcc. Nele o código é pre-processado no cudafe e em seguida compilado pelo nvopencc. A Figura 3.6.2 mostra um exemplo de código do *kernel* e Figura 3.6.3 seu correspondente .ptx.

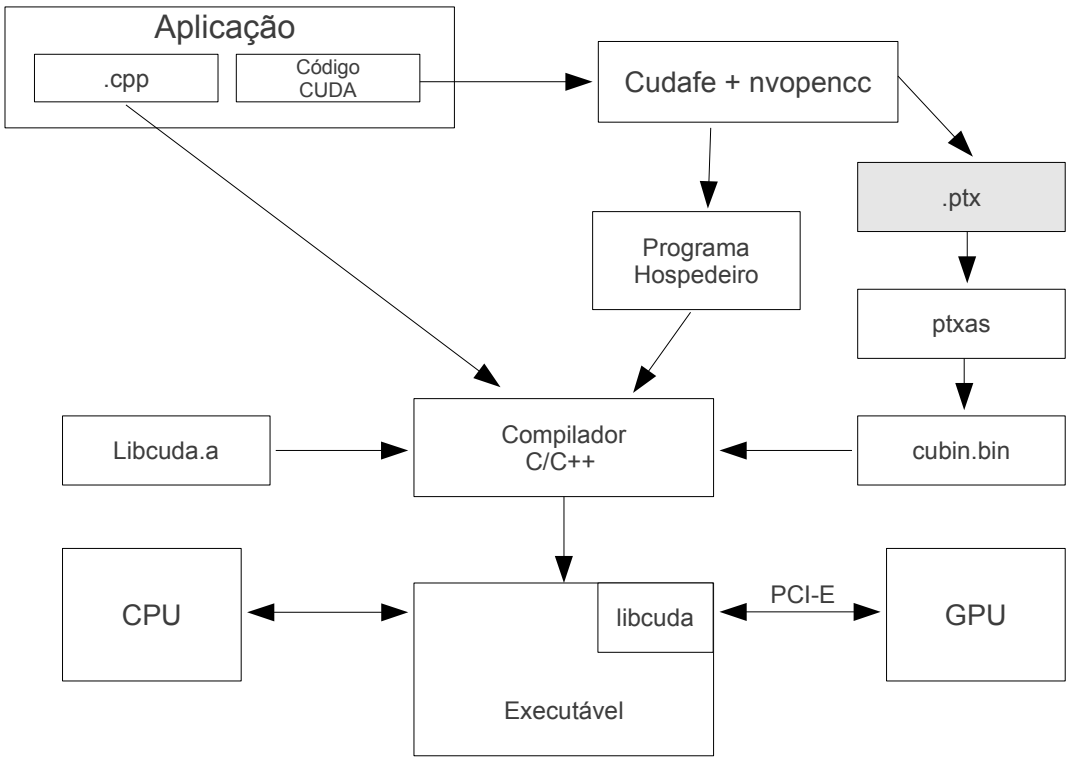


Figura 3.6.1: Fluxo de compilação offline da arquitetura CUDA e geração do PTX.

O compilador nvopencc é uma variante open-source do compilador Open64 que teve ISA alvo alterado para a linguagem PTX. A passada pelo nvcc gera dois arquivos, um com a parte C/C++ do código e outro com o .ptx mostrado na Figura 3.6.3. O primeiro segue para o compilador C/C++ enquanto o arquivo .ptx segue para o compilador específico da arquitetura.

```
__global__ void somaGPU( int n, int *matA, int *matB, int *matC ) {

        int i = threadIdx.x;

        if( i < n ) {
                matC[i] = matA[i]+matB[i]+100;
        }
}
```

Figura 3.6.2: Exemplo de um *kernel*. Código fonte faz a soma de dois vetores.

O .ptx é um arquivo de texto (3.6.3), legível por humanos, resultado da compilação do código fonte do *kernel*. Apesar de ser um código de montagem (assembly) ele possui algumas características de alto nível responsáveis por fazer a abstração das camadas abaixo dele. Isto permite a compilação do *kernel* de forma independente do *hardware* instalado.

Uma das características de alto nível do PTX é a quantidade ilimitada de regis-

```
$somaGPU:
        cvt.s32.u16     %r1, %tid.x;
        ld.param.s32    %r2, [n];
        set.lt.u32.s32  %r6, %r1, %r2;
        neg.s32         %r7, %r6;
        mov.u32         %r9, 0;
        setp.eq.s32     %p1, %r8, %r7;
@%p1 bra                $L1;
        mul24.lo.u32    %r10, %r1, 4;
        ld.param.u32    %r11, [matA];
        add.u32         %r12, %r11, %r10;
        ld.global.s32   %r13, [%r12+0];
        ld.param.u32    %r14, [matB];
        add.u32         %r15, %r14, %r10;
        ld.global.s32   %r16, [%r15+0];
        add.s32         %r17, %r13, %r16;
        add.s32         %r18, %r17, 100;
        ld.param.u32    %r19, [matC];
        add.u32         %r20, %r19, %r10;
        st.global.s32   [%r20+0], %r18;
$L1:
        exit;
```

Figura 3.6.3: PTX correspondente ao *kernel* exibido na Figura 3.6.2 (Adaptado).

tradores, deixando para o compilador seguinte na sequência de compilação a responsabilidade pela alocação dos registradores físicos de fato existentes na arquitetura alvo.

O compilador ptxas gera o binário específico do *hardware* alvo a partir do arquivo .ptx encaminhado do estágio anterior. Esta ferramenta de extrema importância aparece tanto no compilador offline quanto no driver para permitir a compilação online (JIT). No caso da compilação offline, ilustrada pela Figura 3.6.1, ele gera o arquivo binário, cubin.bin, que irá executar na GPU.

O arquivo binário segue para o compilador C/C++, no qual, durante o processo de link-edição, o binário e os demais objetos, entre eles a biblioteca libcuda.a, são anexados ao executável da aplicação. Esta última conduz a comunicação com a GPU, gerencia a compilação online e submete o *kernel* para o dispositivo.

As *threads* dentro de um warp, no caso ideal, executam simultaneamente instrução por instrução. Em outras palavras, apenas uma lógica de controle coordena a execução de um array de SP fazendo cada um executar a mesma instrução simultaneamente. Este sistema conhecido tanto como SIMT ou SIMP permite que a mesma instrução despachada e decodificada execute diversas vezes, uma para cada *thread* no warp.

Uma única instrução controlando várias unidades funcionais assemelha-se ao modelo SIMD onde a largura do processador é exposta ao programador e cabe a ele lidar com divergência no fluxo de execução manualmente, enquanto que no SIMT o controle de divergência é gerenciado automaticamente e as *threads* podem divergir

livremente.

Cada *thread* possui seu próprio contador de programa (pc) que deixa ela independente para seguir seu próprio caminho de execução. Quando após um desvio um grupo de *threads* toma um caminho distinto das demais dizemos que ocorreu uma divergência. Quanto isto acontece o programa terá que executar os dois caminhos possíveis do desvio, em série, impactando severamente o desempenho.

Em um programa com muitos desvios, o quantitativo de instruções executadas cresce rapidamente podendo chegar no pior caso onde todas as ramificações do programa seriam executadas. Por este motivo, para conseguir alta *performance* os programadores de GPU são orientados a repensar os algoritmos que desejam implementar de forma a incluir menos instruções de desvio no *kernel*.

Para correta execução do algoritmo o *hardware* controla quais *threads* deveriam está executando a cada instante. Para isso ele desativa as *threads* quando a instrução executada não pertencer ao ramo de execução escolhido por ela de forma que cada uma só execute seu próprio caminho. O *hardware* inclui ainda um mecanismo de re-convergência para o programa confluir em uma única linha de execução após os caminhos distintos terem sido completados. Caso não houvesse re-convergência o programa paralelo degeneraria para um caso serial com mais facilidade.

A seguir exibiremos um exemplo de uma aplicação com potencial de ocorrer divergência e mostraremos como ocorre o controle do fluxo de forma que as *threads* só executam as instruções corretas. Nele foram instanciadas cinco *threads* que executaram no mesmo warp. Observa-se na Figura 3.6.4 um *kernel* com uma instrução condicional que pode gerar uma divergência em tempo de execução. Na Figura 3.6.5 é exibido o PTX correspondente ao *kernel* mostrado.

Observando o PTX pode-se extrair algumas de suas características. Ele possui quatro blocos básicos marcados como *bb* 00, *bb* 01, *bb* 02 e *bb* 03. Possuiu também duas instruções de desvio, uma condicional e outro incondicional. Inclui ainda dois labels, L1 e L2, que marcam os alvos dos desvios.

A Figura 3.6.5 mostra ainda uma correspondência entre o código em C e o PTX. Como esperado, algumas instruções de alto nível mapeiam em uma sequência de instruções PTX. Em particular, vemos que a expressão "$if(b < 5)$"é a decisão que determinará, em tempo de execução, a ocorrência ou não de divergência. Em ptx ela aparece como uma sequência de três instruções que gravam o valor do predicado p1 que será usado na decisão de executar ou não a instrução "*bra L*1".

A Figura 3.6.6 mostra o grafo de fluxo de controle. Nele é exibido quais blocos básicos são executados em cada uma das *threads*. Observa-se que as *threads* 0 e 4 compartilham um caminho comum enquanto as *threads* 1, 2 e 3 outro.

Detalhes da divergência são apresentados na Figura 3.6.7, na imagem as instruções estão agrupadas em quatro blocos básicos, as colunas centrais representam

```
__global__ void desvioGPU( int n, int *matA, int *matB, int *matC ) {

        int i = threadIdx.x;

        int a;
        int b;
        int c;

        a = matA[i];
        b = matB[i];

        if(b < 5){
                c = a + b*10;
        }else{
                c = b + a*2;
        }

        matC[i] = c;

}
```

Figura 3.6.4: Código fonte de um *kernel* com potencial para ocorrer divergência de código (Adaptado).

as instruções executadas por cada uma da cinco *threads*. Os espaços em branco no traço das *threads* representam os momentos que elas foram desativadas. A última coluna exibe a máscara utilizadas para controlar a ativação de cada thread, nela 1 representa ativada e 0 desativada.

Acompanhando o fluxo de execução vemos que todas as *threads* executam em paralelo até o pc=88. Neste ponto o predicado p1 é avaliado. As *threads* 0 e 4 seguem a execução pelo bloco básico 01 do pc=96 até o pc=112 enquanto as demais aguardam desativadas. Após a execução do bloco básico 01, o bloco básico 02 é executado invertendo-se a máscara de execução. Do pc=120 até o pc=128 as *threads* 1, 2 e 3 executam o outro ramo do desvio. Finalmente no pc=136 o fluxo de execução re-converge e as *threads* passam a seguir juntas novamente.

3.7 Simulador de GPU

GPGPU-sim é um simulador arquitetural capaz de executar o conjunto de instruções virtuais de *threads* paralelas (PTX). A arquitetura do simulador consiste de uma coleção de pequenos núcleos paralelos chamados shader core. Estes núcleos estão conectados a múltiplos módulos de memória através de uma rede de interconexões.

O shader core é o equivalente arquitetural ao streaming multiprocessor da NVIDIA. Cada shader core é um processador SIMD de largura 8 e possui 24 estágios de pipeline com execução em ordem sem adiantamento de instrução. O pipeline esta dividido em seis estágios lógicos: despacho, decodificação, execução, memória1,

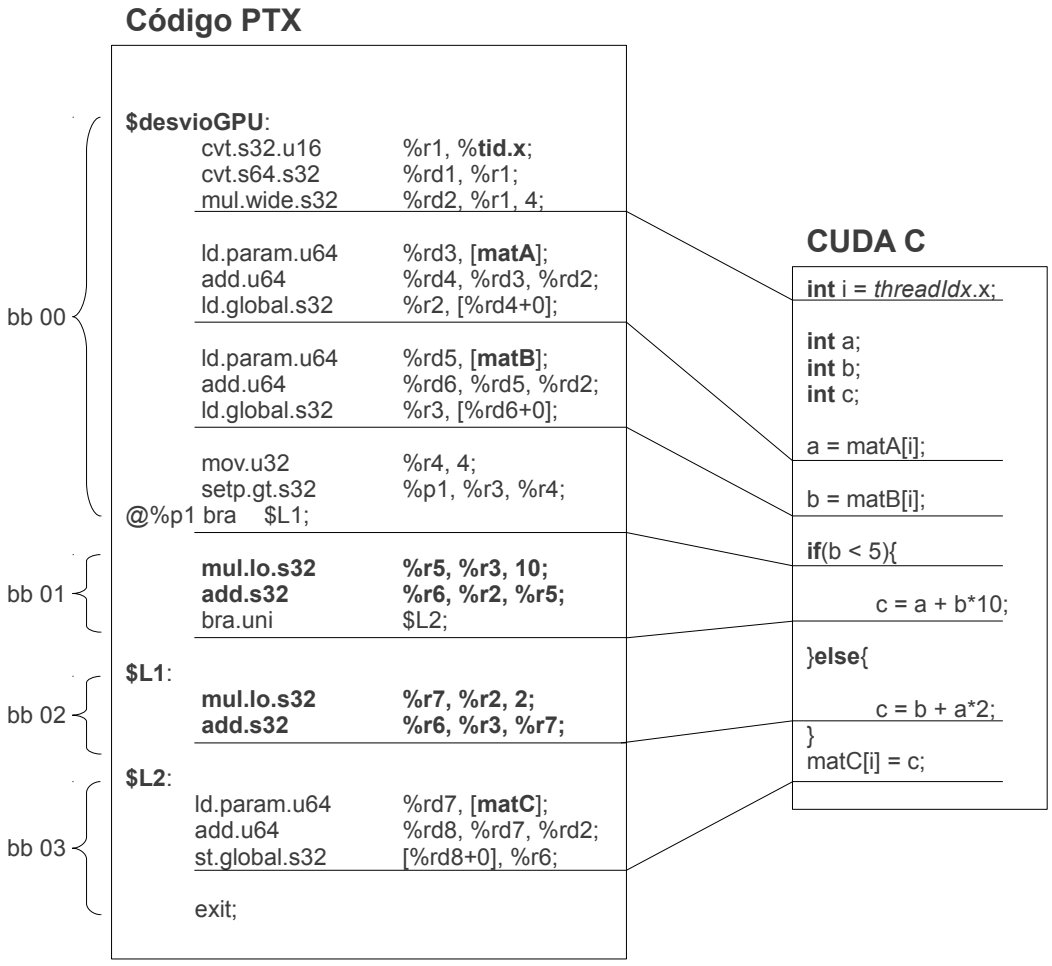


Figura 3.6.5: PTX de um *kernel*, à esquerda, com potencial para ocorrer divergência de código associado ao fonte, à direita, em CUDA C (Adaptado).

memória2, writeback.

As *threads* são distribuídas pelos shader core com granularidade de um CTA inteiro. Os recursos dentro do núcleo apenas são liberados quando todas as *threads* do CTA terminam de executar. Se os recursos permitirem mais de um CTA pode ser associado a um shader core.

As *threads* são escalonadas no pipeline SIMD em um grupo fixo de 32 *threads* chamados warps. Todas elas executam a mesma instrução em diferentes dados por quatro ciclos consecutivos. Como a largura de cada shader core são de 8 unidades funcionais ele precisa de quatro ciclos para poder executar uma instrução nas 32 *threads* do warp.

O escalonamento das *threads* é feito sem nenhuma sobrecarga. A cada quatro ciclos warps prontos para execução entram no pipeline. Warps que possuem *threads*

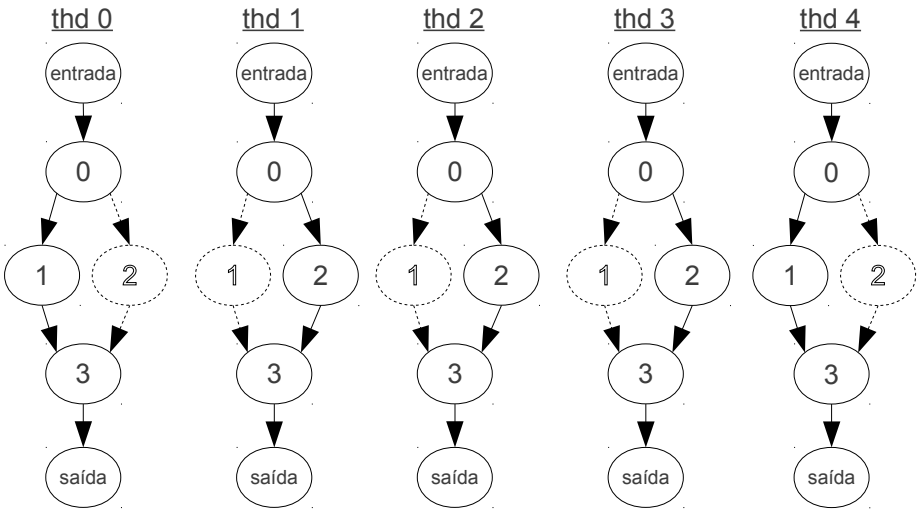


Figura 3.6.6: Grafo de execução ilustrando traço divergente. Cada *thread* segue seu próprio caminho.

bb	PC	thd 0	thd 1	thd 2	thd 3	thd 4	execução
bb 00	0	cvt.s32.u16	cvt.s32.u16	cvt.s32.u16	cvt.s32.u16	cvt.s32.u16	1 1 1 1 1
	8	cvt.s64.s32	cvt.s64.s32	cvt.s64.s32	cvt.s64.s32	cvt.s64.s32	1 1 1 1 1
	16	mul.wide.s32	mul.wide.s32	mul.wide.s32	mul.wide.s32	mul.wide.s32	1 1 1 1 1
	24	ld.param.u64	ld.param.u64	ld.param.u64	ld.param.u64	ld.param.u64	1 1 1 1 1
	32	add.u64	add.u64	add.u64	add.u64	add.u64	1 1 1 1 1
	40	ld.global.s32	ld.global.s32	ld.global.s32	ld.global.s32	ld.global.s32	1 1 1 1 1
	48	ld.param.u64	ld.param.u64	ld.param.u64	ld.param.u64	ld.param.u64	1 1 1 1 1
	56	add.u64	add.u64	add.u64	add.u64	add.u64	1 1 1 1 1
	64	ld.global.s32	ld.global.s32	ld.global.s32	ld.global.s32	ld.global.s32	1 1 1 1 1
	72	mov.u32	mov.u32	mov.u32	mov.u32	mov.u32	1 1 1 1 1
	80	setp.gt.s32	setp.gt.s32	setp.gt.s32	setp.gt.s32	setp.gt.s32	1 1 1 1 1
	88	@%p1 bra	@%p1 bra	@%p1 bra	@%p1 bra	@%p1 bra	1 1 1 1 1
bb 01	96	mul.lo.s32				mul.lo.s32	1 0 0 0 1
	104	add.s32				add.s32	1 0 0 0 1
	112	bra.uni				bra.uni	1 0 0 0 1
bb 02	120		mul.lo.s32	mul.lo.s32	mul.lo.s32		0 1 1 1 0
	128		add.s32	add.s32	add.s32		0 1 1 1 0
bb 03	136	ld.param.u64	ld.param.u64	ld.param.u64	ld.param.u64	ld.param.u64	1 1 1 1 1
	144	add.u64	add.u64	add.u64	add.u64	add.u64	1 1 1 1 1
	152	st.global.s32	st.global.s32	st.global.s32	st.global.s32	st.global.s32	1 1 1 1 1
	160	exit	exit	exit	exit	exit	1 1 1 1 1

Figura 3.6.7: Exemplo de trace divergente.

bloqueadas são ignorados até que estejam prontos. Com esta característica, se existirem warps suficientes a latência de memória pode ser mascarada. Warps prontos são aqueles que possuem todas as suas *threads* em estado de pronto. Em outras palavras muitas *threads* executando no shader core tornam o sistema tolerante a operações de longa latência sem redução de *performance*.

Permitir que mais CTA executem em um shader core aumenta mais a tolerância a operações de longa latência. Em contrapartida também aumenta a demanda pela utilização de recursos. Entretanto, mesmo que exitam recursos suficientes, adicionar

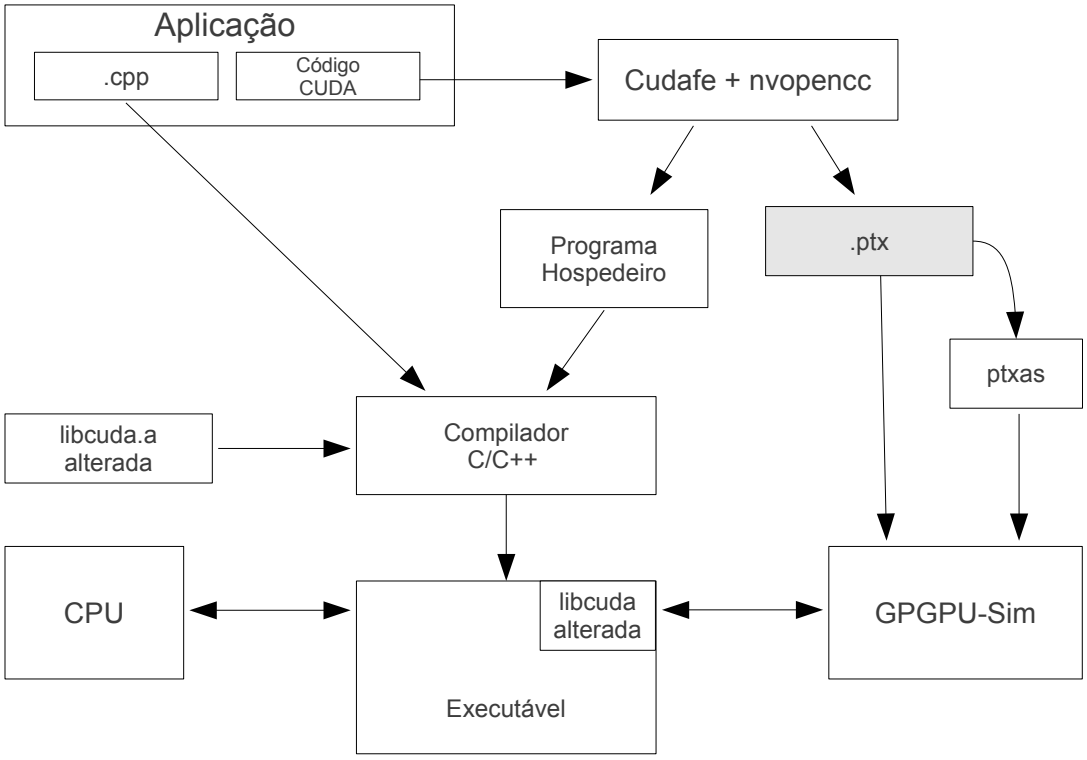


Figura 3.7.1: Fluxo de execução do GPGPU-sim.

threads ao shader core de forma a ocupar completamente os recursos e se a aplicação fizer uso intenso de memória pode causar perda de *performance* pelo aumento da disputa pelas redes de interconexões e controladores de memória [1].

A dificuldade de possuir aplicações que pudessem ser executadas no simulador foi um problema nas primeiras versões. Para isso o simulador foi estendido para ser compatível com a arquitetura CUDA.

Expandir o simulador para que se tornasse compatível com o código CUDA permitiu que inúmeras aplicações já prontas e otimizadas para GPU pudessem ser avaliadas no simulador. Característica que serve de grande fonte de informações para investigações arquiteturais.

A versão atual do simulador consegue executar aplicações CUDA sem modificação no código, porém necessita de acesso ao código fonte da aplicação.

Na Figura 3.7.1 é mostrado como o fluxo de compilação do CUDA foi alterado para permitir a execução no *kernel* no simulador.

As diferenças em relação ao fluxo normal ocorrer a partir do nvcc. O código .ptx gerado segue diretamente para o simulador. O código do programa hospedeiro segue para o compilado C/C++ para ser link-editado. Uma versão alterada do libcuda.a é incluída no programa.

A biblioteca libcuda.a fica responsável por fazer as chamadas na GPU. Ela foi alterada de forma a redirecionar as chamadas que iriam para GPU para o simulador.

3.7.1 Outros Trabalhos em GPU

Encontramos também diversos trabalhos propondo melhorias nas GPUs. Eles englobavam deste uma método para visualização da complexa dinâmica de execução de um programa na GPU, passando por investigações em modos de formação e diferentes tamanhos de warps.

Junto com o GPGPU-sim [6] [7] seu autor apresenta um precioso recurso do simulador que torna possível visualizar diversos aspectos da dinâmica de execução dos programas na GPU. Entre as funcionalidades do simulador em conjunto com o visualizador encontramos uma linha do tempo das estatísticas de execução com informações como uso dos canais de memória e IPC, detalhamento da ocupação do warp, mapeamento entre o código C e o PTX mostrando as partes do código onde o programa mais executou e anotação do código fonte com as estatísticas de *performance*.

Suas contribuições ajudaram a identificar duas importantes características a respeito de programação para GPU que devem ser levadas em consideração quando o objetivo for alcançar a máxima *performance*. São elas:

1. Entre as aplicações estudadas, *performance* é mais sensível a gargalos na rede de interconexões do em relação a latência;

2. Em algumas aplicações executar menos *threads* concorrentemente do que as suportadas pelo processador pode levar a incremento na *performance* pela redução na contenção da memória do sistema.

Capítulo 4

Medindo Reuso de Traços em GPUs

Reuso na GPU trata da técnica de Memorização Dinâmica de traços aplicada a Processadores Gráficos. Chamamos este sistema de DTM@GPU. Nele detalhamos como aplicar o DTM para medir o potencial de reuso presente nas aplicações projetadas para ambientes massivamente *multithread*.

O DTM@GPU possui as seguintes características:

1. Apenas instruções consecutivas no fluxo dinâmico de execução formam traços redundantes;

2. Apenas instruções redundantes de uma mesma *thread* formam traços redundantes, ou seja para serem incluídas no traço a mesma deverá ter sido anteriormente rotulada como *redundante intra-thd*;

3. Uma vez concluída a formação do traço redundante, qualquer *thread* com a mesma MTT em comum poderá reaproveitá-lo independente de qual tenha sido a *thread* que o originou;

4. De forma a prover a criação de traços independente para cada *threads*, cada uma possui seu próprio *buffer* temporário.

Neste capítulo abordaremos os porquês destas características dando informações sobre construção e uso de traços em ambiente *multithread*.

Na primeira seção definimos as alterações arquiteturais necessárias para acomodar o DTM na GPU. Começamos mostrando as alterações no simulador GPGPU-sim e como ele se relaciona com o modelo DTM@GPU que implementamos. Descrevemos ainda as alterações feitas nas tabelas de memorização, a quantidade de tabelas utilizadas e como as distribuímos de forma a não atrapalhar o desempenho da arquitetura.

Na seção seguinte explanamos o processo de formação dos traços redundantes deste de a identificação das instruções redundantes até a finalização da construção do traço redundante. Mostramos como devem ser as tabelas de memorização, seus campos, os rótulos possíveis e como utilizá-los.

Apresentamos também o conceito de *reuso inter-thread*, este novo rótulo para instruções é o principal artifício para proporcionar reuso entre diversas *threads* sem comprometer a computação. Como veremos, esta ideia expande o uso do DTM aumentando seu potencial para além do que cada *thread* poderia manifestar isoladamente.

Por fim, na terceira seção, detalhamos como identificar as oportunidades de reuso e como evitar a formação de traços redundantes falsos. Explicaremos como este novo tipo de armadilha se manifesta e como evitá-la. Terminamos a seção com um exemplo de execução acompanhado passo-a-passo.

4.1 Alterações arquiteturais

A arquitetura estudada neste trabalho possui grandes diferenças em relação a arquitetura substrato usada no DTM. De forma a adequar a técnica aos novos requisitos impostos, precisamos colher o máximo de informações possíveis sobre a plataforma alvo.

Além das diferenças tecnológicas, as GPUs possuem também um paradigma de programação próprio. Neste cenário responsabilidades geralmente atribuídas ao arquiteto de processadores, aos projetistas de compiladores e sistema operacional e aos programadores não necessariamente encontram-se nas mesmas posições que tradicionalmente ocupariam em um sistema feito para CPU.

Por exemplo, nas GPUs a troca de contextos das *threads* migrou da responsabilidade do SO para o processador. Desta forma, o custo normalmente associado a troca de contexto não se aplica mais. Isto permitiu à GPU lidar com grande quantidade de *threads* com um tipo de paralelismo difícil de ser obtido em CPU nas mesmas proporções.

Com tantas diferenças em relação a tradicional computação serial para CPUs e dada a relativa juventude da computação em GPU, algumas de nossas dúvidas geradas no processo de implementação do DTM@GPU não puderam ser respondidas pelas documentação disponível. Por conta disto, executamos testes preliminares na arquitetura alvo e observamos as características do *benchmark* utilizado afim de responde-las.

Nas próximas subseções, indicamos algumas escolhas feitas durante a implementação do DTM@GPU e justificamos sua adoção em função do nosso entendimento a respeito da arquitetura alvo e dos perfis observados nos testes preliminares.

4.1.1 Alterações no GPGPU-sim

Alteramos o simulador GPGPU-sim de forma a registrar o traço dinâmico gerado pela execução do *kernel*. Cada linha do traço possui informações sobre a instrução atualmente executada e o núcleo que a executou. A Tabela 4.1.1 traz informações contidas em uma linha do traço dinâmico de execução junto com seu significado.

laneid	Posição da *thread* no warp, indica qual SP executou a thread;
num1	Contagem de instruções executadas globalmente
uid	Identificador único da *thread*
CTA	Coordenada x,y,z do CTA no Grid
Thread	Coordenada x,y,z da *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

Tabela 4.1.1: Informações contidas no traço dinâmico da GPU, elas são capturadas pelo simulador gpgpu-sim e usadas para alimentar o modelo DTM.

A análise do traço dinâmico de execução pelo DTM@GPU determinará a quantidade de reuso de instruções, o número de traços redundantes criados e o número de traços redundantes re-utilizados.

O fluxo de execução usado neste trabalho para medição do reuso na GPU pode ser conferido na Figura 4.1.1. Nele usamos o GPGPU-sim alterado para gerar o fluxo dinâmico de instrução. Em seguida redirecionamos esta saída para alimentar nosso simulador com o DTM@GPU implementado. Do nosso modelo extraímos as estatísticas e as informações de reuso e execução.

4.1.2 Requisitos arquiteturais para DTM@GPU

Implementamos um simulador separado que recebe como entrada o fluxo de instruções dinâmicas gerado pelo GPGPU-sim modificado. Nesta aplicação modelamos tanto o mecanismo do DTM quanto os elementos arquiteturais da GPU pertinentes a organização das *threads*. Observando a execução deste modelo pudemos entender o impacto do reuso de instruções numa arquitetura massivamente *multithread*.

Para contabilizar a quantidade de instruções redundantes empregamos a tabela MTG, ilustrada na Figura 4.1.2, com seis campos em cada entrada: *pc*, $sv1$, $sv2$, $sv3$, *uid* e *res*. Assim como no DTM original, *pc* refere-se ao contador de programa da instrução, $sv1$, $sv2$ e $sv3$, referem-se aos operandos de origem e *res* representa o resultado da operação.

Uma vez que esta arquitetura de GPU não possui preditor de desvios, os campos

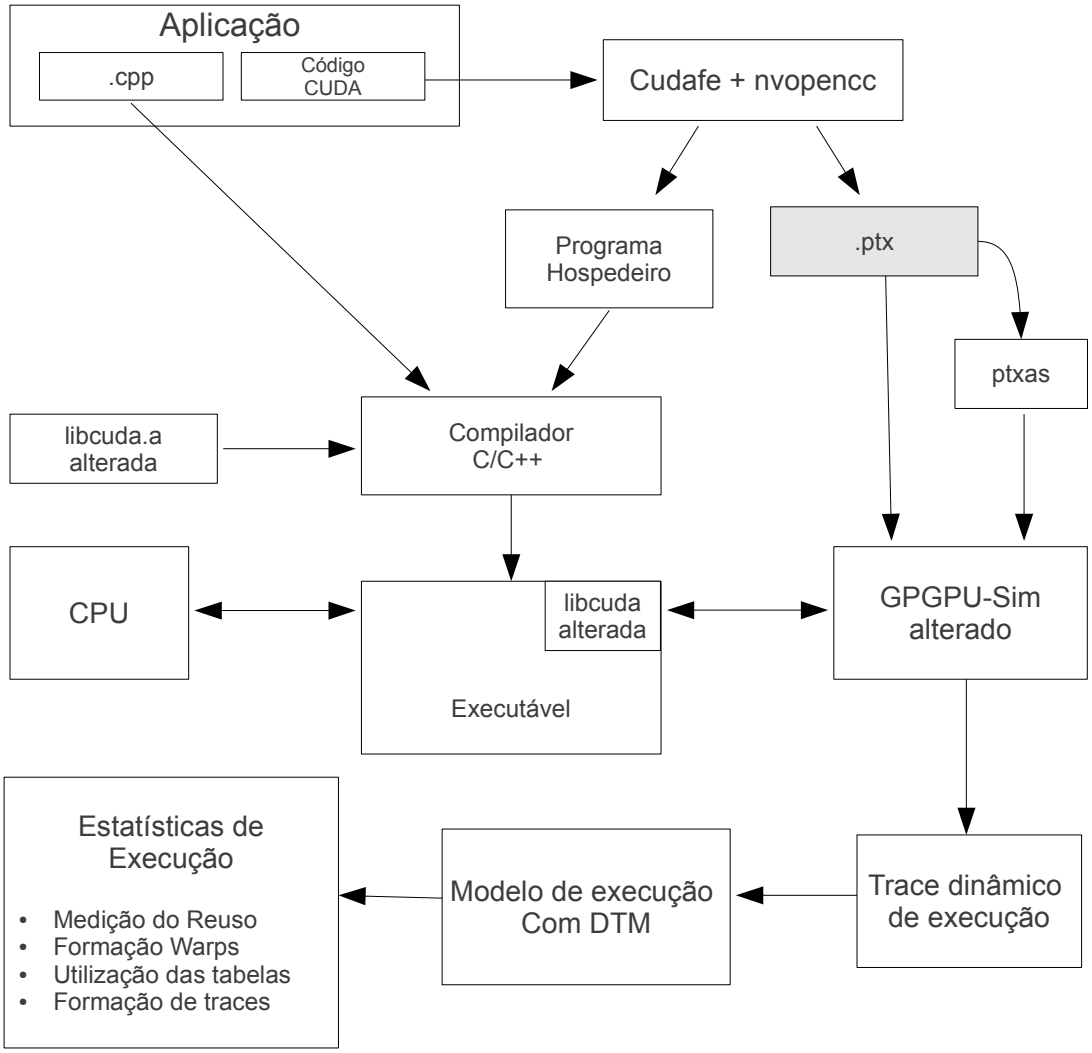


Figura 4.1.1: Fluxo de execução do DTM@GPU.

pc	sv1	sv2	sv3	uid	res

Figura 4.1.2: Campos da tabela MTG.

pc	npc	icr(N)	icv(N)	ocr(N)	ocv(N)

Figura 4.1.3: Campos da tabela MTT.

no DTM original responsáveis por tratar esta unidade foram substituídos por novos campos para contemplar a execução *multithread* e a utilizações de instruções com predicativos. Sendo assim, os campos *jmp*, *brc* e *btaken* deram lugar ao identificador único da *thread* (*uid*) e ao campo *sv*3 que podem contemplar tanto a instruções com três operandos explícitos, por exemplo MAD, quanto instruções com dois operandos e um predicativo.

A principal modificação na MTG foi a adição do campo *uid* objetivando rastrear a *thread* de origem da instrução. Isto permitiu o seu reuso e ao mesmo tempo impediu a formação de traços indevidos na MTT.

Em um ambiente multiprocessado onde diferentes *threads* concorrem pelos recursos a ordem que elas executarão não pode ser determinada, em função disto cuidados adicionais devem ser tomados na formação dos traços. Este processo será abordado em detalhes na subseção 4.2.5.

Nesta implementação necessitamos de entradas para tabela MTT também com seis campos, ilustrados eles na Figura 4.1.3. Os campos *pc* e *npc* representam o contador de programa da instrução atual e da instrução a ser executada após o traço. Os items *icr*, *icv*, *ocr* e *ocv* representam os registradores e os valores dos contextos de entrada e saída respectivamente assim como no DTM original.

Como a estrutura do preditor de desvio não existe na arquitetura abordada dispensamos, em nossa implementação, os campos necessários para atualizá-la. Desta forma, pudemos utilizar uma versão mais enxuta do *buffer* de formação de traços. Dispensamos também o campo *uid* para esta tabela, pois verificamos a possibilidade de reaproveitar traços redundantes oriundos de diferentes *threads*. Mostraremos os detalhes da formação dos traços na subseção 4.2.

4.1.3 Definido campos da MTG

Para definir a quantidade de campos necessários pela MTG precisamos traçar um perfil das instruções usadas pelo PTX. Para isso, além de consultar a documentação do conjunto de instruções tivemos que coletar estatísticas de execução do benchmark.

De acordo com o manual do PTX ISA [30] uma instrução pode ter de zero até quatro operandos além de um predicado opcional utilizado para execução condicional. Em princípio qualquer instrução pode ter sua execução condicionada ao valor

do predicativo associado a ela.

Antes de adotarmos uma tabela com campos suficientes pretendendo cobrir todas as combinações possíveis de instruções com múltiplos operandos e predicativo, nós resolvemos analisar os traços dinâmicos gerados pela execução dos programas de teste[2]. Com isso, pudemos determinar a frequência de instruções executadas agrupadas pela quantidade de operandos.

Observamos também que, dentro das aplicações avaliadas, o compilador utilizou predicativos apenas nas instruções de desvio nas quais eram desejados salto condicional. Nesta contagem estas instruções foram contabilizamos como possuindo um operando de entrada. Incluímos na contagem apenas instruções válidas, pois apenas estas entram na MTG.

	Quantidade de Operandos (%)			
Nome	0	1	2	3
AES	0,00	37,91	62,09	0,00
BFS	16,53	30,29	53,18	0,00
CP	24,43	1,57	74,00	0,00
LPS	23,54	27,25	49,21	0,00
MUM	17,98	38,67	43,36	0,00
NN	11,34	12,70	75,96	0,00
NQU	25,31	35,89	38,81	0,00
RAY	25,93	33,48	40,59	0,00
STO	1,05	18,78	80,13	0,04

Tabela 4.1.2: Distribuição da quantidade de operandos em instruções válidas. A maioria das instruções possuem apenas dois operandos fontes.

Na Tabela 4.1.2 podemos observar uma grande concentração de instruções com dois operandos, seguido por instruções de um e zero operandos. Uma pequena quantidade de instruções de três operandos foi identificada em apenas uma das aplicações avaliadas. Instruções de quatro operandos não apareceram, pois dentro de nossas observações, ocorrem em instruções de textura e estas, como veremos na seção 4.1.4, consideramos inválidas em nosso modelo.

De posse destas informações decidimos por implementar a MTG com três operandos apesar da ocorrência quase nula deste último tipo. Desta forma, cobrimos todas as instruções presentes nas aplicações testadas e ainda assim conseguimos economizar dois campos que seriam necessários à MTG caso levássemos em conta apenas o documentado nos manuais em vez de averiguar na prática como o ISA estava sendo utilizado.

[2]Os programas de testes utilizados englobam várias áreas do conhecimento, entre elas criptografia (AES), busca em largura (BFS), simulação física (CP), aplicação financeira (LPS), sequenciamento de DNA (MUM), reconhecimento de escrita (NN), N rainhas (NQU), Ray Tracing (RAY) e MD5 (STO). Detalhes sobre estas aplicações serão apresentados em 5.1.2

4.1.4 Escolhendo as instruções válidas

Diferenciamos as instruções entre válidas e inválidas classificando-as de acordo com a viabilidade de pertencerem ao DTM@GPU, pois nem todas as instruções podem ser reusadas.

As instruções válidas submetem-se ao mecanismo de reuso podendo ser reaproveitadas de acordo com avaliação subsequente, enquanto as instruções inválidas nunca são reusadas.

Como o reuso utiliza tabelas de memorização as informações nela armazenadas devem permitir buscas pelas instruções e seus parâmetros de entrada e retornar os valores de saída previamente armazenados. Por conta disto apenas instruções cujas saídas são exclusivamente determinadas em função das entradas fazem parte do grupo das válidas.

Convém identificarmos primeiro os conjuntos das inválidas, pois instruções válidas aparecem em maior quantidade que as inválidas. O conjunto de instruções válidas resulta então do complemento do conjunto de instruções inválidas no universo de instruções existentes.

Tomando como referência as instruções descritas como inválidas no DTM original, ampliamos a ideia base aplicando-a de forma a identificar aquelas não condizentes com os requisitos de validade na nossa arquitetura alvo. Com isto definimos os seguintes grupos de instruções inválidas:

1. **Instruções de sincronização** de *thread* como *bar* e *exit*. Utilizadas para forçar a convergência das *threads*. Elas definem barreiras onde as *threads* devem esperar as outras *threads* do CTA antes de prosseguir com a execução. No caso da instrução *exit* marca o termino da execução da *thread*. O reuso deste tipo de instrução poderia levar a uma perda do ponto de sincronização levando algumas *threads* a acessarem posições de memória antes do tempo correto;

2. **Instruções de acesso a memória** como *ld* e *st*. Assim como no DTM as instruções de load e store não entram devido aos custos associados à manutenção das tabelas de memorização com valores instanciados na memória;

3. **Instruções de textura e superfície** como *tex*, *txq*, *tld*4, *suld*, *sust*, *sured* e *suq* sofrem dos mesmos problemas das instruções de acesso a memória e por isso também não serão incluídas no DTM@GPU.

4. **Instruções de ponto flutuante** como as marcadas com os modificadores $f16$, $f32$ e $f64$. Também não entram no DTM@GPU pelos mesmos motivos de não entrarem no DTM, ou seja, a baixa localidade dos valores de ponto flutuante inviabiliza a implementação da memorização destes valores.

Para determinar se existem instruções suficientes potencialmente redundantes, resolvemos realizar a contagem das frequências dos grupos de instruções nos fluxos dinâmicos de execução. Desta forma, obtivemos uma visão do percentual de instruções candidatas ao reuso.

	Válidas (%)			Inválidas (%)			
	Inteiros	Desvio	Total	Memória	PF e Tex	Sincronismo	Total
AES	66,91	0,00	66,91	27,29	2,66	3,14	33,09
BFS	59,63	11,81	71,44	25,19	0,00	3,37	28,56
CP	16,25	5,25	21,50	5,28	73,19	0,03	78,50
LPS	50,36	15,51	65,87	3,21	27,78	3,15	34,14
MUM	64,70	14,18	78,88	3,51	17,54	0,07	21,12
NN	51,96	6,64	58,60	2,27	39,04	0,08	41,39
NQU	53,65	18,18	71,83	10,62	0,00	17,56	28,18
RAY	25,10	8,79	33,89	1,85	64,05	0,20	66,10
STO	89,04	0,82	89,86	10,10	0,00	0,04	10,14

Tabela 4.1.3: Frequência das instruções dinâmicas agrupadas por tipo. Sete dos nove programas testes apresentam mais de 58% de instruções validas.

Na Tabela 4.1.3 exibimos os resultado obtido neste teste preliminar. Como a maioria dos acessos a textura retornaram valores de ponto flutuante agrupamos estas colunas. Ela nos mostra o grande potencial de reuso uma vez que grande fração das instruções dinâmicas são válidas.

Sete dos noves programas apresentam mais de 58% das instruções válidas. Os dois restantes, apesar de apresentarem mais instruções de acesso à textura ou memória, ainda contam com mais de 20% de instruções válidas.

4.1.5 Distribuição das tabelas de memorização

Diferente do mecanismo DTM original onde existe apenas um par de tabelas e um fluxo de execução, no DTM@GPU existe um par de tabelas por SP e centenas de fluxos de execução paralelos distribuídos pelos núcleos.

As *threads* além de salvarem seu contexto possuem também um *buffer* temporário utilizado para construção do traço antes dele ser adicionado à MTT. Mostramos na Figura 4.1.4 a localização da MTG, MTT e do *buffer* temporário.

O mesmo algoritmo DTM@GPU executa em cada um dos SP. As várias *threads* que compartilham um SP trocam informações por intermédio das tabelas de memorização. O conjunto de suas interações resulta no reuso de instruções no sistema paralelo. Este algoritmo permite, sempre que possível, aproveitar instruções já executadas em fluxos diferentes.

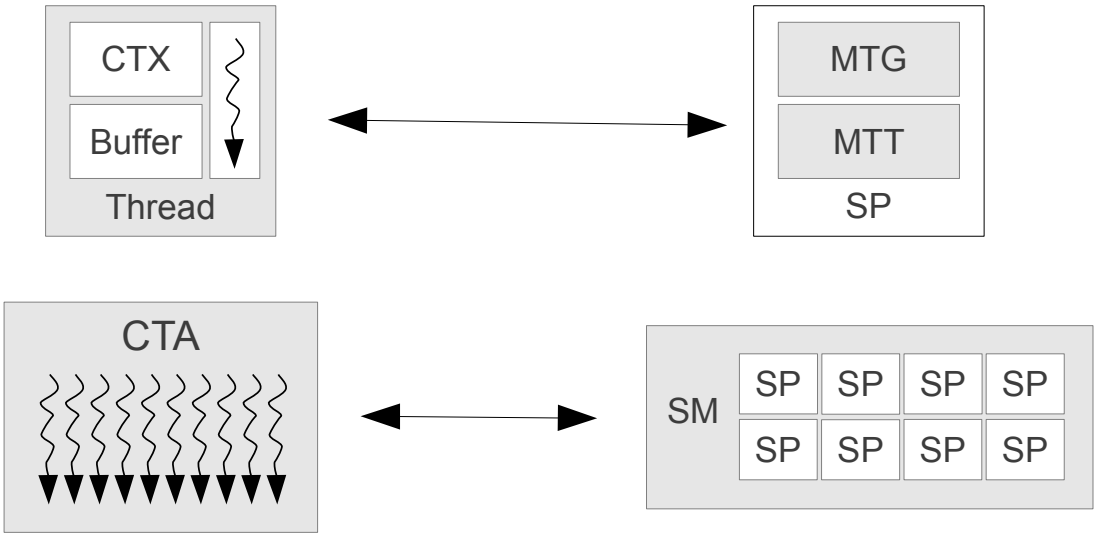


Figura 4.1.4: Localização do MTG, MTT e Buffer no DTM@GPU.

A quantidade de tabelas de memorização em uma GPU com DTM@GPU pode ser calculada multiplicando-se a largura do multiprocessador pela quantidade deles na GPU. Por exemplo, a quantidade de MTT de uma GPU com 10 SM de largura 32 é de 320 unidades.

A maneira como as tabelas de memorização foram distribuídas estão limitadas pela quantidade de recursos consumidos por elas e pelo impacto negativo que sua inclusão indiscriminada pode causar.

Não pudemos usar um único par de tabelas para toda a GPU pois isto implicaria na necessidade de sincronismo ao acesso delas. Todas a *threads* disputando as tabelas centralizadas poderia ocasionar serialização dos acessos reduzindo o desempenho.

No extremo oposto, se cada *thread* tivesse seu próprio par de tabelas também não seria viável. Por um lado a implementação seria muito facilitada, pois assim cada fluxo de execução funcionaria como se fosse um DTM serial independe. Por outro teria dois pontos negativos que nos fez descartar esta possibilidade.

Uma desvantagem estaria no consumo exagerado de recursos exigidos, pela grande quantidade de tabelas necessárias, a segunda desvantagem eliminaria a possibilidade de reuso entre *threads* anulando o potencial do DTM@GPU. Por isso, esta alternativa também foi considerada ineficiente.

Encontramos o meio termo explorando as próprias características de execução do ambiente massivamente *multithread*. Nos valemos no fato dos warps executarem concorrentemente e inserimos as tabelas no SP.

Desta forma, várias *threads* concorrentes compartilham as mesmas tabelas enquanto as *threads* que executam simultaneamente, em SPs distintos, possuem suas próprias tabelas.

Com isso, não precisamos serializar o acesso aos recursos e nem isolarmos completamente cada *thread* permitindo aproveitar o melhor dos dois cenários.

A necessidade de usar um *buffer* temporário para cada *threads* surgiu quando começamos analisar a formação de traços no sistema paralelo. Originalmente cada MTT possuia apenas um *buffer*. Quando notamos que o acesso concorrente estava levando à construção de traços diferentes do esperado resolvemos mudar a localização do *buffer* fazendo com que cada *thread* construísse seu traço privativamente.

Esta abordagem resolveu o problema dos traços construídos errados sem impedir que os traços prontos pudessem ser aproveitados por todas as *threads* que compartilham o mesmo MTT.

4.2 Construção de traços redundantes

Esta seção detalhará os passos necessários para a construção de traços no DTM@GPU. Como visto antes eles são formados por instruções dinâmicas redundantes executadas consecutivamente.

Começamos mostrando como identificar os diferentes tipos de instruções redundantes neste ambiente. Para isso definiremos três rótulos, *não redundante*, *redundante intra-thread* e *redundante inter-thread* e explicaremos a maneira como utilizá-los para classificar as instruções de forma a construir os traços adequadamente.

Daremos em seguida detalhes sobre a formação de traços em sistemas *multithread*. Falaremos sobre como exploramos suas particularidades no sentido de aproveitar a redundância disponível.

Por fim, descreveremos como a redundância em nível de instrução é utilizada para formação de traços indicando também como atualizar os contextos de entrada e saída do *buffer* temporário.

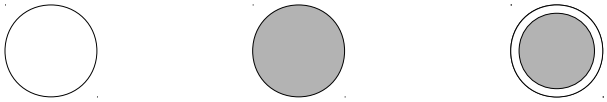
4.2.1 Identificando instruções redundantes

No estudo sobre como aplicar o DTM a sistemas com múltiplos fluxos de execução, identificamos a necessidade de rever a classificação dada as instruções por entender que agora mais de um tipo de redundância poderia ocorrer.

Cobrimos esta exigência pela disponibilização de três rótulos, mostrados na Figura 4.2.1, para classificá-las: *não redundante*, *redundante intra-thread* e *redundante inter-thread*.

O primeiro possui um nome auto explicativo, usamos nos casos de instrução inválida ou nos casos onde a instrução ainda não foi adicionada na MTG.

Nos casos de instruções reexecutadas dentro de um mesmo fluxo de execução aplicamos o rótulo *redundante intra-thread* para marca-las. Utilizamos este rótulo



(a) (b) (c)

Figura 4.2.1: Rótulos das instruções. (a) não redundante, (b) redundante intra-thd, (c) redundante inter-thd

Instrução válida	Presente na MTG	Operandos 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

Tabela 4.2.1: Classificação das instruções para criação da MTG no DTM@GPU.

para identificar aquelas que deverão entrar no traço.

Por último o rótulo *redundante inter-thread* marca instruções reexecutadas em outros fluxos de execução, proporcionando ao algoritmo a oportunidade de reusá-la mesmo sendo a primeira instância dela no fluxo corrente.

A Tabela 4.2.1 reúne as rotulações possíveis e as condições necessárias para cada um. A escolha por qual usar depende de consultar a MTG e avaliar a instrução de acordo com o seguinte algoritmo:

1 Para cada instrução decodificada: avaliar se esta pertence ao conjunto de instruções válidas. No caso de uma instrução inválida, esta recebe o rótulo *não redundante*;

2 Para cada instrução pertencente ao conjunto de válidas: pesquisar na tabela MTG entradas com as chaves *pc*, $sv1$, $sv2$, $sv3$;

2.1 Caso não exista na MTG uma entrada correspondente a chave pesquisada então o mecanismo marca a instrução corrente como *não redundante* e insere a instrução na tabela;

2.2 Caso exista pelo menos uma entrada na MTG com a chave pesquisada avalia-se o campo *uid*;

2.2.1 Caso uma destas entrada possua o campo *uid* igual ao campo *uid* da instrução corrente significa que esta entrada foi adicionada previamente à MTG por esta mesma *thread* e por conta disto trata-se de um *reuso intra-thread*. Nesta situação nenhuma instrução é adicionada à MTG.

2.2.2 Caso contrário, a pesar da instrução ser redundante, trata-se da primeira ocorrência desta instância nesta thread. Desta maneira marca-se ela como *redundante inter-thread*. Neste caso, a instrução deve ser adicionada à MTG.

4.2.2 Entendendo reuso inter-thread

Vimos anteriormente que uma das principais características das GPUs está na quantidade de *threads* instanciadas durante a execução. Um arcabouço sólido proporciona a organização necessária para o gerenciamento delas e obtenção de alta *performance*.

Aplicamos o DTM no cerne da arquitetura onde warps concorrendo pelo uso dos vetores de processamento passam a ter algumas de suas *threads* liberadas da execução pelo reuso.

O identificador único (uid) presente em cada instrução permite rastrear as execuções habilitando o reuso onde não houver risco de por o processador em um estado inconsistente. Como veremos na subseção 4.2.5 a diferenciação entre *reuso inter-thread* e *reuso intra-thread* é crucial para correta formação de traços neste ambiente.

Dentro do warp as *threads* ocupam uma posição fixa denominada *faixa de execução* (lane). Este posicionamento determina qual SP irá executá-la quando o warp que a contém for escalonado. Por conseguinte, esta associação determina quais são as tabelas de memorização utilizadas pela thread.

Somado a isto, na arquitetura da GPU, vários warps intercalam na utilização dos núcleos fazendo com que diferentes *threads* executem, concorrentemente, na mesma *faixa de execução*. Como existe um par de tabelas de memorização para cada SP, podemos interpretar como se cada *faixa de execução* tivesse seu próprio par de tabelas. Desta forma, mais de um *thread* podem compartilha-las.

Este compartilhamento permite as *threads* trocarem informações de redundância ampliando as possibilidades para além de seu próprio escopo. A rotulação diferenciada faz-se necessária para evitar as armadilhas inerentes aos sistemas paralelos e manter o processamento consistente.

Na Figura 4.2.2 podemos observar um CTA com 18 *threads* alocado em um SM com 6 SPs. Nela podemos ver as seis faixas de execução associadas aos SPs. Podemos ver ainda que sempre o mesmo conjunto de thread, uma de cada warp, executam em uma determinada faixa. Com isso, temos as *threads* 01, 07 e 13 na faixa 1, as *threads* 02, 08 e 14 na faixa 2, e assim sucessivamente.

Neste exemplo hipotético arbitramos o tamanho do warp em 6 *threads*[1] para facilitar a visualização da imagem. Por conta disto as 18 *threads* são divididas em 3 warps que se alternam para execução.

O mecanismo DTM@GPU é capaz de compartilhar as tabelas de memorização entre *threads* que executam na mesma *faixa de execução* sem perder a consistência do estado do seu contexto.

No exemplo da Figura 4.2.2, as *threads* 06, 12 e 18 executam suas instruções com os mesmos operandos fontes. Na ordem arbitrada a primeira *thread* executada foi a *thread* 06, depois a *thread* 12, seguido da *thread* 18.

Quando a *thread* 06 executa, atualiza-se a MTG com as informações da instrução. As *threads* seguintes na mesma *faixa de execução*, por terem os mesmos operandos fontes, reaproveitam a computação memorizada. Estas instruções também entram na MTG mas recebem o rótulo de *redundante inter-thread*.

Desta forma, mesmo sendo a primeira ocorrência delas tanto na *thread* 12 quanto na *thread* 18, elas puderam ser reutilizadas, pois a *thread* 06 já tinha executado a mesma anteriormente.

[1] O tamanho do warp é uma decisão arquitetural e não esta disponível para ser alterada pelo programador. Nas simulações executadas neste trabalho utilizamos o tamanho padrão de 32 *threads* por warp.

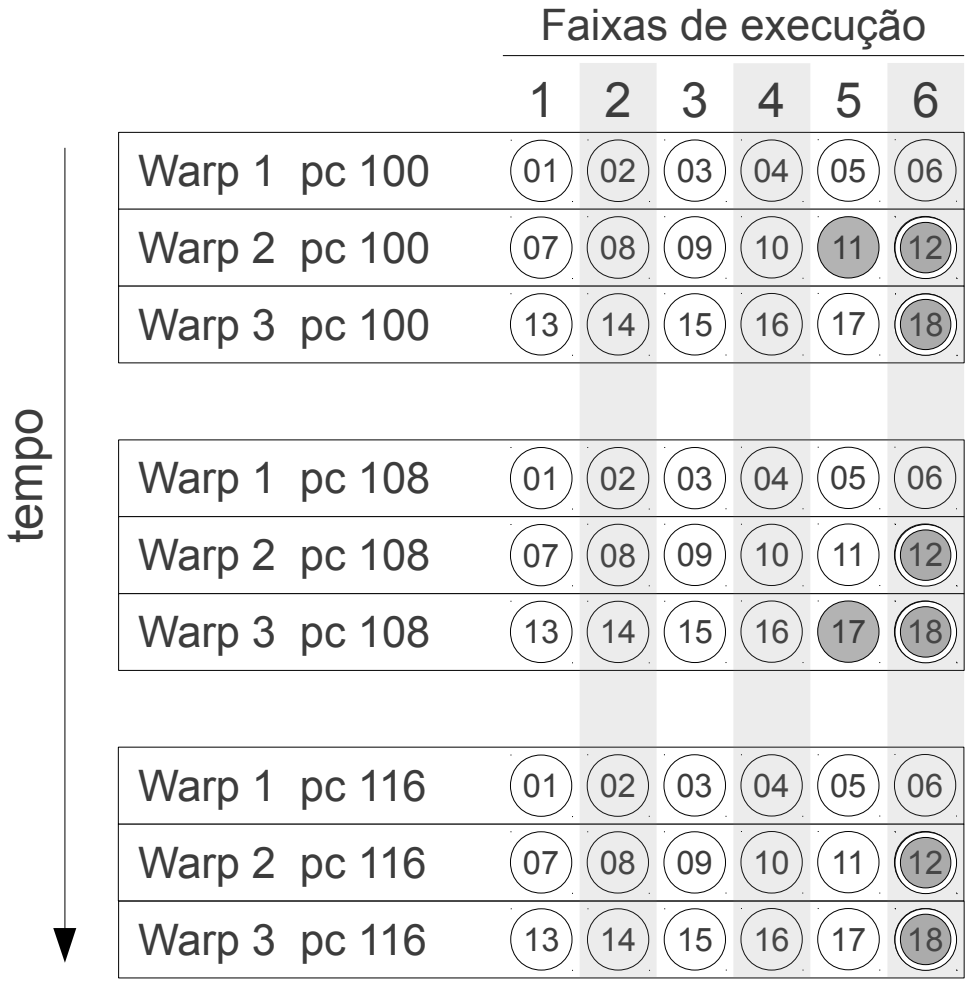


Figura 4.2.2: Exemplo ilustrando as faixas de execução do Stream Multiprocessor. Esta imagem mostra um CTA possuindo 18 *threads* sendo dividido em Warps de largura 6. Cada SP fica responsável por uma faixa. Neste caso existem 3 *threads* por faixa.

4.2.3 Adicionando instruções redundantes aos traços em formação

Um traço redundante é formado exclusivamente por instruções rotuladas como redundante intra-thread consecutivas no fluxo de execução. Instruções não redundantes ou redundantes inter-thread servem para delimitar os traços formados.

Mostramos na Figura 4.2.3 cinco combinações possíveis de rotulação de instruções em pontos de interesse durante execução de um programa. Nela marcamos também como seriam as formações de traços em cada um dos casos.

Em 4.2.3a temos a formação de um traço de tamanho cinco indicado pela seta preta, em 4.2.3b e 4.2.3c temos outros dois traços em cada mostrando como instruções *não redundantes* e *redundantes inter-thread* podem delimita-los e nas imagens 4.2.3d e 4.2.3e não ocorre a formação de nenhum traço redundante.

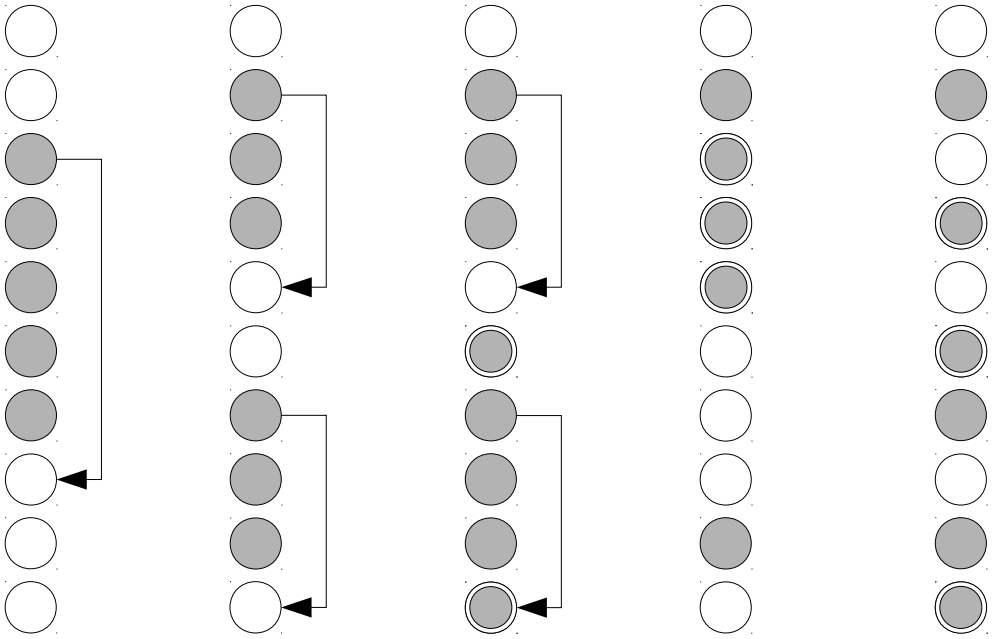


Figura 4.2.3: Exemplo de como as instruções podem ser rotuladas e situações onde traços podem ou não ser formados. (a) Traço de tamanho cinco, (b) dois traços de tamanho três, (c) dois traços de tamanho três, o primeiro terminado em instrução não redundante e o segundo terminado em instrução redundante inter-thread, (d) e (e) nenhum traço formado.

Assim como no DTM, avalia-se o rótulo da instrução durante a fase de decodificação. Em nossa implementação levamos em consideração os novos rótulos quando definimos, na Tabela 4.2.2, as ações tomadas para cada situação.

Rótulo	Existe *buffer*?	Ação
não redundante	não	nenhuma ação
não redundante	sim	encerra construção do traço, adiciona *buffer* ao mtt e libera o *buffer*
reuso inter-thd	não	nenhuma ação
reuso inter-thd	sim	encerra construção do traço, adiciona *buffer* ao mtt e libera o *buffer*
reuso intra-thd	não	cria *buffer*
reuso intra-thd	sim	atualiza contexto de entrada e saída do *buffer*

Tabela 4.2.2: Algoritmo para criação da MTT no DTM@GPU.

Durante a inclusão de instruções na construção do traço utiliza-se um *buffer* temporário para reter as informações de reuso. Para cada uma, seus operandos fontes e destino são analisados e adicionado ao contexto de entrada e saída de acordo

com o algoritmo apresentado na subseção 4.2.4.

Na ocasião de encerramento da construção preenche-se o campo *npc* do *buffer*, antes de incluir o mesmo no MTT, com o valor do *pc* da instrução que determinou o encerramento. Assim no momento da reutilização o fluxo de execução pode ser redirecionado direto para este ponto.

4.2.4 Atualizando contextos de entrada e saída

As instruções adicionadas ao traço fornecem as informações necessárias para o preenchimento dos contextos de entrada e saída. Elas são usadas para atualizar o *buffer* temporário apropriadamente seguindo os seguintes passos:

1 Para cada instrução *redundantes intra-thd* pertencente a um conjunto consecutivos de instruções no fluxo dinâmico de execução;

  1.1 Incluir no contexto de entrada do *buffer* os registradores fonte com seus respectivos valores instanciados se estes não foram incluídos no contexto de saída;

  1.2 Incluir no contexto de saída do *buffer* os registradores de destinos e seus respectivos valores instanciados.

Em outras palavras, adiciona-se ao contexto de entrada todos os registradores e valores que foram alterados fora do traço em construção e adiciona-se ao contexto de saída todos os registradores e valores alterados dentro dele.

Estas informações são suficientes para atualizar o processador para um estado idêntico ao que ficaria caso tivesse executado as instruções contidas dentro do traço.

4.2.5 Evitando traços falsos

Uma instrução redundante dentro do DTM@GPU só pode receber dois rótulos distintos: *redundante intra-thd* ou *redundante inter-thread*. Mostraremos nesta subseção por que apenas instruções *redundante intra-thread* formam traços redundantes e explicaremos como evitar traços falsos.

Se resgatarmos o significado do rótulo *redundante intra-thread* veremos que seu uso ocorre quando:

(a) Uma instrução prestes a executar, emitida e com seus operandos carregados, tem o registrador *pc* e operandos fontes identificados dentro da MTG;

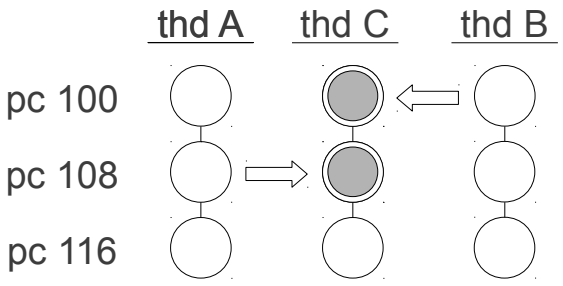
(b) O campo *uid* indicar que a ocorrência anterior aconteceu dentro da própria thread.

pc	código	thd A	thd B	thd C
100	add r4, r5, r6	5, 3, 2	6, 4, 2	6, 4, 2
108	add r3, r1, r2	9, 4, 5	8, 3, 5	9, 4, 5
116	add r7, r8, r9	7, 5, 2	8, 6, 2	8, 5, 3

(a)

pc	sv1	sv2	sv3	res	uid
100	3	2	-	5	A
100	4	2	-	6	B
100	4	2	-	6	C
108	4	5	-	9	A
108	3	5	-	8	B
108	4	5	-	9	C
116	5	2	-	7	A
116	6	2	-	8	B
116	5	3	-	8	C

(b) MTG



(c)

Figura 4.2.4: Exemplo de traço falso.

Este comportamento, de reusar redundância dentro do mesmo fluxo de execução, é precisamente o mesmo indicado no DTM aplicado em um processador serial. Neste caso a formação de traços acontece tal como demonstrado na tese que o apresenta [2].

Em contraste, nos ambientes *multithread*, devido ao chaveamento de contexto necessário para o núcleo atender várias *threads* concorrentemente, pode ser identificado a ocorrência de instruções redundantes cuja instância original pertença a outro fluxo de execução. Este caso denominamos *reuso inter-thread*.

Estas ocorrências não podem formar traços apesar de poderem ser reusadas isoladamente, obtendo o resultado da operação a partir da MTG, dispensando sua reexecução.

Por exemplo, imagine duas instruções como as apresentadas na Figura 4.2.4a com $pc = 100$ e $pc = 108$. Elas aparecem em sequência no código apesar de não possuírem dependência de dados entre elas.

Temos ainda três *threads* pertencentes a warps distintos que ocupam a mesma *faixa de execução*. Ou seja, as três compartilham as mesmas tabelas de memorização.

Como elas executam concorrentemente, estipulamos a ordem executada neste exemplo: Primeiro a *thread* A, depois a *thread* B, seguido da *thread* C. Mantivemos esta sequência para os três *pc* consecutivos 100, 108 e 116.

Sabendo que todas a *threads* na GPU rodam o mesmo programa, sempre que a instrução no $pc = 100$ for executada com operandos fontes $r5 = 4$ e $r6 = 2$ o resultado será 6 e sempre que a instrução no $pc = 108$ for executada com operandos fontes $r1 = 4$ e $r2 = 5$ o resultado será 9. Isto independente da *thread* executora da operação.

Logo, se houver uma entrada na MTG 4.2.4b para estes valores no *pc* com os estes mesmos valores nos campos fontes ela será reusada independente da *thread* que originou a entrada na tabela.

Entretanto, mesmo uma instância de instrução sendo reaproveitada em várias *threads*, não podemos garantir que elas formam traços com as instruções redundantes vizinhas.

Como exibido na Figura 4.2.4c, pode ocorrer uma situação onde uma determinada *thread* consiga reusar instruções vindas de várias outras *threads* distintas gerando uma sequência de instruções redundantes com aparência de traço, porém inédita.

Neste exemplo isto ocorreu com a *thread* C reusando as instruções $pc = 100$ e $pc = 108$ vindas da *thread* B e da *thread* A respectivamente. Chamamos esta sequência formada por instruções redundantes incluindo as *redundantes inter-thread* de traço falso.

Sua construção deve ser impedida, pois apesar de todas as instruções que o compõem serem redundantes a sequência em si não é garantida de já ter ocorrido anteriormente. Para isto, deve-se barrar as instruções *redundantes inter-thread* do processo de formação dos traços. Do contrario a utilização de traços falsos poderá comprometer a consistência do estado do contexto da *thread* inutilizando toda a computação realizada.

4.3 Reusando traços redundantes

Nesta seção explicamos como identificar e reusar traços redundantes no DTM@GPU. O reuso de traços permite, de uma única vez, reaproveitar toda a computação realizada previamente por ele.

Durante a execução do programa o DTM@GPU busca constantemente oportunidades de reutilizar os traços redundantes memorizados. Através da análise do

contexto de entrada ele é capaz de determinar quando utilizar as informações guardadas na MTT.

Apesar da formação de traços depender de instruções redundantes de uma mesma thread, seu reuso não. Todas as *threads* que compartilham a mesma MTT podem utilizar qualquer um dos traços redundantes memorizados.

Esta troca de redundância entre os fluxos de execução permite, dentro de algumas *threads*, reaproveitar traços precocemente, pois nelas dispensariam a fase de criação já realizada por uma *thread* anterior.

Isto ocorre porque na GPU todas as *threads* executam o mesmo programa. Assim as primeiras a executarem tendem a formar a maioria dos traços redundantes permitindo que as demais comecem a reutilizar mais cedo.

A identificação de oportunidade de reuso de traços deve seguir os seguintes passos:

1 Para cada instrução despachada para execução pesquisar a MTT em busca de ocorrências de item que possuam o campo *pc* igual ao *pc* da instrução atual;

1.1 Se nenhuma entrada for encontrada na MTT com o pc fornecido, então não existe traço redundante associado com esta instrução;

1.2 Se existe um item na MTT cujo *pc* corresponde ao *pc* da instrução corrente, então estes itens deverão ser selecionadas;

1.2.1 Para cada um dos itens selecionadas no passo anterior, comparar seu contexto de entrada com o contexto da *thread* atual;

1.2.1.1 Se existir um item cujo contexto de entrada é idêntico ao contexto da *thread* então este item é um traço redundante e será reusado. Para isto, seu contexto de saída será usado para atualizar o contexto de *thread* corrente e o *pc* dela apontará para o *npc* do traço redundante;

1.2.1.2 Se nenhum contexto coincidir, então nenhum traço redundante foi identificado;

1.3 Se o pc não for encontrado, então nenhum traço redundante foi identificado.

Na ocorrência de reuso o fluxo de execução é transferido da instrução atual para o endereço apontado no campo *npc* e o estado do processador é atualizado com as informações do contexto de saída do traço.

A *thread* quando consegue reusar passa para um estado futuro, como se tivesse executado todas as instruções contidas no traço.

4.4 Reuso de Warps

Quando ocorre divergência de código durante a execução, ambos caminhos executam serialmente degradando o desempenho. Ao escolher um caminho para executar, as *threads* do outro fluxo permanecem desativadas até que ele chegue ao final. Nesta ocasião suas *threads* outrora ativas tornam-se desativadas, dando a oportunidade do outro caminho executar.

Quando reusamos instruções agimos da mesma forma. Alteramos a ativação das *threads* para indicar que elas não precisam executar. Com o reuso de instruções esperamos diminuir a quantidade de *thread* divergentes pela desativação tanto do caminho principal quanto do alternativo. Esperamos também reduzir e a quantidade total de warps executados.

Em ambos os casos, pode ocorrer uma situação onde um caminho tenha todas suas *threads* desativadas pelo DTM e em consequência dispensado da execução. Nestas condições, ganhamos desempenho pela diminuição das penalidades oriundas da divergência de código e pela liberação de recursos causado por um warp não executado.

Capítulo 5

Experimentos, Resultados e Análise

5.1 Base Experimental

5.1.1 Ambiente e parâmetros de simulação

Através de modificação no simulador GPGPU-sim foi possível coletar informações sobre a execução das aplicações CUDA e testar e avaliar o grau de redundância presente nelas. Aplicando-se o DTM medimos o quantitativo de instruções que se repetem, a proporção que a repetição representa do total de instruções executadas e o tamanho dos traces identificados.

Na Tabela 5.1.1 listamos os parâmetros arquiteturais utilizados no DTM. Os tamanhos do contexto de entrada, saída, a quantidade máxima de desvios e o número máximo de instruções contidos no traço não foram limitados, pois fizeram parte dos objetos de investigação.

Parametros do DTM	Valores
Heurística	Repetição de instruções redundantes
Instruções válidas	Aritméticas, lógicas, desvios
Politica de substituição	LRU

Tabela 5.1.1: Parâmetros arquiteturais do DTM.

Na simulação realizada escolhemos o tamanho do vetor de SPs igual ao tamanho do warp.

Para medir o reuso total do sistema precisamos olhar para utilização tanto da MTG quanto da MTT. Lidamos com duas situações distintas para fazer as medições:

1. Contagem dos acertos na MTG, o que caracteriza reuso 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 não precisando nem serem emitidas.

5.1.2 Programas de Teste

Para os testes foram utilizados o mesmo conjunto de aplicações propostos pelo autores do simulador. Utilizando um conjunto de programas teste conhecido assim os resultados aqui apresentados poderão ser comparados com os resultados de outras propostas de melhorias existentes ou futuras.

As aplicações que o compõe não possuíram paralelização trivial visto que portalas para plataforma CUDA exigiu grande esforço de seus respectivos autores. Em decorrência disso estas aplicações sofreram intensa otimização. Desta forma ganhos apresentados aqui representam melhorias em programas que atualmente possuem pouca margem para aperfeiçoamento por software.

O conjunto de programas teste é composto pelas seguintes aplicações:

AES Cryptography (AES) [36]: Implementação do algoritmo de criptografia Advanced Encryption Standard em CUDA. Ela foi otimizada para fazer uso da memória de constantes e de textura. No teste foi criptografado um arquivo de 256KB com com criptografia de 128 bits.

Breadth First Search(BFS) [37]: Busca em largura em um grafo onde cada nó é mapeado em uma thread. Desta forma o paralelismo escala com com o tamanho da entrada. Neste teste foi efetuada uma busca em largura em um grafo aleatório de 65.536 nós.

Coulombic Potential(CP) [38]: Simulação física integrante do "Parboil Benckmack suit". Também faz uso da memória constante em suas otimizações. Foram simulados 200 átomos em uma 'grid' de tamanho 256x256.

3D Laplace Solver (LPS) [39]: Aplicação financeira paralela otimizada para uso de memória compartilhada e acessos a memória agrupados (Coalesced).

MUMmerGPU (MUM) [40]: Ferramenta de sequenciamento de DNA. Aplicação foi otimizada para utilizar memória de textura. Nos testes foram utilizado os primeiros 140,000 caracteres genoma do Bacilo anthracis str. Ames.

Neural Network (NN) [41]: Rede neural para reconhecimento de escrita.

Para o teste foram reconhecidos 28 dígitos do "Modified National Institute of Standards Technology database of handwrite digits".

N-Queens Solver (NQU) [42]: Problema das N rainhas no Xadrez. Consiste em tentar colocar N rainhas em um tabuleiro NxN sem que nenhuma ataque a outra. Foi utilizado N=10 nos testes.

Ray Tracing (RAY) [43]: Renderização de imagem foto-realística. Foi executado 5 níveis de reflexão com sombra para renderizar uma imagem de 256x256.

StoreGPU(STO) [44]: Implementação do algoritmo MD5 com um arquivo de entrada de 192KB. Esta aplicação também foi otimizada para fazer uso da memória compartilhada e evitar tráfego fora da GPU.

A Tabela 5.1.2 mostra a quantidade de *threads* e instruções executadas por cada programa de teste.

Benchmark		
Nome	Threads	Instruções
AES	65792	28M
BFS	65536	17M
CP	32768	126M
LPS	12800	82M
MUM	50000	77M
NN	35000	68M
NQU	21408	2M
RAY	65536	71M
STO	49152	134M

Tabela 5.1.2: Total de *threads* e instruções por benchmark

Mais detalhes sobre as aplicações utilizadas no *benchmark* podem ser encontradas no artigo do [1].

5.2 Resultados e Análise

5.2.1 Métricas

Nesta subsecção definimos os itens registrados durante a simulação ou calculados diretamente a partir deles.

O grupo a seguir contém as medidas referentes diretamente ao reuso de instruções.

intra: Quantidade de instruções reusadas dentro da própria thread;

inter: Quantidade de instruções reusadas entre *threads* diferentes;

trace: Quantidade de instruções reaproveitadas pelo reuso de traços;

ir: Quantidade total de instruções reusadas. Inclui as intra, inter e traço;

itot: Quantidade total de instruções executadas;

O próximo grupo contém as medidas relacionadas a ocupação dos warps e aceleração do desempenho.

$w0$: Quantidades de ciclos que o warp deixou de executar por dependência estrutural;

$w0_{dtm}$: Quantidades de ciclos que o warp deixou de executar por causa do DTM. Em outras palavras, warps reusados;

$w1..31$ e $w1..31_{dtm}$: Quantidade de ciclos que warps incompletos foram escalonados. Em testes com DTM desativado e ativado respectivamente;

$w1..32$ e $w1..32_{dtm}$: Quantidade de ciclos que warps com pelo menos uma *thread* ativa foram escalonados. Em testes com DTM desativado e ativado respectivamente;

$w32$ e $w32_{dtm}$: Quantidade de ciclos que warps completos foram escalonados, ou seja, warps com 32 *thread* ativas. Em testes com DTM desativado e ativado respectivamente;

$ciclos_{sem}$: Quantidade de ciclos executados sem utilização do DTM;

$ciclos_{dtm}$: Quantidade de ciclos executados utilizando-se o DTM.

Na Tabela 5.2.1 mostramos as equações dos cálculos utilizados durante análise dos resultados.

5.2.2 Percentual de Reuso

Medimos o percentual de reuso calculando a razão entre o total de instruções redundantes e o total de instruções executadas. O total de instruções redundantes incluem os reusos intra-thread, inter-thread e o reuso por reaproveitamento de traços. Calculamos também a média harmônica entre todos os programas de testes.

Apresentamos na Figura 5.2.1 os níveis de reuso obtidos neste experimento. A média harmônica cresceu de 13,6% para 35,3% utilizando tabelas com tamanho variando de 16 até 8192 entradas.

Nome	Fórmula
Percentual de Reuso	$ir = intra + inter + trace$ $ir/itot$
Warps Incompletos	$w1..31 = w1..32 - w32$
Aceleração de Desempenho	$ciclos_{dtm} = ciclos_{sem} - w0_{dtm}$ $speedup = ciclos_{sem}/ciclos_{dtm}$
Média Aritmética	$AM = n^{-1} * \sum_{i=1}^{n} x_i$
Média Harmônica	$HM = n * (\sum_{i=1}^{n} \frac{1}{x_i})^{-1}$

Tabela 5.2.1: Cálculos realizados

A maior variação no reuso ocorreu quando mudamos o tamanho das tabelas de 16 para 32 entradas. Obtivemos um aumento de 5% no reuso neste caso, entretanto encontramos melhores patamares quando utilizamos tabelas maiores.

Notamos uma estabilização nos níveis de reuso no patamar de 30,7%. Neste platô obtivemos a melhor diferença quando variamos o tamanho das tabelas de 4096 para 8192 resultando num reuso 1,83% mais alto.

Entre os programas testes chamou-nos a atenção o programa STO quando as tabelas mudaram de 4096 para 8192. Nesta transição medimos um aumento de 12% no reuso. Este teste foi, com folga, o maior reuso medido. Ele atingiu a marca de 87,35% de instruções reusadas.

Outros programas como AES, CP, LPS, NN, NQU, RAY e o próprio STO também apresentaram resultados interessantes. Estes testes demonstraram patamares bem característicos no nível de instruções reusadas. Notamos estabilidade próximo aos valores 44%, 21%, 50%, 12%, 63%, 31%, 74% de reuso respectivamente.

Eles sugerem que para várias classes de aplicações, mesmo um número reduzido de entradas nas tabelas de memorização, podem ser capazes de sustentar um bom nível de reuso. Nestes casos um aumento incremental no tamanho das tabelas trariam poucos benefícios.

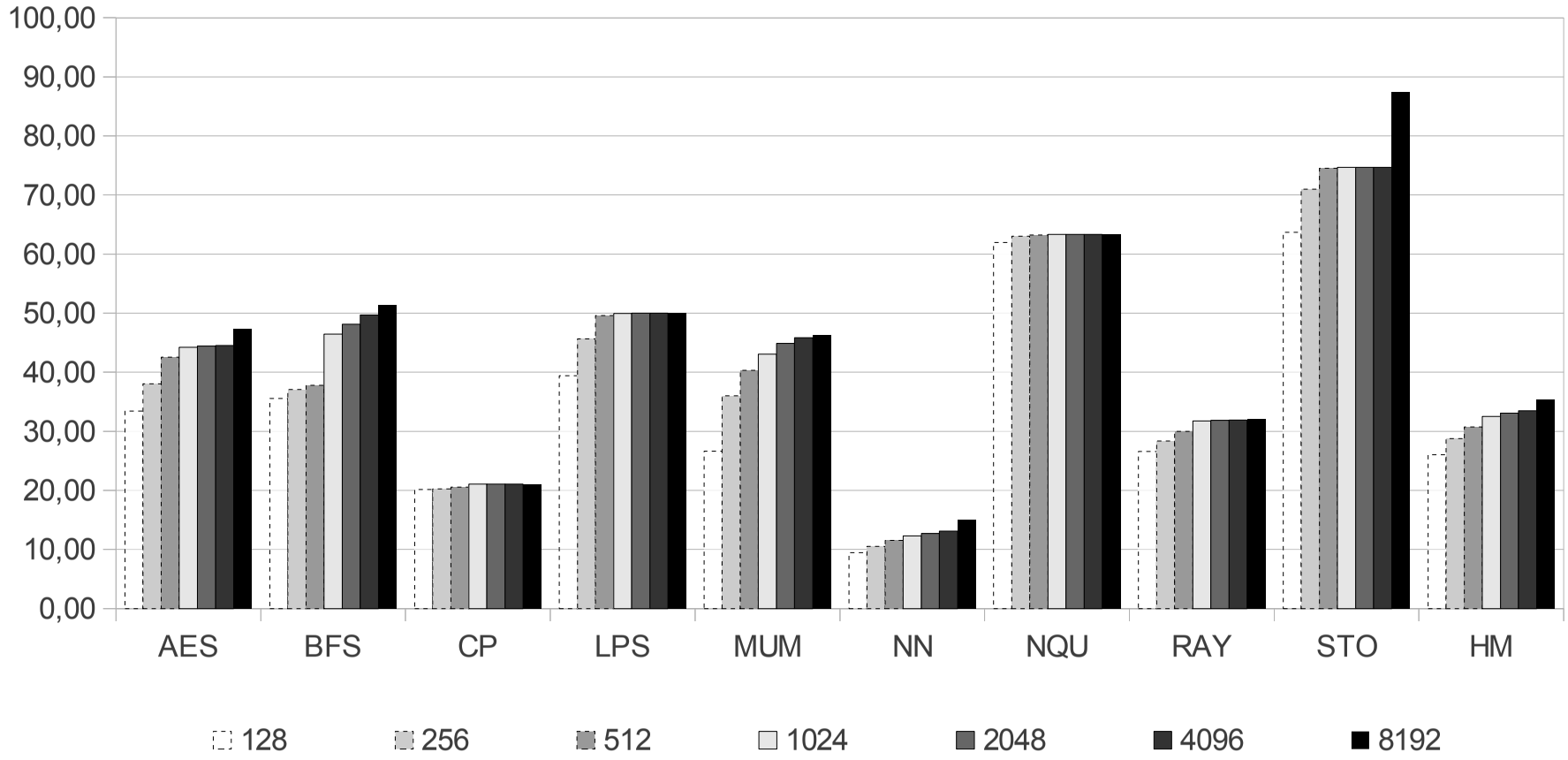


Figura 5.2.1: Variação do percentual de reuso no DTM@GPU. A média harmônica aumentou de 13,6% (16) para 35,3% (8192) variando a quantidade de entradas na tabelas de memorização. O nível de reuso estabiliza-se no patamar de 30,7% (512), a partir deste ponto experimenta aumentos incrementais com a variação das tabelas.

5.2.3 Estimativa da Aceleração de Desempenho

Estimamos a aceleração de desempenho calculando a razão entre a quantidade de ciclos executados sem utilização do DTM($ciclos_{sem}$) e a quantidade de ciclos executados em testes aplicando o DTM($ciclos_{dtm}$). Para calcular $ciclos_{dtm}$ contamos a quantidade de ciclos onde warps deixaram de executar devido possuírem todas suas *threads* desativadas($w0_{dtm}$) e subtraímos de $ciclos_{sem}$.

Esta medida de aceleração representa uma estimativa, pois não leva em consideração os impactos que a suspensão da execução dos warps pode ocasionar ao pipeline. Analisamos os dados obtidos durante a execução e inferimos a provável aceleração de desempenho com base no padrão de reuso observado.

Como as instruções reusadas dispensam reexecução, as *threads* que a contém podem ser desabilitada. O conjunto de *thread* desabilitadas no warp somam-se as desabilitadas pelo controle de fluxo do SIMT. A quantidade de ciclos economizados corresponde ao total de ciclos em que warps deixaram de executar por não possuírem *threads* que necessitassem de execução naquele momento.

Em nossos cálculos, a quantidade de ciclos necessárias para acessar a memória ou os gastos com bolhas no pipeline permaneceram constantes. Desta forma consideramos nossa abordagem conservadora, pois ela refere-se exclusivamente aos warps que deixaram de executar devido ao reuso das instruções em suas *threads*.

	AES	BFS	CP	LPS	MUM	NN	NQU	RAY	STO	HM
16	8,69	1,07	5,80	21,95	0,19	5,64	5,22	8,94	25,18	1,26
32	9,98	1,82	10,69	27,36	0,63	5,75	8,70	10,81	27,08	3,22
64	11,02	2,60	12,49	32,72	1,48	5,84	9,89	12,07	29,36	5,45
128	12,33	3,16	12,64	40,47	2,23	5,90	11,31	13,39	33,39	6,85
256	13,78	3,48	12,73	52,24	3,62	5,96	13,28	14,99	38,69	8,42
512	15,44	3,68	12,96	62,72	4,66	6,03	13,61	16,40	41,46	9,29
1024	16,12	4,07	13,30	66,74	5,32	6,11	13,70	18,36	41,55	9,99
2048	16,14	4,24	13,30	66,90	5,84	6,16	13,70	18,52	41,55	10,32
4096	16,16	4,39	13,30	66,96	6,16	6,18	13,70	18,56	41,55	10,53
8192	16,94	4,47	13,30	66,96	6,35	6,23	13,70	18,66	52,30	10,76

Tabela 5.2.2: Variação da estimativa de aceleração de desempenho no DTM@GPU.

Na Tabela 5.2.2 apresentamos em detalhes os percentuais de aceleração obtidos variando-se as tabelas de memorização de 16 até 8192 entradas. Na Figura 5.2.2 exibimos uma representação gráfica da tabela considerando tabelas a partir de 128 entradas.

A média harmônica da aceleração do desempenho variou entre 1,26% e 10,76% em testes com tabelas de 16 até 8192 entradas. O maior aumento ocorreu na transição de tabelas com 64 para 128 tabelas. Neste ponto medimos incremento de 1,57% na

aceleração.

A partir do uso de tabelas com 1024 entradas notamos estabilização num patamar entorno de 10% de aceleração. Aumentos adicionais ao tamanho das tabelas causaram pouco impacto no desempenho medido.

Analisando individualmente os programas de teste nos surpreendeu uma aparente desconexão, em alguns testes, entre os níveis de reuso e a aceleração calculada.

Programas como BFS, MUM e NQU obtiveram baixa aceleração por nível de reuso medido. Apesar de possuírem percentual de reuso máximo em 50%, 45% e 60% obtiveram aceleração de apenas 4%, 5% e 13% respectivamente. Enquanto a aceleração de programas como CP, LPS, NN, RAY e STO acompanharam os níveis de reuso. Deste grupo o LPS destacou-se por possuir a maior aceleração. Com tabelas de apenas 1024 entradas já apresentava estimativa de aceleração próximas a 70%.

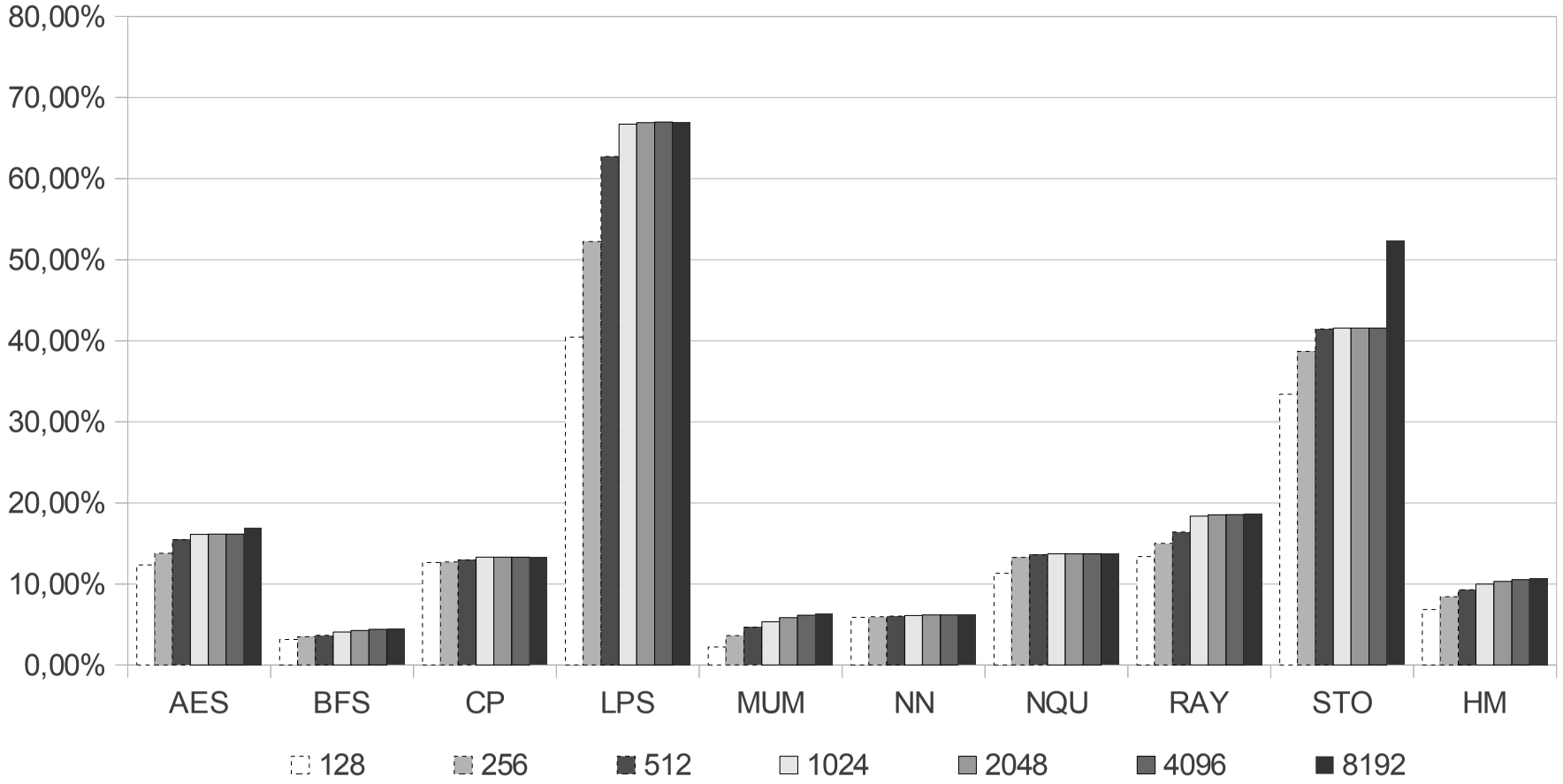


Figura 5.2.2: Variação da estimativa de aceleração de desempenho no DTM@GPU. A média harmônica variou 1,26%(16) até 10,76%(8192) estabilizando no patamar de 10% a partir de testes com 1024 entradas nas tabelas de memorização. Aumentos adicionais causaram pouco impacto.

5.2.4 Variação na distribuição da ocupação do warp

Conforme esperávamos o reuso permitiu uma diminuição da quantidade de warps executados. Percebemos redução tanto nos warps completos($w32$) quanto nos warps incompletos($w1..31$).

A Figura 5.2.3 mostra o percentual de ocupação do warp em relação aos ciclos de execução com warps ativos, ou seja, no cálculo desconsideramos os ciclos onde nenhum warp pôde executar por dependência estrutural ou por latência de memória.

Já analisamos a quantidade de ciclos com warps sem *threads* ativas($w0_{dtm}$) na subseção anterior, ela corresponde à aceleração estimada e cresceu junto com o aumento nos tamanhos das tabelas.

Nas colunas superiores podemos observar a quantidade de ciclos com warps completos($w32_{dtm}$). Como esperado esta medida diminuiu com o aumento do tamanho das tabelas, pois o reuso permitiu desabilitar *threads* transformando uma fração dos warps completos em warps incompletos.

Em tabelas com 16 entradas os warps completos corresponderam a 42% dos ciclos de execução ao passo que com 8192 entradas esta fração limitou-se a 28% dos ciclos. Identificamos as maiores variação na transição da tabelas com 16 para 32 entradas e tabelas com 64 para 128 entradas. Elas diminuíram 3,3% e 2,9% respectivamente.

O grupo de warps incompletos por um lado cresce devido aos warps completos tendo suas *threads* desativadas mas por outro diminui, pois também pela desativação das *threads*, existe a possibilidade de todas elas serem desativadas.

Devido causarem baixa ocupação das unidades funcionais, devemos minimizar a quantidade de warps incompletos. Sua ocorrência impacta na eficiência do processador.

No modelo de programação adotado para esta arquitetura cabe ao programador evitar em seu código construções que levem a divergência de código e em última análise aos warps incompletos.

Existem estudos ativos [7] visando retirar um pouco desta responsabilidade do programador. Eles propõem um *hardware* para reagrupamento dos warps de forma a maximizar a ocupação dos núcleos.

Acreditamos que esta técnica possa complementar a aplicação do DTM@GPU no sentido de diminuir a fração de warps incompletos. Apresentaremos nos trabalhos futuros uma proposta para reagrupar as *threads* e converter estes warps incompletos em ganhos adicionais.

5.2.5 Quantidade média de traços reusados

Analisamos como a quantidade média de traços variou com diferentes tamanhos de tabelas de memorização. Na Figura 5.2.4 observamos um primeiro aumento

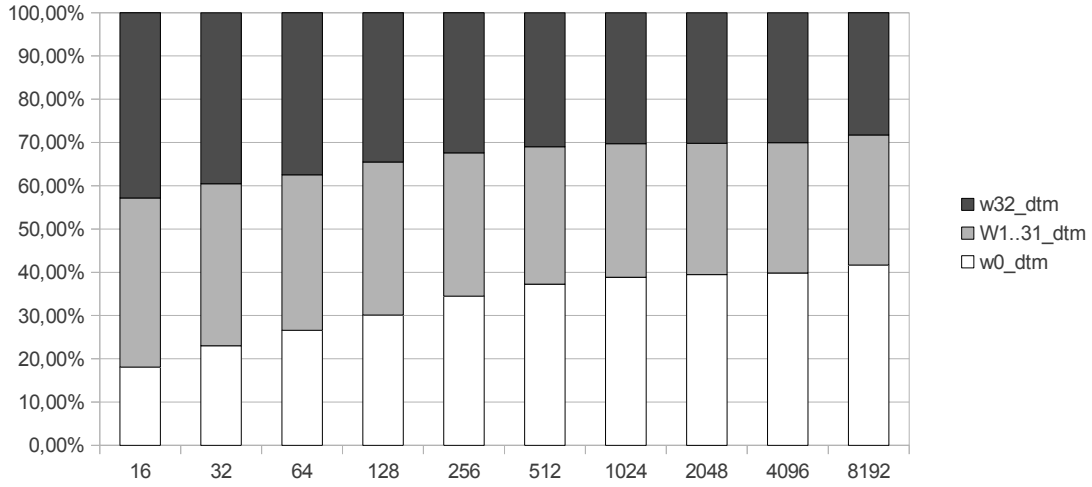


Figura 5.2.3: Variação na distribuição da ocupação do warp. A quantidade de ciclos com warps completos($w32_{dtm}$), ciclos com warps incompletos($w1..31_{dtm}$) e warps não executados devido ao DTM ($w0_{dtm}$) e suas distribuições para diferentes tamanhos de tabelas de memorização.

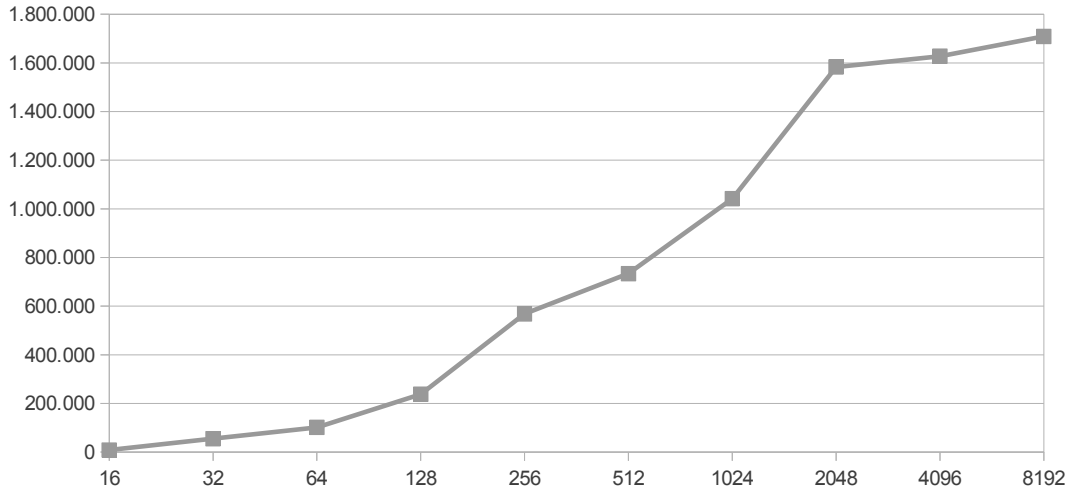


Figura 5.2.4: Média aritmética da quantidade de traços reusados variando no tamanho das tabelas de memorização.

expressivo no número de traços quando aumentamos o tamanho das tabelas de 128 para 256 entradas.

O aumento mais relevante ocorre nos testes com tabelas maiores. Aqueles realizados com tabelas de 2048 entradas geraram 31% mais traços redundantes comparados aos testes com 1024 entradas. Aumentos adicionais no tamanho das tabelas para 4096 e 8192 entradas representaram apenas 2,5% e 4,7% no aumento da quantidade de traços respectivamente.

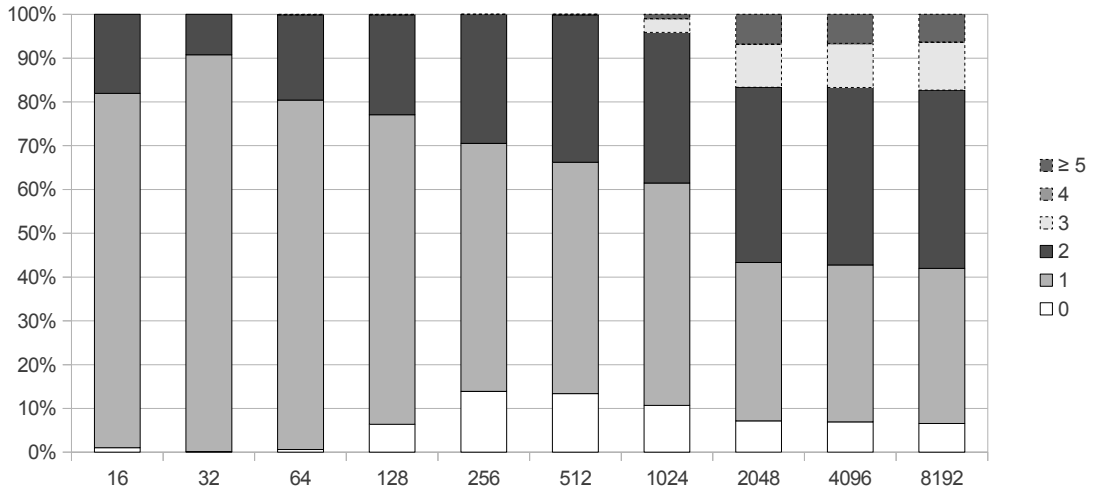


Figura 5.2.5: Distribuição média percentual do número de elementos no contexto de entrada dos traços reusados. Nota-se que até 512 entradas nas tabelas de memorização apenas dois elementos são necessários no contexto de entrada.

5.2.6 Distribuição de elementos nos contexto de entrada e saída

Nos testes realizados não limitamos o tamanho dos contextos de entrada nem de saída. Com trata-se da primeira vez que o potencial de reuso de uma GPU está sendo analisado optamos por não impor barreiras artificiais durante a formação dos traços. Desta forma pudemos observar até que ponto eles podem chegar nesta arquitetura.

Observando a distribuição média percentual do número de elementos no contexto de entrada, apresentado na Figura 5.2.5, notamos que ampla maioria dos traços reusados utilizaram até 2 elementos no contexto de entrada. Eles totalizam 82% dos traços reusados em testes com 8192 entradas nas tabela de memorização.

Se incluirmos traços com 3 entradas alcançamos a marca de 93% de traços reusados. Traços com 4 entradas não contribuíram expressivamente e os traços com 5 ou mais elementos no contexto de entrada somam 6,31% do total de traços reusados.

Em testes realizados com as tabelas de memorização com 512 entradas praticamente 100% dos traços gerados tiveram 2 ou menos itens do contexto de entrada. Em testes com 1024 entradas este percentual foi de 96%. Contextos de tamanho 2 foram os que mais cresceram em participação com o aumento do tamanho das tabelas.

Procedemos avaliação análoga ao observar o contexto de saída dos traços apresentado na Figura 5.2.6. A partir de tabelas com 2048 entradas 89% dos traços reusados possuem até 4 elementos no contexto de saída. A maioria deles(60%)

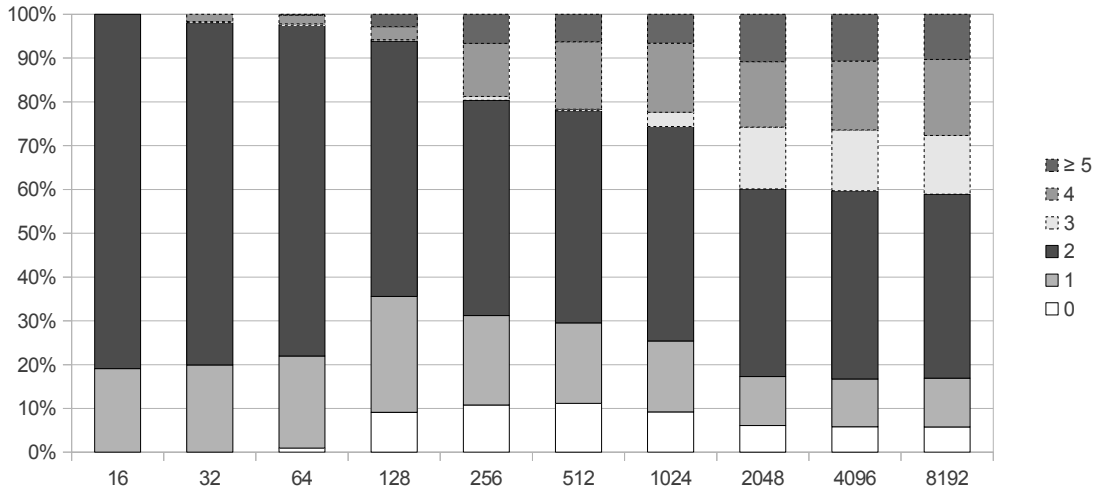


Figura 5.2.6: Distribuição média percentual do número de elementos no contexto de saída dos traços reusados. Traços como até quatro elementos representam 89% do total. Este valor foi de 100%, 93% e 89% para tabelas de memorização com 16, 256 e 4096 entradas respectivamente.

possuem até 2 elementos.

Notamos variação maior na composição dos contextos de saída a medida que aumentávamos os tamanhos das tabelas de memorização. A frequência de ocorrência de traços com mais de 4 elementos no contexto de saída aumentou significativamente. Ela partiu de literalmente nenhuma contribuição até o total de 28% dos traços reusados. Enquanto isto a participação de traços com 2 itens neste contexto diminuiu dos seus 80% para 42%.

A participação de traços como até quatro elementos no contexto de saída estabilizou em 89% dos traços reusados. Este valor foi de 100%, 93% e 89% para tabelas de memorização com 16, 256 e 4096 entradas respectivamente.

5.2.7 Análise do tamanho dos traços reusados

Medimos também o tamanho médio dos traços reusados. Observamos que a maioria dos traços possuem 4 ou menos instruções. Houve um aumento nos tamanhos dos traços na medida que o número de entradas nas tabelas aumentaram.

Podemos notar na Figura 5.2.7 que mais de 60% dos traços possuem 3 ou menos instruções e mais de 70% possuem até 4 instruções, para tabelas com 2048, 4096 e 8192 entradas.

Se definirmos este parâmetro do DTM limitando o tamanhos dos traços em 4 instruções conseguiríamos contemplar pelo menos 80% dos traços com tabelas de 16, 32, 64, 128 e 256 entradas. Para tabelas maiores que isto este limite contemplaria

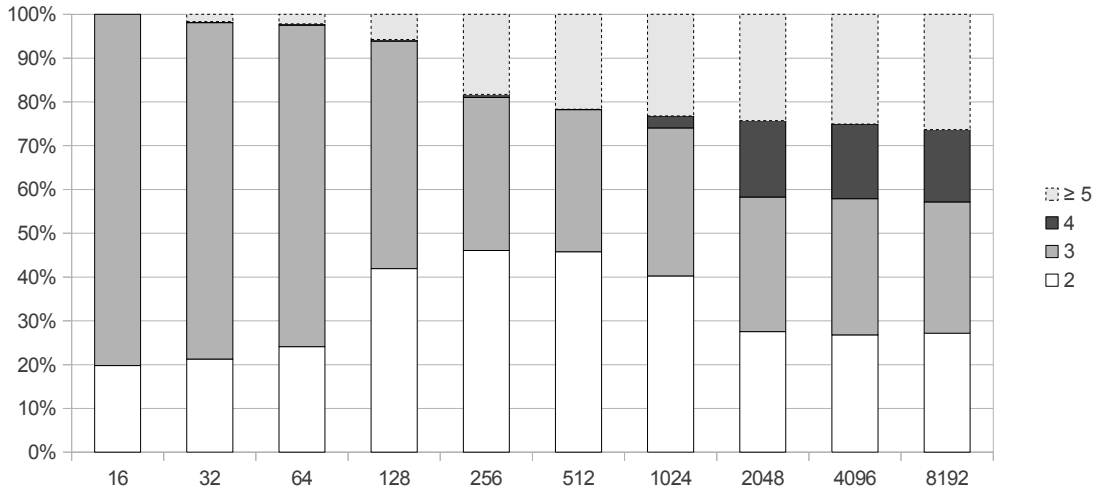


Figura 5.2.7: Distribuição média percentual do número de instruções pertencentes aos traços reusados. Mais de 60% dos traços possuem até 3 e mais de 70% possuem até 4 instruções, para tabelas com 2048, 4096 e 8192 entradas.

pelo menos 60% dos traços reusados.

5.2.8 Análise dos desvios incluídos nos traços reusados

Variando-se a quantidade de entradas nas tabelas de memorização o percentual de traços não contendo nenhum desvio aumentou de 0,7% para 20%. Os traços contendo 1 desvio diminuiu dos 100% para próximo de 50%. Aqueles com 2 desvios atingiu o máximo de 15%.

Traços com 3 ou 4 desvios sofreram pouca alteração variando entorno de 2% e 4% como podemos observar na Figura 5.2.8. Enquanto traços com 5 desvios surgiram repentinamente no patamar de 6,6% a partir de testes com tabelas de 2048 entradas.

Dois patamares podem ser identificados, um em testes com 256 entradas e outro com 2048 entradas nas tabelas de memorização. Em testes com 256, traços com até 2, 3 e 4 desvios ocuparam 93%, 95% e 100% do total de traços reusados respectivamente. Já nos testes com 2048 entradas, traços com até 2, 3 e 4 desvios ocuparam 87%, 91% e 93% do total respectivamente.

5.2.9 Detalhando percentual de Reuso

Nesta subseção analisamos o nível de reuso medido separado por tipo. As Tabelas A.1.1, A.1.2, A.1.3, A.1.4, A.1.5, A.1.6, A.1.7, A.1.8, A.1.9 mostram os resultados do reuso nos programas de testes.

Nelas podemos observar em detalhes o nível de reuso *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redun-*

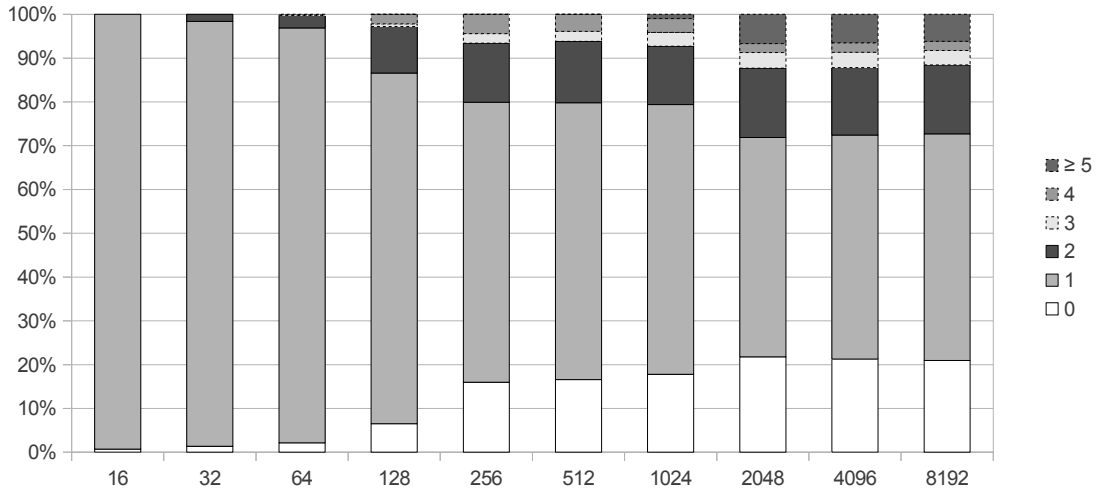


Figura 5.2.8: Distribuição média percentual do número de desvios incluído nos traços reusados. Nota-se patamares em testes com 256 e 2048 entradas. Nestes 100% e 93% dos traços contiveram 4 desvios respectivamente.

dantes(r), *válidas*(v) e ao *total de instruções do programa*(t). Comparamos também o *total de instruções redundantes* com o número de *instruções válidas*(v) e com o *total de instruções do programa*(t). As linhas exibem os resultados dos testes realizado para cada tamanho das tabelas de memorização que variaram de 16 até 8192.

Para o programa AES, exibido na Tabela A.1.1, observa-se a não ocorrência de reuso intra-thread. Por conta disto também não houve nenhuma formação de traço. Entretanto os níveis de reuso inter-thread chegaram a corresponder $70,82\%$ de todas as instruções válidas quando utilizado tabelas com 8192 entradas cada. O maior aumento no nível de reuso ocorreu nos testes com tabelas entre 128 e 256 entradas aproximando-se do patamar identificado para este aplicação em torno dos 40% de reuso.

Os testes com o BFS apresentou melhores níveis de reuso a partir de 1024 entradas em cada tabela ultrapassando a marca dos 46% de reuso, como mostrado na Tabela A.1.2. A maioria dele ocorre entre diferentes thread, representando mais de 73% das instruções redundantes. Tanto o reuso intra-thread quanto o reaproveitamento medido em traços sofrem um aumento entre 128 e 512 entradas estacionando num patamar de 4% e 7% respectivamente. Em relação ao quantitativo de instruções validas o reuso variou entre 35% e 71% nos tamanhos de tabelas testados.

No *benchmark* CP, exibido na Tabela A.1.3, também não ouve reuso de traço, mesmo assim grande parte das instruções válidas foram reusadas. A ampla maioria do reuso medido foi inter-thread. Em tabelas com até 256 entradas a quantidade de reuso inter-thread correspondeu a mais de 98% das instruções redundantes. O maior aumento no nível de reuso ocorre nos testes com tabelas entre 16 e 32 entradas. A

partir de 64 entradas mais de 90% das instruções validas são reusada e a partir de 128 entradas observamos pouco ganho adicional. Quando comparado ao total de instruções medimos 21% de reuso, valor compatível com a média.

Na execução do LPS, medimos mais de 45% de reuso quando usado tabelas a partir de 256 entradas. Até este ponto 100% do reuso era inter-thread. Somente com tabelas a partir de 2048 entradas que um número expressivo instruções redundantes intra-thread começaram a ser detectadas permitindo então a formação e utilização de traços. A partir deste ponto mais de 70% do reuso ocorreu devido reuso de traços.

O resultado do MUM, exibido na Tabela A.1.5, podemos notar mais de 43% de reuso quando utilizamos tabelas com 1024 ou mais entradas. Destas, a maior parte foi devido a reuso dos traços. Eles ocuparam a fatia de quase 70% das instruções redundantes.

No *benchmark* NN, exibido na Tabela A.1.6, não notamos a ocorrência de reuso de traços apesar de termos medido boa quantidade de reuso intra-thread. Com tabelas com entradas variando de 16 até 4096 certa de 50% do reuso foi intra-thread. No geral o nível de reuso aumentou modestamente junto com o aumento do número de entradas na tabela dando um pequeno salto, para $14,95\%$, quando aumentamos as tabelas de 4096 entradas para 8192.

O resultado da execução do *benchmark* NQU, mostrado na Tabela A.1.7, destaca-se pela estabilidade do nível de reuso, na ordem de 63%, entre diferente tamanhos de tabelas. Com apenas 128 entradas já demostra desempenho similar à utilização de tabelas maiores. Quase a totalidade das instruções reusadas provem do reuso inter-thead. Estas representam cerca de 96% do total de instruções reusadas.

Na execução do RAY, mostrado na Tabela A.1.8, observamos estabilidade nos níveis de reuso quando utilizamos tabelas a partir de 32 entradas. Na média obtivemos cerca de 30% das instruções reusadas. Destas mais de 80% foram reuso inter-thread enquanto utilizando tabelas de até 256 entradas. Este valor foi caindo na medida que o reaproveitamento de instruções em traços foi aumentando.

No *benchmark* STO, exibido na Tabela A.1.9, também notamos que não ocorreram reuso em traços e pouquíssimo reuso intra-thread. Apesar disso observamos elevados níveis de reuso no total. Cerca de 70% das instruções foram reusadas e quando utilizamos tabelas de 8192 entradas este valor saltou para 87%.

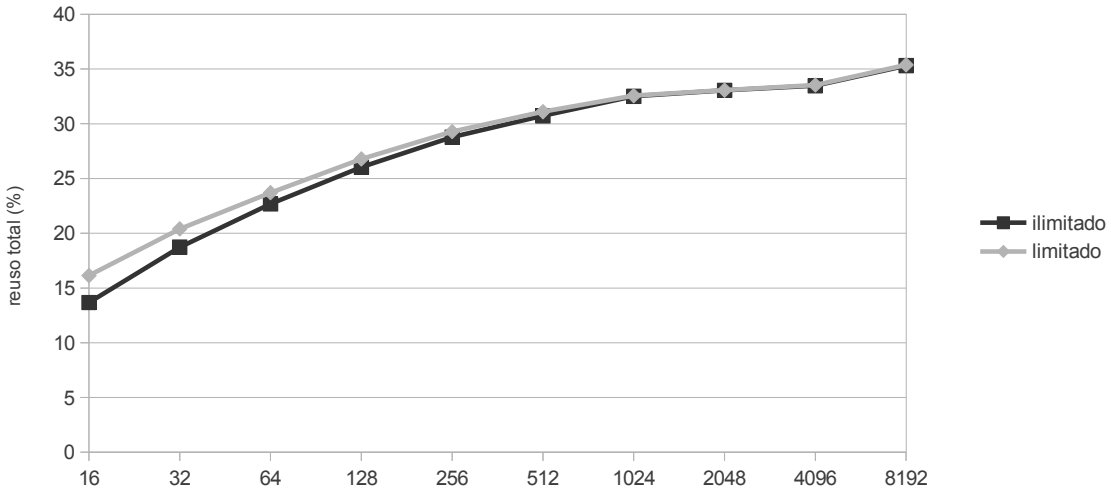


Figura 5.2.9: Reuso total limitando-se o tamanho do contexto. O reuso medido em testes onde os traços redundantes foram limitados em 2 registradores nos contexto de entrada e saída apresentam níveis ligeiramente superiores aos níveis medidos sem esta limitação. Esta diferença ocorre nos testes com pequenas tabelas de memorização e desapareçe quando utilizado tabelas maiores. Este fenômeno ocorre devido a migração de instruções reusadas em traços para instruções reusadas isoladamente que possuem maior probabilidade de reuso.

5.2.10 Reuso limitando tamanho dos contextos

Analisando os dados dos experimentos notamos um ponto de interesse em testes com tabelas de 512 entradas. Nas Figuras 5.2.5 e 5.2.6 é possível observar que, nestes testes, 100% dos traços possuem 2 registradores no contexto de entrada e 83% deles possuem 2 registradores no contexto de saída.

Nesta subseção investigamos qual o potencial de reuso para um cenário onde a formação de traços redundantes está limitada em 2 registradores para cada contexto e comparamos com o cenário original (sem nenhuma limitação). Com isso, pretendemos avaliar o DTM@GPU num cenário mais enxuto aproximando-se de limitações que podem ocorrer num cenário real onde a quantidade de recursos é finita. Nesta rodada de experimentos apenas o tamanho dos contexto de entrada e saída foram limitados. Os demais parâmetros de execução permaneceram os mesmos.

Percentual de reuso

A Figura 5.2.9 exibe a comparação dos níveis de reuso obtidos em testes com e sem limitações no tamanho dos contextos de entrada e saída. O reuso medido com as limitações apresentam níveis ligeiramente superiores aos níveis medidos com recursos ilimitados. Esta diferença apareceu nos testes com tabelas de 16 até 512 entradas. Os níveis de reuso destes dois cenários convergiu à medida que o tamanho das tabelas

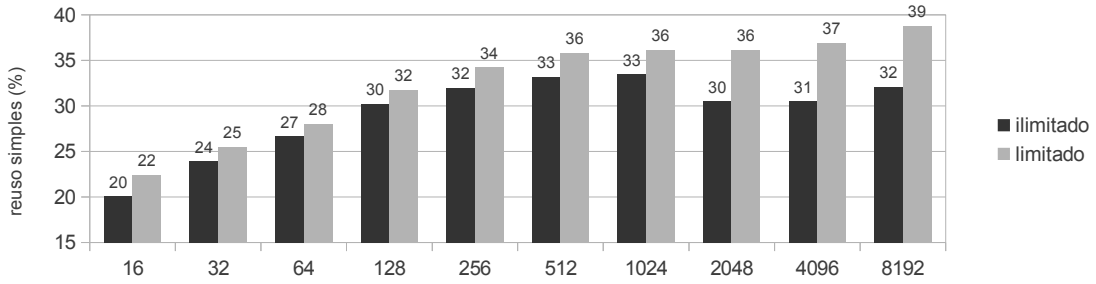


Figura 5.2.10: Comparação das médias aritméticas das instruções reusadas isoladamente entre os testes realizados sem limitação no tamanho dos contextos de entrada e saída e os com esta limitação. Observa-se um aumento no reuso simples devido a diminuição da quantidade de traços redundantes causados pela restrição do tamanho dos contextos.

de memorização aumentava.

A diferença entre os níveis de reuso medido deve-se a migração de instruções reusadas em traços para instruções reusadas isoladamente (reuso *intra-thread* e *inter-thread*), pois com o tamanho do contexto restrito um traço que exceda o limite deve ser dividido em traços menores ou mesmo desfeito caso a divisão implique em menos de duas instruções consecutivas.

O pequeno aumento no nível de reuso para os casos de tabelas pequenas pode ser explicado pela maior probabilidade de uma instrução ser reusada isoladamente do que em um traço. Isto acontece pois como tabelas menores possuem menos entradas é provável que mais acertos ocorram quando o contexto de entrada ou operandos de entrada possuírem menos registradores.

A Figura 5.2.10 mostra a média aritmética das instruções reusadas isoladamente comparando os dois cenários propostos. Já a Figura 5.2.11 mostra a comparação das média aritmética das instruções reusadas dentro de traços entre os cenários avaliados.

Estes resultados demonstram que a limitação dos tamanhos dos traços leva a um aumento no número de instruções reusadas isoladamente e a uma diminuição das instruções reusadas em traços. Esta compensação condiz com a convergência observada para o total de instruções reusadas.

Para os programas AES, CP, NN e STO exibido nas Tabelas A.2.1, A.2.3, A.2.6 e A.2.9, observa-se que não houve mudanças entre os cenários analisados. Eles não foram afetados pois não houve criação de traços nestas aplicações.

Os testes com o BFS, MUM, NQU e RAY, apresentados na Tabela A.2.2, A.2.5, A.2.7 e A.2.8 respectivamente mostram um aumento das instruções redundantes *intra-thread* e redundantes *inter-thread* e uma diminuição das redundantes em traços. Estas aplicações se comportaram de acordo com o valor médio.

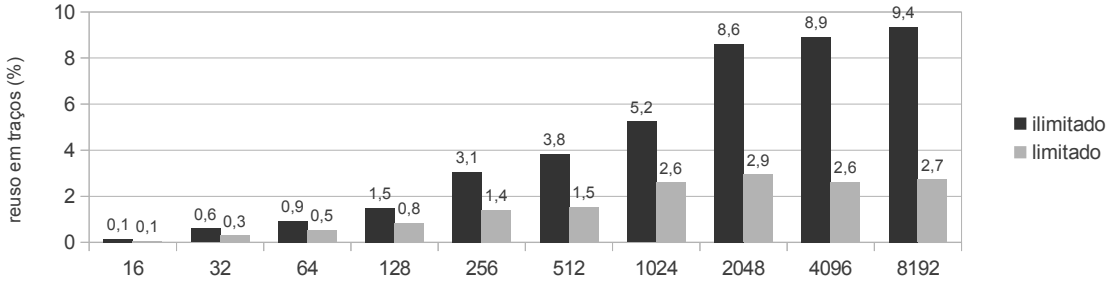


Figura 5.2.11: Reuso em traços limitando-se o tamanho dos contextos. Comparação das médias aritméticas das instruções reusadas dentro de traços entre os testes realizados sem limitação no tamanho dos contextos de entrada e os com esta limitação. Observa-se uma diminuição na quantidade deste tipo de instrução redundante.

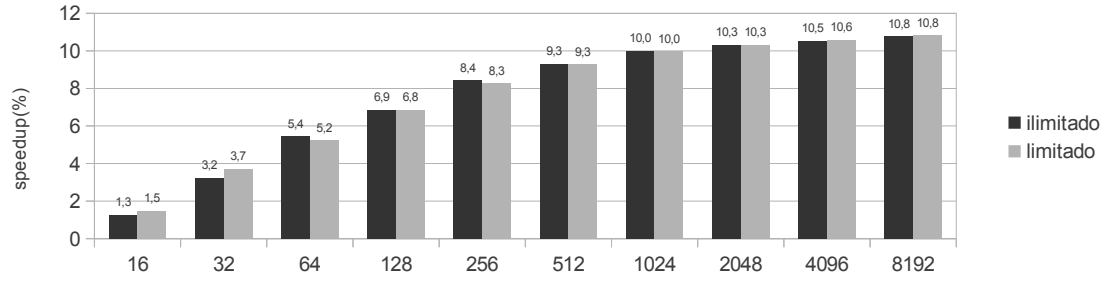


Figura 5.2.12: Estimativa da aceleração de desempenho limitando-se o tamanho dos contextos comparada aos testes sem limitação.

O *benchmark* LPS, exibido na Tabela A.2.4, apresentou aumento das instruções redundantes *intra-thread* e redundantes *inter-thread* e queda da quantidade de instruções reusadas em traços apenas a partir de testes com 1024 entradas. Antes disto, não ouve ocorrência de traços e a distribuição de reuso permaneceu similar em ambos cenários.

Estimativa da aceleração de desempenho

A estimativa da aceleração de desempenho não apresentou grandes variações. Na Figura 5.2.12 observa-se uma oscilação nos valores medidos, porém eles convergem em testes com tabelas maiores.

Este resultado está de acordo com o esperado, pois os níveis de reuso permaneceram os mesmo apesar da ocorrência de uma migração no tipo de reuso predominante. Como a formação dos warps não é sensível ao tipo de reuso, a quantidade de warps dispensados da execução se alterou pouco.

Capítulo 6

Conclusão e Trabalhos Futuros

6.1 Conclusão

A técnica de DTM visa reaproveitar o resultado de traços redundantes de forma que resultados já conhecidos não precisem ser reprocessados. Nos resultados apresentados fica claro que esta proposta de mudança arquitetural consegue expor expressivo nível de redundância.

Impressiona saber que toda esta redundância se manifesta em aplicações que já foram intensamente otimizadas. Esgotadas as alternativas de melhoria em software restam aprimoramentos em *hardware* para aumento de *performance*. O excelente resultado apresentado em um conjunto de testes tão diverso sugere que diferentes classes de aplicações se beneficiariam da aplicação do DTM em GPU.

Neste trabalho, contribuímos identificando as alterações arquiteturais necessárias para implementação do DTM em ambientes massivamente paralelos e definimos um método para compartilhamento das tabelas de memorização entre diversas *threads*. Para isso, distribuímos as tabelas de memorização, MTG e MTT, de forma que cada SP possuísse um par. Nos aproveitamos de que a GPU executa os warps no mesmo SM concorrentemente para permitir o compartilhamento de reuso entre suas *threads*.

Identificamos e medimos três tipos de redundância que podem ocorrer em ambientes *multithreads*, são elas: redundância *intra-thread*, redundância *inter-thread* e reuso de *traços redundantes*. A redundância *intra-thread* ocorre dentro da mesma *thread* e é identificada pelo campo *uid* da MTG, a redundância *inter-thread* ocorre quando uma computação memorizada por uma *thread* diferente é reusada e o reuso de *traços redundantes*, que não faz distinção de quem o criou, representa o reuso de múltiplas instruções de um só vez.

Constatamos que para a correta formação de traços redundantes apenas deveríamos considerar as instruções rotuladas como *intra-thread*. Apesar de criados dentro da mesma *thread* eles podem ser reusados por qualquer outra. Demos o

nome de *traço falso* a aparente construção de traços compostos por sequências de instruções redundantes *inter-thread*. Com esta distinção, habilitamos o compartilhamento de informações de redundância entre várias *threads* sem tornar o estado do processador inconsistente.

Observamos também, que devido a condição de corrida, não poderia haver compartilhamento do buffer de formação dos traços redundantes durante o processo de construção. Sendo assim, implementamos ele junto ao contexto da *thread* tornando-o privativo dela. Não houve diminuição no compartilhamento de redundância, pois apenas a construção é privativa, depois de concluído o traço é adicionado na MTT comum a várias *threads*.

As alterações nos parâmetros arquiteturais do DTM determinadas durantes os preparativos para o experimento, garantiram um bom nível de reuso. Associando um par de tabela de memorização para cada SP conseguimos o compartilhamento das informações de redundância entre todas na mesma faixa de execução. Alterando a MTG para incluir três valores fontes contemplamos 100% das instruções presentes nos programas de testes garantindo uma boa margem para utilização de outras instruções tendo em vista que ampla maioria das instruções computadas possuem dois ou menos operandos fontes.

Constatamos que existe nível de redundância suficiente para que programas não gráficos se beneficiem da aplicação do DTM@GPU. Em nossos experimentos medimos reusos de até 35,3% (média harmônica) em relação ao total de instruções executadas. Detalhando o resultado obtido, apuramos níveis de redundância *intra-thread* da ordem de 10,4% do total de instruções redundantes. Só não medimos valor maior, pois o reuso dos traços memorizados exerce prioridade sobre *redundância intra-thread*. O reuso de traços atingiu a marca de 23,1% das instruções redundantes. Ambos relativos ao testes com 8196 entradas nas tabelas de memorização.

Já o reuso *inter-thread* obteve a maior fatia das instruções redundantes, corresponderam a cerca de 66,4% das instruções reusadas. Apesar de não poderem ser utilizados na formação de traços corresponderam a maior parte do reuso obtido.

Vimos também que o mecanismo desativou as faixas de execução, onde houve reuso de instrução, diminuindo o número de *thread* ativas no warp. Em decorrência disto, observamos um deslocamento no padrão de ocupação da GPU. O número de warps completos diminuiu mais que o número de warps incompletos enquanto o número de warps que deixaram de executar cresceu.

Warps completos, possuindo as 32 *threads* ativas, experimentaram redução de 34% enquanto warps incompletos, possuindo entre 1 e 31 thrads ativas, reduziram 23% em decorrência da variação dos tamanhos de tabelas. Já os warps que deixaram de executar aumentaram em 130% quando utilizado tabelas de 8192 entradas comparados aos testes com 16 entradas.

Os warps incompletos podem representar tanto as *threads* remanescentes da divisão em warps quanto a serialização devido a divergência no fluxo de execução. A redução do conjunto de warps incompletos indica que houve uma diminuição na serialização da execução, como ele possui menos *threads* ativas, necessitaram de menos redundância para serem completamente reusados. Entendemos com isto que o reuso ajudou a diminuir as penalidades devido a divergência de código.

Considerando a natureza SIMD das GPUS, a presença de redundância consegue alcançar, em alguns casos, todas as faixas de execução do processador de forma a ter o SM momentaneamente desabilitado economizando processamento. Medimos que 41,6% dos warps deixaram de executar devido a isto.

Num primeiro momento, a diminuição da ocupação pareceu algo ruim, mas quando o reuso atinge todas as *threads* no warp então ele pode ser dispensado de executar. Contando os ciclos dos warps que deixaram de executar pelo reuso chegamos a estimativa de ganho de desempenho com média harmônica de 9,99%, 10,32%, 10,53% e 10,76% para tabelas de 1024, 2048, 4096 e 8192 entradas respectivamente.

6.2 Trabalhos Futuros

Mesmo com o reuso a quantidade de warps incompletos remanescentes ainda foi expressiva. Neste caso, houve uma piora na taxa de ocupação do processador, pois estes warps tiveram que executar com menos *thread* em relação ao sistema sem DTM@GPU. Em [7] é mostrado um método para formação dinâmica de warps que reagrupa as *threads* de forma a maximizar sua ocupação. Como trabalho futuro acreditamos que a aplicação desta técnica em conjunto com o DTM@GPU poderá aumentar o número de warps reusados uma vez que as *threads* inativadas pelo reuso poderiam ser melhor agrupadas.

Como outra ideia, também consideramos que a presença de warps incompletos sugere a oportunidade de uma política mais agressiva de gestão de energia. Propomos avaliar o impacto energético do desligamento físico das faixas de execução que não estiverem sendo utilizadas neste casos de diminuição da ocupação do processador.

Numa terceira proposta de trabalho futuro sugerimos avaliar o DTM@GPU frente aos cenários apresentados em [8] e medir se o reuso de instruções conseguiria mitigar os efeitos da maior ocorrência de warps divergentes. Este artigo mostra o impacto na *performance* da GPU causado pela variação do tamanho do warp, nele o autor mostra como warps maiores poderiam melhorar diversos aspectos da execução e levar a um melhor aproveitamento dos recursos da GPU. Entretanto, o principal limitante ocorre pelo aumento da incidência de warps divergentes, degradando os ganhos potenciais.

Neste trabalho foi demonstrado o potencial de reuso presente nas GPU pela aplicação da ideia base do DTM, porém investigações anteriores defendidas em [10], [9] e [11] demostrou-se que a adição do suporte à especulação de valores permitiu um melhor aproveitamento dos traços criados. Por causa disso, consideramos importante, nas próximas etapas do desenvolvimento, incorporar esta capacidade especulativa.

Vimos nos resultados apresentados a presença de algumas aplicações com níveis de reuso bem diferentes umas das outras. Aplicações como BFS, LPS, MUM, RAY alcançaram máximo de 15%, 76%, 67% e 54% de reuso em traços respectivamente, enquanto AES, CP, NN e STO não obtiveram nenhum reuso oriundo da formação de traços. Esta diferença dividiu ao meio os programas de testes indicando que estratégias bem distintas devem ser tomadas para melhorar a *performance* delas por meio do DTM.

Atender todas as demandas dos programas exige do arquiteto de computadores grande parcimônia nas decisões tomadas. Em algumas situações delegar parte delas para o programador pode levar a um melhor aproveitamento dos recursos, por exemplo, existe na GPU a possibilidade de dividir o espaço da memória mais nobre do circuito entre cache e memória compartilhada. Com isto em mente, propomos como outro trabalho futuro a introdução de um mecanismo para que o tamanho das tabelas de memorização pudessem ser um parâmetro modificável do processador. Desta forma, uma vez analisado o perfil da aplicação o ajuntes deste parâmetro redirecionaria os recurso de uma tabela para outra.

Referências Bibliográficas

[1] BAKHODA, A., YUAN, G., FUNG, W., et al. "Analyzing CUDA workloads using a detailed GPU simulator". In: *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 163–174, April 2009. doi: 10.1109/ISPASS.2009.4919648.

[2] DA COSTA, A. T. *Explorando Dinamicamente o Reuso de Traces em Nível de Arquitetura de Processador*. Tese de D.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, 1999.

[3] PHAM, D., ASANO, S., BOLLIGER, M., et al. "The design and implementation of a first-generation CELL processor". In: *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pp. 184–592 Vol. 1, Feb 2005. doi: 10.1109/ISSCC.2005.1493930.

[4] FLACHS, B., ASANO, S., DHONG, S., et al. "A streaming processing unit for a CELL processor". In: *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pp. 134–135 Vol. 1, Feb 2005. doi: 10.1109/ISSCC.2005.1493905.

[5] "OpenCL by Example - SIGGRAPH Asia 2010". `http://sa10.idav.ucdavis.edu/`, . Accessed: 2014-02-14.

[6] ARIEL, A., FUNG, W., TURNER, A., et al. "Visualizing complex dynamics in many-core accelerator architectures". In: *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pp. 164–174, March 2010. doi: 10.1109/ISPASS.2010.5452029.

[7] FUNG, W., SHAM, I., YUAN, G., et al. "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow". In: *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pp. 407–420, Dec 2007. doi: 10.1109/MICRO.2007.30.

[8] LASHGAR, A., BANIASADI, A., KHONSARI, A. "Investigating Warp Size Impact in GPUs", *CoRR*, v. abs/1205.4967, 2012.

Disponível em: <http://dblp.uni-trier.de/db/journals/corr/corr1205.html#abs-1205-4967>.

[9] PILLA, M., NAVAUX, P., CHILDERS, B., et al. "Value predictors for reuse through speculation on traces". In: *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, pp. 48–55, Oct 2004. doi: 10.1109/SBAC-PAD.2004.42.

[10] PILLA, M., NAVAUX, P., DA COSTA, A., et al. "The limits of speculative trace reuse on deeply pipelined processors". In: *Computer Architecture and High Performance Computing, 2003. Proceedings. 15th Symposium on*, pp. 36–44, Nov 2003. doi: 10.1109/CAHPC.2003.1250319.

[11] PILLA, M., CHILDERS, B., DA COSTA, A., et al. "A Speculative Trace Reuse Architecture with Reduced Hardware Requirements". In: *Computer Architecture and High Performance Computing, 2006. SBAC-PAD '06. 18TH International Symposium on*, pp. 47–54, Oct 2006. doi: 10.1109/SBAC-PAD.2006.7.

[12] SILVA, B., ABREU, E., FRANCA, F. "JDTM Memorização e reuso dinâmico de traços em uma arquitetura de processador Java". In: *VI Workshop em Sistemas Computacionais de Alto Desempenho, 2005*, pp. 57–64, 2005.

[13] "OpenGL API Documentation Overview". http://www.opengl.org/documentation/, . Accessed: 2014-02-14.

[14] "Direct3D Pipeline Stages". http://msdn.microsoft.com/en-us/library/windows/desktop/bb205123%28v=vs.85%29.aspx. Accessed: 2014-02-14.

[15] PATTERSON, D. A., HENNESSY, J. L. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. 5th ed. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2013. ISBN: 0124077269, 9780124077263.

[16] "Programming Guide for HLSL". http://msdn.microsoft.com/en-us/library/windows/desktop/bb509635%28v=vs.85%29.aspx. Accessed: 2014-02-14.

[17] "Programming Guide for HLSL". http://www.opengl.org/documentation/glsl/. Accessed: 2014-02-14.

[18] "Cg Toolkit". http://developer.nvidia.com/cg-toolkit. Accessed: 2014-02-14.

[19] "GPU Computing". http://www.nvidia.com/object/what-is-gpu-computing.html. Accessed: 2014-02-14.

[20] FAN, Z., QIU, F., KAUFMAN, A., et al. "GPU Cluster for High Performance Computing". In: *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pp. 47–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN: 0-7695-2153-3. doi: 10.1109/SC.2004.26. Disponível em: <http://dx.doi.org/10.1109/SC.2004.26>.

[21] GOVINDARAJU, N., GRAY, J., KUMAR, R., et al. "GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pp. 325–336, New York, NY, USA, 2006. ACM. ISBN: 1-59593-434-0. doi: 10.1145/1142473.1142511. Disponível em: <http://doi.acm.org/10.1145/1142473.1142511>.

[22] LUEBKE, D., HARRIS, M., KRÜGER, J., et al. "GPGPU: General Purpose Computation on Graphics Hardware". In: *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, New York, NY, USA, 2004. ACM. doi: 10.1145/1103900.1103933. Disponível em: <http://doi.acm.org/10.1145/1103900.1103933>.

[23] SHEN, G., GAO, G.-P., LI, S., et al. "Accelerate Video Decoding with Generic GPU", *IEEE Trans. Cir. and Sys. for Video Technol.*, v. 15, n. 5, pp. 685–693, maio 2005. ISSN: 1051-8215. doi: 10.1109/TCSVT.2005.846440. Disponível em: <http://dx.doi.org/10.1109/TCSVT.2005.846440>.

[24] HARRIS, M. "Fast Fluid Dynamics Simulation on the GPU". In: *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM. doi: 10.1145/1198555.1198790. Disponível em: <http://doi.acm.org/10.1145/1198555.1198790>.

[25] "CUDA". http://www.nvidia.com.br/object/cuda_home_new_br.html, . Accessed: 2014-02-14.

[26] "OpenMP Application Program Interface". http://www.openmp.org/mp-documents/spec30.pdf, . Accessed: 2014-02-14.

[27] "The OpenACC Application Programming Interface". http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf, . Accessed: 2014-02-14.

[28] RYOJI TSUCHIYAMA, T. N. *The OpenCL Programming Book: parallel Programming for MultiCore CPU and GPU*. 1.2 ed. , Fixstars, 2012. Disponível em: <http://www.fixstars.com/en/opencl/book/>.

[29] " CUDA C Programming Guide - Design Guide". http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, . Accessed: 2014-02-14.

[30] "Parallel Thread Execution version 3.1". http://docs.nvidia.com/cuda/pdf/ptx_isa_3.1.pdf, . Accessed: 2014-02-14.

[31] "Papers related to GPU/Manycores Bioinformatics". https://biomanycores.org/papers. Accessed: 2014-02-14.

[32] FATICA, M., PHILLIPS, E. "Pricing American Options with Least Squares Monte Carlo on GPUs". In: *Proceedings of the 6th Workshop on High Performance Computational Finance*, WHPCF '13, pp. 5:1–5:6, New York, NY, USA, 2013. ACM. ISBN: 978-1-4503-2507-3. doi: 10.1145/2535557.2535564. Disponível em: <http://doi.acm.org/10.1145/2535557.2535564>.

[33] STANTCHEV, G., JUBA, D., DORLAND, W., et al. "Using Graphics Processors for High-Performance Computation and Visualization of Plasma Turbulence", *Computing in Science and Engg.*, v. 11, n. 2, pp. 52–59, mar. 2009. ISSN: 1521-9615. doi: 10.1109/MCSE.2009.42. Disponível em: <http://dx.doi.org/10.1109/MCSE.2009.42>.

[34] ALLUSSE, Y., HORAIN, P., AGARWAL, A., et al. "GpuCV: An Opensource GPU-accelerated Framework Forimage Processing and Computer Vision". In: *Proceedings of the 16th ACM International Conference on Multimedia*, MM '08, pp. 1089–1092, New York, NY, USA, 2008. ACM. ISBN: 978-1-60558-303-7. doi: 10.1145/1459359.1459578. Disponível em: <http://doi.acm.org/10.1145/1459359.1459578>.

[35] ZHANG, B., XU, S., ZHANG, F., et al. "Accelerating MatLab code using GPU: A review of tools and strategies". In: *Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC), 2011 2nd International Conference on*, pp. 1875–1878, Aug 2011. doi: 10.1109/AIMSEC.2011.6010978.

[36] MANAVSKI, S. "CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography". In: *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pp. 65–68, Nov 2007. doi: 10.1109/ICSPC.2007.4728256.

[37] HARISH, P., NARAYANAN, P. J. "Accelerating Large Graph Algorithms on the GPU Using CUDA". In: *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pp. 197–208, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN: 3-540-77219-7, 978-3-540-77219-4. Disponível em: <http://dl.acm.org/citation.cfm?id=1782174.1782200>.

[38] " CP". http://www.crhc.uiuc.edu/IMPACT/parboil.php. Accessed: 2014-02-14.

[39] GILES, M. "Jacobi iteration for a Laplace discretisation on a 3D structured grid", 2008.

[40] SCHATZ, M., TRAPNELL, C., DELCHER, A., et al. "High-throughput sequence alignment using Graphics Processing Units", *BMC Bioinformatics*, v. 8, n. 1, pp. 1–10, 2007. doi: 10.1186/1471-2105-8-474. Disponível em: <http://dx.doi.org/10.1186/1471-2105-8-474>.

[41] " A Neural Network on GPU". http://www.codeproject.com/KB/graphics/GPUNN.aspx. Accessed: 2014-02-14.

[42] "N-Queens Solver". http://forums.nvidia.com/index.php?showtopic=76893. Accessed: 2014-02-14.

[43] "Ray tracing". http://www.nvidia.com/cuda. Accessed: 2014-02-14.

[44] AL-KISWANY, S., GHARAIBEH, A., SANTOS-NETO, E., et al. "StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems". In: *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pp. 165–174, New York, NY, USA, 2008. ACM. ISBN: 978-1-59593-997-5. doi: 10.1145/1383422.1383443. Disponível em: <http://doi.acm.org/10.1145/1383422.1383443>.

Apêndice A

Resultados Tabelados

A.1 Reuso sem limitações

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,00	0,00	0,00	100,00	33,40	22,35	0,00	0,00	0,00	33,40	22,35
32	0,00	0,00	0,00	100,00	38,92	26,04	0,00	0,00	0,00	38,92	26,04
64	0,00	0,00	0,00	100,00	43,91	29,38	0,00	0,00	0,00	43,91	29,38
128	0,00	0,00	0,00	100,00	49,99	33,45	0,00	0,00	0,00	49,99	33,45
256	0,00	0,00	0,00	100,00	56,83	38,03	0,00	0,00	0,00	56,83	38,03
512	0,00	0,00	0,00	100,00	63,56	42,52	0,00	0,00	0,00	63,56	42,52
1024	0,00	0,00	0,00	100,00	66,08	44,21	0,00	0,00	0,00	66,08	44,21
2048	0,00	0,00	0,00	100,00	66,42	44,44	0,00	0,00	0,00	66,42	44,44
4096	0,00	0,00	0,00	100,00	66,56	44,53	0,00	0,00	0,00	66,56	44,53
8192	0,00	0,00	0,00	100,00	70,82	47,38	0,00	0,00	0,00	70,82	47,38

Tabela A.1.1: Percentual de Reuso do AES separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,14	0,05	0,04	94,93	33,47	23,91	4,93	1,74	1,24	35,26	25,19
32	0,68	0,28	0,20	82,39	34,59	24,71	16,93	7,11	5,08	41,98	29,99
64	1,77	0,82	0,59	78,72	36,59	26,14	19,51	9,07	6,48	46,48	33,20
128	4,45	2,22	1,58	75,37	37,50	26,79	20,18	10,04	7,17	49,76	35,55
256	8,73	4,53	3,24	70,95	36,84	26,32	20,33	10,55	7,54	51,93	37,09
512	11,57	6,12	4,37	67,90	35,90	25,64	20,53	10,85	7,75	52,87	37,77
1024	10,05	6,54	4,67	73,14	47,58	33,99	16,81	10,94	7,81	65,05	46,47
2048	9,76	6,58	4,70	73,99	49,85	35,61	16,25	10,95	7,82	67,37	48,13
4096	9,45	6,58	4,70	74,82	52,07	37,20	15,73	10,95	7,82	69,59	49,71
8192	9,14	6,58	4,70	75,65	54,44	38,89	15,21	10,95	7,82	71,97	51,41

Tabela A.1.2: Percentual de Reuso do BFS separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,00	0,00	0,00	100,00	45,72	9,83	0,00	0,00	0,00	45,72	9,83
32	0,00	0,00	0,00	100,00	80,59	17,33	0,00	0,00	0,00	80,59	17,33
64	0,14	0,13	0,03	99,86	92,54	19,90	0,00	0,00	0,00	92,66	19,93
128	1,48	1,39	0,30	98,52	92,27	19,84	0,00	0,00	0,00	93,66	20,14
256	1,62	1,53	0,33	98,38	92,71	19,94	0,00	0,00	0,00	94,24	20,26
512	16,42	15,72	3,38	83,58	80,01	17,21	0,00	0,00	0,00	95,74	20,59
1024	24,44	23,94	5,15	75,56	74,03	15,92	0,00	0,00	0,00	97,98	21,07
2048	24,44	23,94	5,15	75,56	74,04	15,92	0,00	0,00	0,00	97,98	21,07
4096	24,44	23,94	5,15	75,56	74,04	15,92	0,00	0,00	0,00	97,98	21,07
8192	24,44	23,94	5,15	75,56	74,04	15,92	0,00	0,00	0,00	97,98	21,07

Tabela A.1.3: Percentual de Reuso do CP separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,00	0,00	0,00	100,00	39,58	26,07	0,00	0,00	0,00	39,58	26,07
32	0,00	0,00	0,00	100,00	45,77	30,14	0,00	0,00	0,00	45,77	30,14
64	0,00	0,00	0,00	100,00	51,83	34,13	0,00	0,00	0,00	51,83	34,13
128	0,00	0,00	0,00	100,00	59,80	39,38	0,00	0,00	0,00	59,80	39,38
256	0,00	0,00	0,00	100,00	69,32	45,65	0,00	0,00	0,00	69,32	45,65
512	0,04	0,03	0,02	99,96	75,30	49,59	0,00	0,00	0,00	75,33	49,61
1024	1,78	1,35	0,89	87,74	66,54	43,83	10,48	7,95	5,23	75,84	49,95
2048	12,91	9,80	6,45	16,85	12,79	8,42	70,25	53,32	35,12	75,90	49,99
4096	12,87	9,77	6,43	16,87	12,81	8,44	70,26	53,35	35,14	75,93	50,01
8192	12,87	9,77	6,43	16,87	12,81	8,44	70,26	53,35	35,14	75,93	50,01

Tabela A.1.4: Percentual de Reuso do LPS separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,00	0,00	0,00	100,00	7,14	5,63	0,00	0,00	0,00	7,14	5,63
32	0,13	0,02	0,01	92,88	11,76	9,27	6,99	0,89	0,70	12,66	9,99
64	0,47	0,10	0,08	88,22	19,09	15,06	11,31	2,45	1,93	21,64	17,07
128	1,14	0,38	0,30	75,27	25,39	20,03	23,59	7,96	6,28	33,74	26,61
256	1,92	0,88	0,69	52,07	23,78	18,76	46,01	21,01	16,58	45,67	36,03
512	3,38	1,73	1,36	45,86	23,44	18,49	50,76	25,94	20,47	51,11	40,32
1024	4,99	2,72	2,15	30,88	16,85	13,29	64,14	35,00	27,61	54,58	43,05
2048	7,29	4,14	3,27	23,84	13,56	10,69	68,87	39,16	30,89	56,86	44,85
4096	8,17	4,75	3,74	23,43	13,62	10,74	68,40	39,75	31,36	58,12	45,84
8192	8,71	5,11	4,03	23,32	13,68	10,79	67,97	39,89	31,47	58,68	46,29

Tabela A.1.5: Percentual de Reuso do MUM separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	64,62	7,01	4,11	35,38	3,84	2,25	0,00	0,00	0,00	10,86	6,36
32	59,89	7,63	4,47	40,11	5,11	2,99	0,00	0,00	0,00	12,74	7,46
64	52,73	7,69	4,51	47,27	6,89	4,04	0,00	0,00	0,00	14,58	8,54
128	50,87	8,23	4,82	49,13	7,95	4,66	0,00	0,00	0,00	16,17	9,48
256	53,38	9,57	5,61	46,62	8,36	4,90	0,00	0,00	0,00	17,93	10,51
512	52,50	10,33	6,05	47,50	9,34	5,48	0,00	0,00	0,00	19,67	11,53
1024	51,48	10,81	6,33	48,52	10,18	5,97	0,00	0,00	0,00	20,99	12,30
2048	49,94	10,83	6,35	50,06	10,86	6,36	0,00	0,00	0,00	21,69	12,71
4096	48,53	10,83	6,35	51,47	11,49	6,73	0,00	0,00	0,00	22,32	13,08
8192	42,45	10,83	6,35	57,55	14,69	8,61	0,00	0,00	0,00	25,52	14,95

Tabela A.1.6: Percentual de Reuso do NN separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,20	0,10	0,07	99,09	49,94	35,87	0,71	0,36	0,26	50,40	36,20
32	0,17	0,11	0,08	99,28	65,42	46,99	0,55	0,36	0,26	65,89	47,33
64	0,89	0,63	0,46	97,64	69,24	49,74	1,46	1,04	0,74	70,91	50,94
128	0,91	0,79	0,57	97,19	83,80	60,19	1,90	1,64	1,17	86,22	61,93
256	1,12	0,98	0,71	96,67	84,80	60,91	2,21	1,94	1,39	87,72	63,01
512	1,19	1,05	0,75	96,36	84,80	60,91	2,45	2,15	1,55	88,01	63,21
1024	1,23	1,08	0,78	96,24	84,82	60,93	2,54	2,24	1,61	88,14	63,31
2048	1,17	1,03	0,74	96,12	84,72	60,85	2,71	2,38	1,71	88,14	63,31
4096	1,16	1,02	0,73	96,09	84,69	60,83	2,75	2,43	1,74	88,14	63,31
8192	1,16	1,02	0,73	96,09	84,69	60,83	2,75	2,43	1,74	88,14	63,31

Tabela A.1.7: Percentual de Reuso do NQU separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,00	0,00	0,00	100,00	56,79	19,25	0,00	0,00	0,00	56,79	19,25
32	0,00	0,00	0,00	100,00	66,61	22,58	0,00	0,00	0,00	66,61	22,58
64	0,00	0,00	0,00	100,00	72,50	24,57	0,00	0,00	0,00	72,50	24,57
128	0,01	0,01	0,00	98,77	77,44	26,25	1,22	0,95	0,32	78,41	26,57
256	0,93	0,78	0,26	80,97	67,73	22,96	18,10	15,14	5,13	83,65	28,35
512	8,07	7,14	2,42	63,17	55,85	18,93	28,76	25,43	8,62	88,42	29,97
1024	12,26	11,49	3,89	55,94	52,43	17,77	31,80	29,81	10,10	93,73	31,77
2048	12,36	11,63	3,94	54,03	50,80	17,22	33,61	31,59	10,71	94,02	31,87
4096	11,96	11,26	3,82	47,35	44,57	15,11	40,69	38,30	12,98	94,13	31,90
8192	13,74	13,00	4,41	32,16	30,43	10,31	54,10	51,19	17,35	94,62	32,07

Tabela A.1.8: Percentual de Reuso do RAY separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,00	0,00	0,00	100,00	56,95	51,18	0,00	0,00	0,00	56,95	51,18
32	0,00	0,00	0,00	100,00	60,33	54,21	0,00	0,00	0,00	60,33	54,21
64	0,10	0,07	0,06	99,90	64,18	57,67	0,00	0,00	0,00	64,25	57,73
128	0,66	0,46	0,42	99,34	70,39	63,25	0,00	0,00	0,00	70,86	63,67
256	0,61	0,48	0,43	99,39	78,49	70,53	0,00	0,00	0,00	78,97	70,96
512	0,58	0,48	0,43	99,42	82,48	74,12	0,00	0,00	0,00	82,96	74,55
1024	0,58	0,48	0,43	99,42	82,62	74,24	0,00	0,00	0,00	83,10	74,67
2048	0,58	0,48	0,43	99,42	82,62	74,24	0,00	0,00	0,00	83,10	74,67
4096	0,58	0,48	0,43	99,42	82,62	74,24	0,00	0,00	0,00	83,10	74,67
8192	0,49	0,48	0,43	99,51	96,73	86,92	0,00	0,00	0,00	97,21	87,35

Tabela A.1.9: Percentual de Reuso do STO separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

A.2 Reuso limitando tamanho dos contextos

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,00	0,00	0,00	100,00	35,80	23,95	0,00	0,00	0,00	35,80	23,95
32	0,00	0,00	0,00	100,00	40,81	27,30	0,00	0,00	0,00	40,81	27,30
64	0,00	0,00	0,00	100,00	46,27	30,96	0,00	0,00	0,00	46,27	30,96
128	0,00	0,00	0,00	100,00	52,20	34,93	0,00	0,00	0,00	52,20	34,93
256	0,00	0,00	0,00	100,00	58,97	39,45	0,00	0,00	0,00	58,97	39,45
512	0,00	0,00	0,00	100,00	65,16	43,60	0,00	0,00	0,00	65,16	43,60
1024	0,00	0,00	0,00	100,00	66,21	44,30	0,00	0,00	0,00	66,21	44,30
2048	0,00	0,00	0,00	100,00	66,46	44,47	0,00	0,00	0,00	66,46	44,47
4096	0,00	0,00	0,00	100,00	67,16	44,93	0,00	0,00	0,00	67,16	44,93
8192	0,00	0,00	0,00	100,00	71,20	47,64	0,00	0,00	0,00	71,20	47,64

Tabela A.2.1: Percentual de Reuso do AES limitando-se o tamanho dos contextos separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,45	0,16	0,12	97,46	35,56	25,40	2,10	0,77	0,55	36,49	26,07
32	2,11	0,89	0,63	91,41	38,46	27,47	6,48	2,73	1,95	42,08	30,06
64	4,96	2,32	1,66	88,06	41,13	29,38	6,97	3,26	2,33	46,70	33,36
128	10,66	5,33	3,81	82,25	41,10	29,36	7,09	3,54	2,53	49,97	35,70
256	17,88	9,29	6,64	74,92	38,94	27,82	7,20	3,74	2,67	51,98	37,13
512	21,28	11,25	8,04	71,47	37,81	27,01	7,25	3,83	2,74	52,89	37,79
1024	18,09	11,78	8,42	75,99	49,49	35,35	5,92	3,86	2,75	65,13	46,52
2048	17,54	11,82	8,44	76,74	51,72	36,95	5,72	3,86	2,76	67,40	48,15
4096	16,98	11,82	8,44	77,48	53,93	38,53	5,54	3,86	2,76	69,61	49,73
8192	16,43	11,82	8,44	78,21	56,29	40,21	5,36	3,86	2,76	71,97	51,41

Tabela A.2.2: Percentual de Reuso do BFS limitando-se o tamanho dos contextos separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	1,47	1,18	0,25	98,53	78,76	16,94	0,00	0,00	0,00	79,94	17,19
32	1,59	1,44	0,31	98,41	88,90	19,12	0,00	0,00	0,00	90,34	19,43
64	5,14	4,79	1,03	94,86	88,46	19,02	0,00	0,00	0,00	93,25	20,05
128	18,96	17,78	3,82	81,04	75,97	16,34	0,00	0,00	0,00	93,75	20,16
256	25,37	23,93	5,15	74,63	70,39	15,14	0,00	0,00	0,00	94,32	20,28
512	25,00	23,94	5,15	75,00	71,82	15,44	0,00	0,00	0,00	95,76	20,59
1024	24,44	23,94	5,15	75,56	74,03	15,92	0,00	0,00	0,00	97,98	21,07
2048	24,44	23,94	5,15	75,56	74,04	15,92	0,00	0,00	0,00	97,98	21,07
4096	24,44	23,94	5,15	75,56	74,04	15,92	0,00	0,00	0,00	97,98	21,07
8192	24,44	23,94	5,15	75,56	74,04	15,92	0,00	0,00	0,00	97,98	21,07

Tabela A.2.3: Percentual de Reuso do CP limitando-se o tamanho dos contextos separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,00	0,00	0,00	100,00	42,57	28,03	0,00	0,00	0,00	42,57	28,03
32	0,00	0,00	0,00	100,00	48,03	31,63	0,00	0,00	0,00	48,03	31,63
64	0,00	0,00	0,00	100,00	54,33	35,78	0,00	0,00	0,00	54,34	35,79
128	0,00	0,00	0,00	100,00	62,40	41,10	0,00	0,00	0,00	62,40	41,10
256	0,03	0,02	0,01	99,97	70,40	46,37	0,00	0,00	0,00	70,42	46,38
512	0,19	0,14	0,09	99,32	75,04	49,42	0,49	0,37	0,24	75,55	49,76
1024	48,31	36,64	24,13	36,16	27,42	18,06	15,53	11,78	7,76	75,84	49,95
2048	66,77	50,68	33,38	16,85	12,79	8,42	16,38	12,44	8,19	75,90	49,99
4096	65,65	49,85	32,83	16,88	12,82	8,44	17,48	13,27	8,74	75,94	50,01
8192	65,48	49,72	32,75	16,88	12,81	8,44	17,65	13,40	8,83	75,94	50,01

Tabela A.2.4: Percentual de Reuso do LPS limitando-se o tamanho dos contextos separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,00	0,00	0,00	100,00	8,11	6,40	0,00	0,00	0,00	8,11	6,40
32	0,38	0,06	0,04	93,31	13,81	10,89	6,32	0,93	0,74	14,80	11,67
64	1,40	0,34	0,26	92,52	22,16	17,48	6,08	1,46	1,15	23,96	18,90
128	3,33	1,17	0,92	87,95	30,82	24,31	8,72	3,05	2,41	35,05	27,65
256	6,26	2,82	2,23	73,74	33,23	26,21	20,00	9,01	7,11	45,07	35,55
512	12,34	6,30	4,97	65,40	33,38	26,33	22,27	11,37	8,97	51,04	40,26
1024	19,96	10,90	8,60	56,04	30,61	24,14	24,00	13,11	10,34	54,61	43,08
2048	30,24	17,22	13,58	41,52	23,64	18,65	28,24	16,08	12,68	56,94	44,91
4096	5,84	3,42	2,70	75,32	44,20	34,87	18,85	11,06	8,72	58,68	46,29
8192	11,77	7,04	5,55	67,70	40,48	31,93	20,53	12,28	9,68	59,79	47,16

Tabela A.2.5: Percentual de Reuso do MUM limitando-se o tamanho dos contextos separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	61,84	7,53	4,41	38,16	4,65	2,72	0,00	0,00	0,00	12,18	7,14
32	57,26	7,84	4,60	42,74	5,86	3,43	0,00	0,00	0,00	13,70	8,03
64	58,62	8,80	5,16	41,38	6,22	3,64	0,00	0,00	0,00	15,02	8,80
128	56,92	9,44	5,53	43,08	7,14	4,19	0,00	0,00	0,00	16,58	9,72
256	54,31	9,91	5,81	45,69	8,34	4,89	0,00	0,00	0,00	18,24	10,69
512	53,70	10,68	6,26	46,30	9,21	5,40	0,00	0,00	0,00	19,89	11,66
1024	51,35	10,82	6,34	48,65	10,25	6,01	0,00	0,00	0,00	21,07	12,35
2048	49,90	10,83	6,35	50,10	10,88	6,37	0,00	0,00	0,00	21,71	12,72
4096	48,49	10,83	6,35	51,51	11,51	6,74	0,00	0,00	0,00	22,34	13,09
8192	42,31	10,83	6,35	57,69	14,77	8,65	0,00	0,00	0,00	25,60	15,00

Tabela A.2.6: Percentual de Reuso do NN limitando-se o tamanho dos contextos separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,37	0,21	0,15	99,32	57,19	41,08	0,31	0,18	0,13	57,58	41,36
32	0,34	0,23	0,17	99,40	68,56	49,25	0,27	0,18	0,13	68,98	49,54
64	1,70	1,22	0,88	98,05	70,28	50,49	0,25	0,18	0,13	71,69	51,49
128	2,34	2,03	1,46	97,27	84,48	60,68	0,39	0,34	0,25	86,86	62,39
256	2,97	2,61	1,87	96,57	84,81	60,92	0,46	0,40	0,29	87,82	63,08
512	3,43	3,02	2,17	96,32	84,83	60,93	0,25	0,22	0,16	88,07	63,26
1024	3,56	3,14	2,26	96,18	84,77	60,89	0,26	0,23	0,16	88,14	63,31
2048	3,66	3,22	2,31	96,09	84,69	60,83	0,26	0,23	0,16	88,14	63,31
4096	3,66	3,22	2,31	96,09	84,69	60,83	0,26	0,23	0,16	88,14	63,31
8192	3,66	3,22	2,31	96,09	84,69	60,83	0,26	0,23	0,16	88,14	63,31

Tabela A.2.7: Percentual de Reuso do NQU limitando-se o tamanho dos contextos separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,00	0,00	0,00	100,00	67,46	22,86	0,00	0,00	0,00	67,46	22,86
32	0,01	0,01	0,00	99,42	74,28	25,18	0,56	0,42	0,14	74,71	25,32
64	0,23	0,19	0,06	95,29	76,85	26,05	4,48	3,61	1,22	80,65	27,33
128	7,73	6,57	2,23	84,12	71,48	24,23	8,15	6,93	2,35	84,98	28,80
256	21,72	19,59	6,64	69,92	63,08	21,38	8,35	7,54	2,55	90,21	30,58
512	26,38	24,72	8,38	63,58	59,58	20,19	10,04	9,40	3,19	93,71	31,76
1024	25,19	23,65	8,02	58,98	55,38	18,77	15,83	14,87	5,04	93,90	31,83
2048	32,10	30,22	10,24	50,32	47,36	16,05	17,58	16,54	5,61	94,12	31,90
4096	36,16	34,03	11,53	45,51	42,83	14,52	18,34	17,26	5,85	94,11	31,90
8192	36,23	34,10	11,56	45,43	42,76	14,49	18,34	17,26	5,85	94,11	31,90

Tabela A.2.8: Percentual de Reuso do RAY limitando-se o tamanho dos contextos separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).

	Redundante										
	intra (%)			inter (%)			traço (%)			total (%)	
tabela	r	v	t	r	v	t	r	v	t	v	t
16	0,00	0,00	0,00	100,00	57,59	51,75	0,00	0,00	0,00	57,59	51,75
32	0,02	0,01	0,01	99,98	60,68	54,53	0,00	0,00	0,00	60,69	54,54
64	0,20	0,13	0,11	99,80	64,93	58,35	0,00	0,00	0,00	65,06	58,46
128	0,66	0,47	0,43	99,34	71,13	63,92	0,00	0,00	0,00	71,60	64,34
256	0,60	0,48	0,43	99,40	79,21	71,18	0,00	0,00	0,00	79,69	71,61
512	0,58	0,48	0,43	99,42	82,61	74,24	0,00	0,00	0,00	83,09	74,67
1024	0,58	0,48	0,43	99,42	82,62	74,24	0,00	0,00	0,00	83,10	74,67
2048	0,58	0,48	0,43	99,42	82,62	74,24	0,00	0,00	0,00	83,10	74,67
4096	0,58	0,48	0,43	99,42	82,62	74,24	0,00	0,00	0,00	83,10	74,67
8192	0,49	0,48	0,43	99,51	96,73	86,92	0,00	0,00	0,00	97,21	87,35

Tabela A.2.9: Percentual de Reuso do STO limitando-se o tamanho dos contextos separado por *intra-thread*, *inter-thread* e *reuso de instruções contidas nos traços* em relação ao número de *instruções redundantes*(r), *válidas*(v) e *total de instruções do programa*(t). E ainda, o *total de instruções redundantes* comparado ao número de *instruções válidas*(v) e ao *total de instruções do programa*(t).