Márcio Macedo Gonçalves

Técnicas de Tolerância a Falhas Baseadas em Software de Baixo Nível para Detectar SEUs em Bancos de Registradores de GPUs

Dissertação de Mestrado apresentada ao Programa de Pós Graduação em Computação da Universidade Federal do Rio Grande, como requisito parcial à obtenção do grau de Mestre em Engenharia de Computação.

Universidade Federal do Rio Grande – FURG

Centro de Ciências Computacionais

Programa de Pós Graduação em Computação

Mestrado em Engenharia de Computação

Orientador: Prof. Dr. José Rodrigo Furlanetto de Azambuja

Rio Grande

2017

Ficha catalográfica

G635t Gonçalves, Márcio Macedo.
Técnicas de tolerância a falhas baseadas em software de baixo nível para detectar SEUs em bancos de registradores de GPUs / Márcio Macedo Gonçalves. – 2017.
91 p.

Dissertação (mestrado) – Universidade Federal do Rio Grande – FURG, Programa de Pós-graduação em Computação, Rio Grande/RS, 2017.
Orientador: Dr. José Rodrigo Furlanetto de Azambuja.

1. Tolerância a falhas 2. Técnicas de software 3. GPU I. Azambuja, José Rodrigo Furlanetto de II. Título.

CDU 004.4

Catalogação na Fonte: Bibliotecário Me. João Paulo Borges da Silveira CRB 10/2130

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO RIO GRANDE
CENTRO DE CIÊNCIAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO
CURSO DE MESTRADO EM ENGENHARIA DE COMPUTAÇÃO

DISSERTAÇÃO DE MESTRADO

Técnicas de Tolerância a Falhas Baseada em Software de Baixo Nível para Detectar SEUs em Bancos de Registradores de GPUs

Márcio Macedo Gonçalves

Banca examinadora:

Por videoconferência

Prof. Dr. Raphael Martins Brum

Profa. Dra. Cristina Meinhardt

Prof. Dr. José Rodrigo Furlanetto Azambuja
Orientador

Agradecimentos

Ao Zé, meu orientador, agradeço pela orientação e disponibilidade que me foram concedidas durante a elaboração deste trabalho e pela confiança e oportunidades demonstradas a mim desde a graduação até a conclusão desta dissertação. Ao colega Mateus, agradeço pela sua participação neste trabalho, que foi muito importante para a produção dos dados estudados. Ademais, sou grato à Cristina, professora a quem muitas vezes recorri para esclarecimentos durante a graduação e o mestrado, a qual demonstra muita solicitude e dedicação aos estimados integrantes e colegas do GSDE. Por fim, agradeço a minha família e namorada, por todo o apoio e incentivo nesta etapa da minha vida.

Resumo

Unidades de Processamento Gráfico (GPUs) oferecem grande desempenho computacional para processamento paralelo de dados e contam com ferramentas de desenvolvimento que dão suporte à programação e utilização desses dispositivos em aplicações de propósito geral. Portanto, GPUs são utilizadas em diversos tipos de aplicações e têm despertado o interesse de programadores para utilizá-las também em aplicações que requerem alto grau de confiabilidade, tais como aplicações automotivas, médicas e espaciais. Porém, GPUs são circuitos integrados construídos com as mais modernas tecnologias de circuitos nanométricos, apresentando altíssima densidade de transistores em sua estrutura e operando a altas frequências de *clock*, o que torna GPUs sensíveis a falhas induzidas por partículas de radiação. Deste modo, é imprescindível que técnicas de tolerância a falhas sejam aplicadas para detecção e correção de falhas induzidas por radiação sempre que GPUs forem utilizadas em aplicações críticas.

Este trabalho apresenta uma abordagem de tolerância a falhas baseada em técnicas em *software* de baixo nível para detectar *Single Event Upsets* (SEUs) nos registradores de GPUs, o que inclui os registradores do bancos de registradores de dados, de endereço e de predicado, e os registradores de *pipeline*. As técnicas foram aplicadas sobre quatro algoritmos de estudo de caso, os quais foram executados em uma GPU de propósito geral (GPGPU) baseada na arquitetura NVIDIA G80. Campanhas de injeção de falhas foram feitas em simulação a nível de transferência entre registradores (RTL) nos bancos de registradores e no *pipeline*. Como caso de uso, foram utilizados quatro algoritmos, em suas versões originais e protegidas. Os resultados mostram uma redução em erros de até 100% e custos de tempo de execução e de ocupação de memória de até 77% e 115% superiores aos valores obtidos das aplicações originais, respectivamente.

Palavras-chave: tolerância a falhas; técnicas de *software*; GPU.

Abstract

Graphics Processing Units (GPUs) provide high computational performance for parallel data processing and count with development tools that support the programming and use of these devices in general purpose applications. Therefore, GPUs are used in many types of applications and have attracted developers to use them in applications that require high degrees of reliability, such as automotive, medical and space applications. However, GPUs are integrated circuits built with the most modern nanometric circuitry, presenting very high transistors density in their structure and operating at high clock frequencies, which makes GPUs sensitive to faults induced by radiation particles. Therefore, it is indispensable that fault-tolerance techniques be applied for detecting and correcting radiation-induced faults whenever GPUs are used in critical applications.

This work presents a fault tolerance approach based on low-level software techniques to detect Single Event Upsets (SEUs) in GPUs' registers, which include vector, address, and predicate register files, and the pipeline registers. The techniques were applied to four case-study algorithms, which were executed on a General Purpose GPU (GPGPU) based on the NVIDIA G80 architecture. Fault injection campaigns at Register Transfer Level (RTL) simulation is performed on the register files and on the pipeline. As case-study, four algorithms were used, in their original and hardened versions. Results show reduction in errors up to 100%, and costs of overheads in execution time and program memory footprint up to 77% and 115% superior than original values, respectively.

Key-words: fault tolerence; software technique; GPU.

Lista de ilustrações

Figura 1 – GPU vs CPU: evolução do número de operações de ponto flutuante por segundo . . . 16
Figura 2 – Distribuição de recursos para uma CPU e para uma GPU . . . 17
Figura 3 – Visão Geral da Arquitetura de uma GPU . . . 18
Figura 4 – Exemplo de *grid* configurado para utilizar 128 *threads* em 4 blocos . . . 19
Figura 5 – Exemplo do efeito de SET e SEU em circuitos digitais . . . 22
Figura 6 – Transformações Variáveis . . . 27
Figura 7 – Transformações Desvios Condicionais . . . 28
Figura 8 – Diagrama do Multiprocessador (SM) (MERCHANT, 2013) . . . 32
Figura 9 – Exemplo de tratamento de um *warp* divergente . . . 34
Figura 10 – Distribuição de registradores por *thread* na FlexGrip . . . 36
Figura 11 – Fluxo de *software* de uma GPU NVIDIA . . . 38
Figura 12 – Comportamento dinâmico do algoritmo de Ordenação de Vetores . . . 40
Figura 13 – Comportamento dinâmico do algoritmo de Autocorreção de Vetores . . . 40
Figura 14 – Comportamento dinâmico do algoritmo de Multiplicação de Matrizes . . . 41
Figura 15 – Comportamento dinâmico do algoritmo de Redução de Vetores . . . 41
Figura 16 – VRF Hard: transformações para instruções de operação . . . 47
Figura 17 – VRF Hard: transformações para instruções de *store* . . . 47
Figura 18 – VRF Hard: transformações para instruções de *load* . . . 48
Figura 19 – VRF Hard: transformações para instruções de *branch* . . . 48
Figura 20 – Exemplo de VRF Hard sobre o algoritmo de Multiplicação de Matrizes 49
Figura 21 – PRF Hard: transformações para instruções de desvio condicional . . . 51
Figura 22 – PRF Hard: transformações para instruções de execução condicional . . . 53
Figura 23 – ARF Hard: transformações para operação de leitura . . . 55
Figura 24 – ARF Hard: transformações para operação de escrita . . . 55
Figura 25 – Passos para injeção de uma falha . . . 57
Figura 26 – Exemplo de simulação de um SEU em um banco de registradores . . . 58
Figura 27 – Exemplo de simulação de um SEU em um registrador do *pipeline* . . . 60
Figura 28 – Multiplicação de Matrizes: ocorrência de erros em VRF . . . 64
Figura 29 – Multiplicação de Matrizes: ocorrência de erros em VRF x intervalo de instruções . . . 64
Figura 30 – Autocorrelação de Vetores: ocorrência de erros por registrador em VRF 65
Figura 31 – Autocorrelação de Vetores: ocorrência de erros em VRF x intervalo de instruções . . . 65

Figura 32 – Ordenação de Vetores: ocorrência de erros por registrador em VRF . . 66
Figura 33 – Ordenação de Vetores: ocorrência de erros em VRF x intervalo de instruções 66
Figura 34 – Redução de Vetores: ocorrência de erros por registrador em VRF . . . 67
Figura 35 – Redução de Vetores: ocorrência de erros em VRF x intervalo de instruções 67
Figura 36 – Ponto de Falha de VRF Hard 69
Figura 37 – Ordenação de Vetores: ocorrência de erros em PRF x intervalo de instruções 71
Figura 38 – Multiplicação de Matrizes: ocorrência de erros em PRF x intervalo de instruções 71
Figura 39 – Ordenação de Vetores: ocorrência de erros em ARF x intervalo de instruções 73
Figura 40 – Redução de Vetores: ocorrência de erros em ARF x intervalo de instruções 74
Figura 41 – Taxas de redução de erros no *pipeline* obtidas a partir das técnicas de detecção de falhas 77
Figura 42 – Multiplicação de Matrizes: SDCs x registradores do *pipeline* 78
Figura 43 – Ordenação de Vetores - SDCs x registradores do *pipeline* 79
Figura 44 – Autocorrelação de Vetores - SDCs x registradores do *pipeline* 79
Figura 45 – Redução de Vetores - SDCs x registradores do *pipeline* 79
Figura 46 – Lista de registradores do *pipeline* 92

Lista de tabelas

Tabela 1 – Especificações CUDA para capacidade computacional 1.0 20
Tabela 2 – Tempos de execução e recursos de *hardware* utilizados pelos algoritmos originais 39
Tabela 3 – Recursos de *hardware* demandados pela FlexGrip para sintetização em uma placa Virtex-6 43
Tabela 4 – VRF Hard: desempenho, recursos de memória e registradores utilizados 50
Tabela 5 – PRF Hard: desempenho, recursos de memória e registradores utilizados 53
Tabela 6 – ARF Hard: desempenho, recursos de memória e registradores utilizados 55
Tabela 7 – Tamanho dos blocos de registradores de pipeline 59
Tabela 8 – Injeções em VRF - resultados para os algoritmos originais 63
Tabela 9 – Injeções em VRF - resultados para VRF Hard 68
Tabela 10 – Injeções em VRF - resultados para PRF Hard e ARF Hard 69
Tabela 11 – Injeções em PRF - resultados para os algoritmos originais 70
Tabela 12 – Injeções em PRF - resultados para PRF Hard 72
Tabela 13 – Injeções em PRF - resultados para VRF Hard e ARF Hard 72
Tabela 14 – Injeções em ARF - resultados para os algoritmos originais 73
Tabela 15 – Injeções em ARF - resultados para ARF Hard 74
Tabela 16 – Injeções em ARF - resultados para VRF Hard e PRF Hard 75
Tabela 17 – Injeções no *pipeline* - resultados para os algoritmos originais 75
Tabela 18 – Injeções nos registradores do *pipeline* - resultados para as técnicas de detecção de falhas 76
Tabela 19 – Conjunto de instruções suportadas e testadas na FlexGrip 91

Lista de abreviaturas e siglas

ARF	Banco de Registradores de Endereços
CI	Circuito Integrado
CC	Capacidade Computacional
CPU	Unidade Central de Processamento
CUDA	*Compute Unified Device Architecture*
CTA	*Cooperative Thread Arrays*
DRAM	*Dynamic Random Access Memory*
ECC	Código de Correção de Erros
FlexGrip	*FLEXible GRaphIcs Processor*
EEPROM	*Electrically-Erasable Programmable Read-Only Memory.*
FPGA	*Field Programmable Gate Array*
GPGPU	Unidade de Processamento Gráfico de Propósito Geral
GPP	Processador de Uso Geral
GPU	Unidade de Processamento Gráfico
GPUFI	*GPU Fault Injector*
HDL	*Hardware Description Language*
ISA	*Instruction Set Architecture.*
NRE	Não Recorrentes de Engenharia
PTX	*Parallel Thread Execution*
PC	Contador de Programa
RTL	Nível de Transferência entre Registradores
SDC	*Silent Data Corrruption*
SEE	*Single Event Effect*

SET	*Single Event Transient*
SEU	*Single Event Upset*
PRF	Banco de Registradores de Predicados
SFU	Unidades de Funções Especiais
SIMD	*Single Instruction Multiple Data*
SM	*Streaming Multiprocessor*
SP	Processador Escalar
SRAM	*Static Random Access Memory*
SASS	*Source And Assembly*
TCL	*Tool Command Language*
ILP	Paralelismo em Nível de Instrução
TLP	Paralelismo em Nível de *Thread*
VHDL	*VHSIC Hardware Description Language*
VRF	Banco de Registradores de Dados

Sumário

1 Introdução **14**

2 Fundamentação Teórica **16**

2.1 Unidades de Processamento Gráfico 16

2.2 Falhas Induzidas por Radiação 20

2.2.1 *Single Event Effect* 21

2.2.2 Definição de Falha, Erro e Defeito 22

2.2.3 Métodos de Injeções de Falhas 23

2.3 Técnicas de Tolerância a Falhas para Processadores 24

2.3.1 Técnica de Tolerância a Falhas em Assembly: Variáveis 26

2.3.2 Técnica de Tolerância a Falhas em Assembly: Desvios Condicionais 27

2.4 Trabalhos Relacionados 28

3 Metodologia Proposta **31**

3.1 FlexGrip (*FLEXible GRaphIcs Processor*) 31

3.1.1 Pipeline 32

3.1.1.1 Unidade de *Warp* 32

3.1.1.2 Estágios de Busca e de Decodificação 33

3.1.1.3 Estágios de Leitura e de Escrita 33

3.1.1.4 Estágio de Controle 33

3.1.2 Bancos de Registradores VRF, ARF e PRF 35

3.1.3 Fluxo de *Software* 37

3.2 Algoritmos de Estudo de Caso 38

3.3 Técnicas de Tolerância a Falhas 41

3.4 Campanha de Injeção de Falhas 43

4 Implementação **45**

4.1 Técnicas de Tolerância a Falhas 45

4.1.1 Técnica de Detecção para VRF (VRF Hard) 46

4.1.2 Técnica de Detecção para PRF (PRF Hard) 50

4.1.3 Técnica de Detecção para ARF (ARF Hard) 54

4.2 Injetor de Falhas (GPUFI) 56

4.2.1 SEUs nos Bancos de Registradores 57

4.2.2 SEUs nos Registradores do Pipeline 59

4.2.2.1 Classificação de Falhas 60

5 Resultados e Análise **62**
5.1 Campanhas de Injeção de Falhas 62
5.1.1 Banco de Registradores de Dados 63
5.1.2 Banco de Registradores de Predicado 69
5.1.3 Banco de Registradores de Endereço 72
5.1.4 Registradores de Pipeline 75

6 Conclusão e Trabalhos Futuros **81**

Referências **83**

ANEXO A Códigos Fonte Utilizados **87**
A.1 CUDA Kernel em Assembly: Reduction.sass 87
A.2 CUDA Kernel em Assembly: Autocorr.sass 88
A.3 CUDA Kernel em Assembly: MulMat.sass 89
A.4 CUDA Kernel em Assembly: BitonicSort.sass 90

ANEXO B Conjunto de Instruções **91**

ANEXO C Registradores do Pipeline **92**

1 Introdução

Unidades de processamento gráfico são sistemas dedicados para o processamento paralelo de alto desempenho que possuem múltiplos núcleos em sua arquitetura e tiram proveito do Paralelismo em Nível de *Thread* (TLP) para lidar com cálculos em computação gráfica. A capacidade de manipular rapidamente grandes blocos de memória e executar várias tarefas elementares em paralelo com alto desempenho faz com que GPUs sejam mais eficientes do que os Processadores de Uso Geral (GPPs) para o processamento massivo de dados onde tarefas possam ser executadas em paralelo. Exemplos de aplicações onde tais algoritmos são utilizados são: exploração de petróleo, análise do fluxo de tráfego aéreo, processamento de imagens médicas, álgebra linear, estatística, reconstrução 3D, determinação de preço de venda de ações no mercado financeiro, dentre outras (KRÜGER; WESTERMANN, 2003).

Além do grande desempenho computacional, o advento significativo do suporte à programação de GPUs, como o oferecido pela ferramenta *Compute Unified Device Architecture* (CUDA), provocou a rápida disseminação desses dispositivos, tendo atraído programadores para utilizar GPUs em diversos tipos de aplicações. Neste sentido, surgiu o termo GPGPU, ou *General Purpose Graphics Processing Unit*, que é a utilização de GPUs em aplicações de propósito geral. Entre estas, encontram-se as que requerem alto grau de confiabilidade, tais como aplicações automotivas, médicas e espaciais (NVIDIA, 2016) (HU; XIAO; FRISWELL, 2011) (STRANO et al., 2011). Nessas aplicações, o uso de técnicas de tolerância a falhas é obrigatório para detectar e até mesmo corrigir falhas, uma vez que esses sistemas devem continuar funcionando corretamente mesmo sob a existência de falhas. No entanto, a confiabilidade de GPUs ainda é uma questão em aberto na literatura.

A demanda por desempenho computacional fez com que os últimos modelos de GPUs passassem a ser desenvolvidos em processos de fabricação de até 16 nm, operando a frequências de *clock* superiores a 1 GHz. O aumento na frequência de operação de sistemas digitais, combinado com os respectivos aumentos de densidade de transistores e redução de tensão de alimentação, tornam os Circuitos Integrados (CIs) cada vez mais suscetíveis a falhas induzidas por radiação (BAUMANN, 2001). Tais falhas, causadas principalmente por partículas energizadas de radiação, fazem com que as mais recentes GPUs estejam sujeitas a experimentar erros induzidos por estas partículas (SLAYMAN, 2010) (DIXIT; WOOD, 2011), até mesmo em aplicações terrestres, em execução ao nível do solo, onde os nêutrons são as principais fontes de origem de erros (RECH et al., 2013).

Um dos principais tipos de falhas observado em circuitos de alta densidade de

transistores expostos a radiação é o *Single Event Upset*, o qual é um evento transiente que pode ocorrer em qualquer elemento de memória (DODD; MASSENGILL, 2003). GPUs possuem uma grande quantidade de elementos de memória e o efeito de SEUs deve ser investigado em todas estas estruturas. Uma maneira de qualificar o comportamento de uma GPU sob a ocorrência desses eventos é expondo o circuito integrado a um acelerador de partículas para gerar um feixe de nêutrons ou íons pesados sobre o dispositivo (RECH et al., 2012). Outra opção é a simulação de campanhas de injeção de falhas.

Para lidar com erros induzidos por radiação, técnicas em *software* de tolerância a falhas têm sido propostas para GPPs nos últimos anos, demonstrando taxas elevadas de detecção e baixa degradação de desempenho. O objetivo principal dessas técnicas é proteger o sistema contra erros de fluxo de dados, como SEUs em registradores. Estas estratégias podem ser aplicadas automaticamente sobre o código *assembly* de um programa, simplificando, assim, a tarefa para desenvolvedores de *software*. Deste modo, os custos de desenvolvimento podem ser reduzidos significativamente (RHOD et al., 2008).

Este trabalho tem como objetivo proteger os bancos de registradores de GPUs contra SEUs, através do uso de técnicas clássicas de tolerância a falhas baseadas em *software*, originalmente criadas para processadores de uso geral (AZAMBUJA, 2013). Com isso, pretende-se avaliar os custos dessas técnicas em termos de consumo de memória e de desempenho, bem como o potencial para detecção de falhas em GPUs.

O trabalho é composto por seis capítulos, que podem ser resumidos como segue. O Capítulo 2 apresenta a fundamentação teórica utilizada como base para o desenvolvimento da pesquisa. No Capítulo 3, é exibida a metodologia aplicada para o desenvolvimento da pesquisa. No Capítulo 4, são apresentadas as técnicas de tolerância a falhas implementadas e seus respectivos custos de desempenho e ocupação de memória. O Capítulo 5 expõe as análises e os resultados obtidos das campanhas de injeção de falhas realizadas. Por fim, a conclusão é descrita no Capítulo 6.

2 Fundamentação Teórica

Este capítulo apresenta uma revisão da literatura sobre o funcionamento de GPUs, falhas induzidas por radiação e estado da arte relacionado à proteção de GPUs. Além disso, aborda técnicas de tolerância a falhas utilizadas para proteção de GPPs.

2.1 Unidades de Processamento Gráfico

Originalmente desenvolvidas para processamento gráfico, as arquiteturas de GPUs consistem em um conjunto de multiprocessadores capazes de executar milhares de *threads* em paralelo. A maior parte da área do silício de uma GPU é dedicada para o processamento de dados, enquanto apenas uma pequena porção é destinada para os circuitos que compreendem unidades de controle e *cache*. Esta organização faz com que GPUs sejam mais eficientes do que GPPs para o processamento massivo de dados. GPPs são comercializados em diversos modelos com diferentes capacidades de processamento, podendo ser encontrados em modelos que variam desde microcontroladores de baixos custo e desempenho computacional até processadores de alto desempenho, como os processadores modernos da Intel, os quais, geralmente, são utilizados como Unidades Centrais de Processamento (CPUs) em sistemas computacionais. A Figura 1 mostra um comparativo entre a evolução de CPUs da Intel e GPUs da NVIDIA relacionado ao número de operações de ponto flutuante por segundo (NVIDIA, 2017).

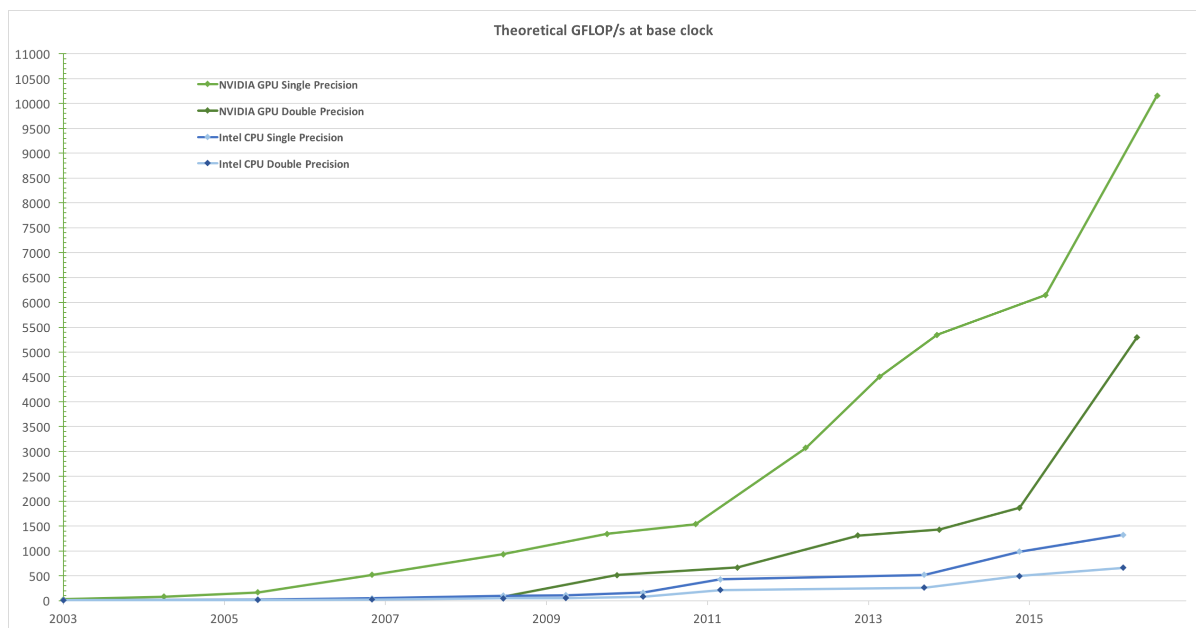

Figura 1 – GPU vs CPU: evolução do número de operações de ponto flutuante por segundo

CPUs, por outro lado, são dispositivos construídos para executar algoritmos mais genéricos de forma sequencial, isto é, as *threads* são escalonadas uma após a outra. Para otimizar o desempenho, CPUs implementam mecanismos de *caches* e de fluxo de controle inteligentes, como por exemplo, predição de desvio dinâmica. Sendo assim, CPUs contam com uma robusta unidade de controle para gerenciar a execução sequencial dos programas e também contam com uma grande quantidade de memória *cache* para minimizar a latência de acesso às instruções e aos dados. As GPUs, por sua vez, contam com várias unidades de processamento, com unidades de controle simples, que instanciam e executam uma grande quantidade de instruções simultaneamente. Neste contexto, a Figura 2 apresenta uma comparação uma CPU e uma GPU.

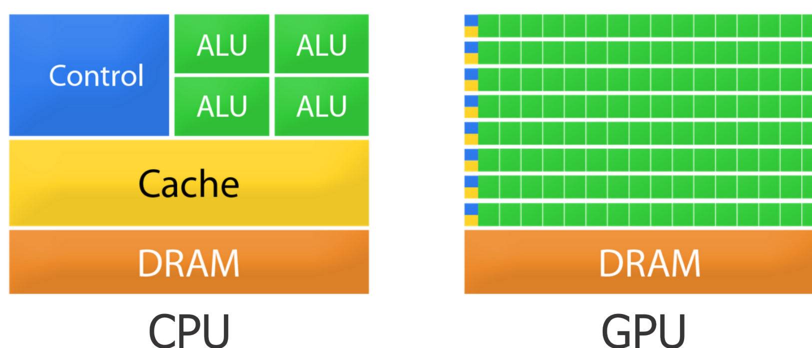

Figura 2 – Distribuição de recursos para uma CPU e para uma GPU

A arquitetura de uma GPU é desenvolvida a partir do modelo *Single Instruction Multiple Thread* (SIMT), que é um modelo computacional que combina *Single Instruction Multiple Data* (SIMD) com *multithreading*, que descreve um método de operação no qual a mesma instrução é aplicada simultaneamente em diversos conjuntos de dados, proporcionando um alto grau de paralelismo. GPUs são constituídas por um conjunto de multiprocessadores, denominados *Streaming Multiprocessors* (SM), que são responsáveis por realizar operações simultâneas em diferentes conjuntos de blocos de dados. Os SMs são constituídos por um conjunto de Processadores Escalares, ou *Scalar Processors* (SPs), ou, ainda, *cores*, que são responsáveis por efetivar a execução das instruções de uma *thread*.

A Figura 3 mostra uma visão geral da arquitetura de uma GPU e nela pode-se observar os SMs e SPs mencionados. Além desses componentes, a figura mostra outros elementos que constituem a GPU, como bancos de registradores, memória global, memória compartilhada e Unidades de Funções Especiais (SFUs). Os bancos de registradores de uma GPU são igualmente divididos entre os SPs de cada multiprocessador, possibilitando que cada SP utilize seu próprio conjunto de registradores para realizar suas computações de forma paralela. A memória compartilhada serve como um meio de comunicação entre os SPs residentes em um mesmo SM. A memória global é visível por todas as *threads*, servindo como meio de comunicação entre *threads* de diferentes SMs. SFUs executam operações aritméticas especiais, tais como seno, cosseno, logaritmos, dentre outras. Além

disso, a arquitetura de uma GPU é constituída por outros elementos que não constam na imagem acima como, memória de instrução e memória constante. A memória de instrução é utilizada para manter o código do programa que é executado pelas *threads* enquanto a memória constante atua como uma memória *cache* para cada multiprocessador e é acessível a todas as *threads* residentes em um bloco em execução.

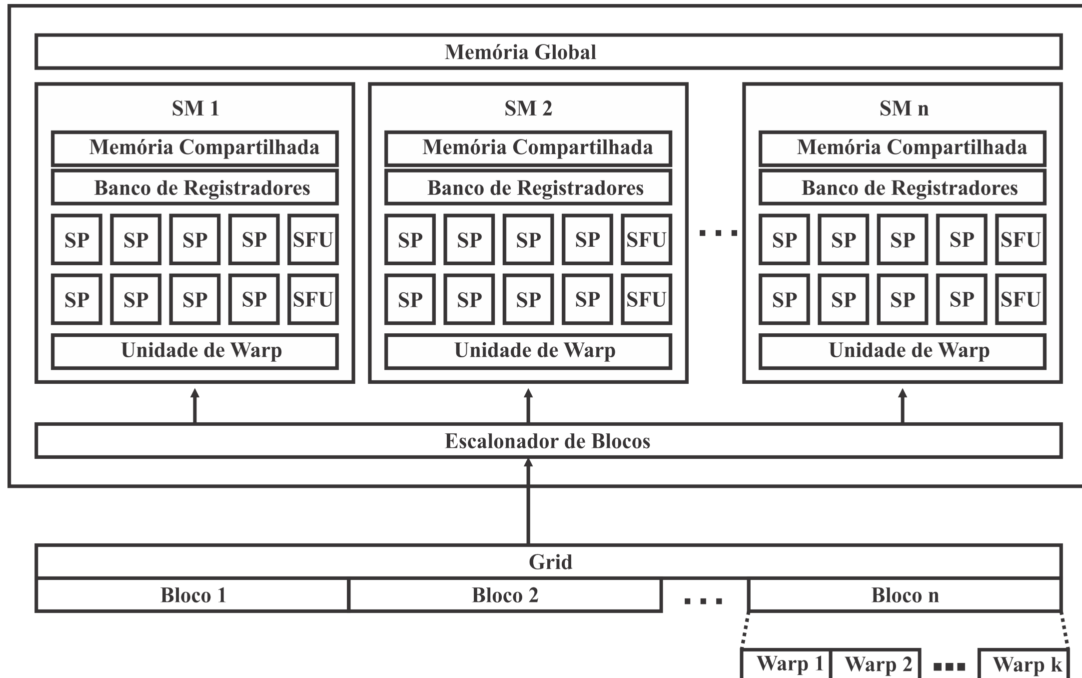

Figura 3 – Visão Geral da Arquitetura de uma GPU

As GPUs da fabricante NVIDIA contam com o modelo de programação CUDA, que foi criado com a finalidade de facilitar a programação de GPUs através do uso de linguagens de programação de alto nível. O modelo CUDA introduz os conceitos de bloco e de *grid*, utilizados para organizar a repartição dos dados entre as *threads*, bem como a distribuição e a organização das *threads* sobre *hardware* da GPU. Um bloco, ou *Cooperative Thread Arrays* (CTA), é a unidade básica de organização das *threads* e de mapeamento para o *hardware*. O *grid* é a unidade básica onde estão distribuídos os blocos e representa a estrutura completa de distribuição das *threads* que executam um *kernel*, que é a função que será executada pela GPU. Portanto, no *grid* está definido o número total de blocos e de *threads* que serão criados e gerenciados pela GPU para processar um dado *kernel*, que é o código do programa que é armazenado na memória de programa da GPU e executado pelas *threads*.

Na programação CUDA, cada *thread* e cada CTA, possuem um endereço único (*id*), que é uma referência que possibilita ao programador manipular as *threads* individualmente, através de seu *id*, para atuar sobre o *kernel* e sobre determinados conjuntos de dados da memória global. A Figura 4 mostra um exemplo de estrutura hierárquica de organização das *threads* em que a GPU é configurada para executar um *kernel* por meio

de um *grid* composto por 4 CTAs, os quais são constituídos por 32 *threads* cada, totalizando 128 *threads*. O *grid* e os CTAs, neste caso, apresentam dimensões de 2x2 e 16x2, respectivamente. Os CTAs podem ter uma, duas ou três dimensões. As *threads* podem ser endereçadas de acordo com a posição em que elas se encontram no *grid*. Considerando o exemplo da figura, a primeira *thread* do primeiro bloco é a *Thread* (0,0) do CTA (0,0) enquanto a última *thread* do último bloco é a *Thread* (15,1) do CTA (1,1). A partir destes parâmetros e do *kernel*, a GPU calcula os endereços dos dados serem processados na memória pelas *threads* em execução.

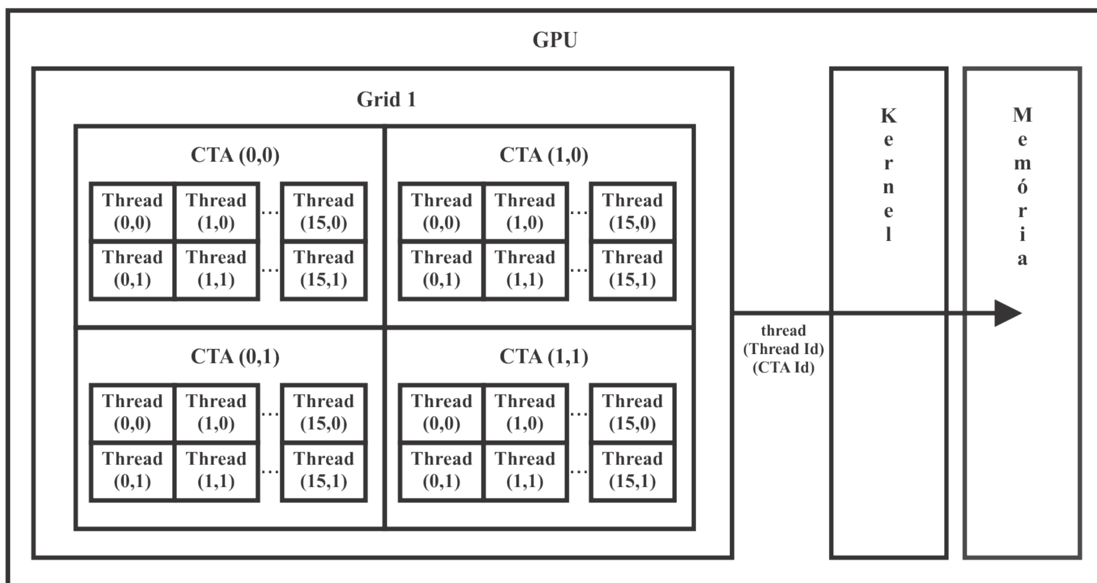

Figura 4 – Exemplo de *grid* configurado para utilizar 128 *threads* em 4 blocos

Sobre o *hardware* da GPU, anteriormente demonstrado na Figura 3, os CTAs são escalonados nos SMs onde são particionados em unidades lógicas menores denominadas *warps*. Um *warp* representa um conjunto de 32 *threads* que tem suas instruções executadas simultaneamente em um SM, de acordo com o número de SPs disponíveis na arquitetura. Quando o número de SPs é inferior ao tamanho do *warp*, o *warp* também é particionado e gerenciado na unidade de warp, que pode ser observada na Figura 3. Múltiplos *warps* podem ser atribuídos a um único SM para o processamento dos blocos, sendo escalonados ao longo do tempo na Unidade de *Warp*. A Figura 3 demonstra os componentes mencionados bem como o particionamento de um *grid* em blocos e *warps*.

A configuração do *grid*, o código do programa e os dados a serem processados pela GPU são transferidos para a GPU através de um dispositivo hospedeiro (*host*), que geralmente é uma CPU. O código CUDA, quando compilado, é dividido em código *host* e código *device*, sendo que o primeiro é executado na CPU e o segundo é executado na GPU. O código conta com regras de sintaxe que possibilitam ao compilador diferenciar os códigos *host* e *device*. Para rodar uma aplicação em uma GPU, primeiramente, a CPU copia os dados a serem processados de sua memória RAM para a memória global da

GPU. Então, a CPU transfere para a GPU a configuração do *grid* e o *kernel* que deve executado. Após a execução do *kernel* na GPU, os dados resultantes, armazenados na memória memória global da GPU, são copiados para a memória da CPU.

As especificações gerais e os recursos disponíveis em GPUs como o número de SMs e de SPs, a quantidade de memória disponível e o número máximo de blocos e *threads* residentes por SM, variam de acordo com o modelo da GPU. Além disso, as instruções disponíveis para os dispositivos também podem variar conforme o modelo. Neste sentido, o modelo de programação CUDA introduziu o conceito de *Compute Capability* (CC), que diferencia modelos distintos de arquitetura de GPUs a fim de serem programáveis pelo mesmo modelo de programação (NVIDIA, 2016). A primeira arquitetura de GPU a contar com o modelo de programação CUDA foi a G80, que implementa a versão 1.0, ou CC 1.0.

A NVIDIA G80 possui 16 SMs e cada um deles contém 8 SPs, totalizando 128 núcleos. Cada SM conta com até 8192 registradores de 32 bits, uma memória compartilhada de leitura e escrita de 16 kB e uma memória constante somente de leitura de 8 kB. As memórias compartilhadas e constantes são implementadas *on-chip*, através de memórias *Static Random Access Memory* (SRAM), e gerenciadas de forma explícita pelo *software*. A memória principal, ou memória global, implementada *off-chip*, é do tipo *Dynamic Random Access Memory* (DRAM), e leva centenas de ciclos para ser acessada. Algumas das especificações de *hardware* CUDA de CC 1.0 são resumidas na Tabela 1 enquanto algumas das instruções suportadas pelo dispositivo são mostradas no Anexo B.

Tabela 1 – Especificações CUDA para capacidade computacional 1.0

Número máximo de blocos residentes por SM	3
Número máximo de threads residentes por SM	256
Número máximo de threads por SM	768
Número máximo de warps residentes por SM	24
Tamanho do warp	32
Quantidade máxima de memória compartilhada por SM	16 kB
Tamanho da memória constante	8 kB
Número máximo de registradores de 32 bits por SM	8192

2.2 Falhas Induzidas por Radiação

Os avanços tecnológicos na indústria de semicondutores têm possibilitado que circuitos integrados mais complexos, mais rápidos e de menor consumo energético sejam construídos em áreas cada vez menores devido a alta densidade de transistores nanométricos presentes nesses circuitos. Porém, ao mesmo tempo em que esses fatores elevam o

desempenho computacional dos circuitos por milímetro quadrado, eles também tornam os circuitos mais suscetíveis a falhas induzidas por partículas de radiação, as quais podem interagir com o circuito e causar erros no sistema (BAUMANN, 2001).

Os circuitos ficam mais suscetíveis a essas falhas pois, ao se reduzir a tensão de operação dos transistores, a fim de reduzir o consumo, esses componentes ficam mais sensíveis a partículas carregadas, as quais poderão ser interpretadas como sinais internos do circuito. Ao elevar a frequência de operação dos circuitos, um pulso transiente introduzido por uma partícula carregada poderá ser capturado mais facilmente por uma ou mais bordas de *clock*. Por fim, a alta densidade de transistores pode fazer com que uma partícula carregada incidente no silício afete vários transistores devido à proximidade em que eles se encontram.

A probabilidade de ocorrência dessas partículas aumenta conforme eleva-se a altitude, sendo mais intensa no espaço (BARTH; DYER; STASSINOPOULOS, 2003) do que ao nível do mar. Muitos trabalhos têm relatado problemas em dispositivos eletrônicos devido à radiação, não apenas em satélites (BINDER; SMITH; HOLMAN, 1975) e aeronaves (DICELLO; PACIOTTI; SCHILLACI, 1989), mas também no nível do mar (MAY; WOODS, 1979). Em terra, os nêutrons são as principais fontes da origem de erros nesses circuitos. Nêutrons podem interagir com circuitos gerando partículas secundárias, como partículas alfa, que perturbam os transistores gerando pulsos de efeito transientes (ZIEGLER, 1996). No espaço, há partículas como prótons, íons pesados e elétrons que podem ionizar os transistores e gerar pulsos transientes. Estes, por sua vez, podem ser interpretados como sinais internos do circuito e gerar erros no sistema.

As falhas causadas por essas partículas nos circuitos integrados podem ser transientes, intermitentes ou permanentes (AZAMBUJA, 2013). Falhas transientes são efeitos únicos que podem ocorrer por um curto período de tempo, ou seja, ocorrem e, em seguida, desaparecem. Falhas intermitentes, por sua vez, são caracterizadas por uma falha que ocorre repetidas vezes, ou seja, passam repetidamente pelo sistema. Finalmente, falhas permanentes são aquelas que permanecem no sistema até que o componente defeituoso seja reparado ou substituído. Neste trabalho, os estudos serão realizados sobre as falhas transientes, que são comumente geradas por partículas de radiação e que podem, muitas vezes, ser detectadas e corrigidas por meio de *software*.

2.2.1 *Single Event Effect*

Os efeitos transientes mais comuns observados em circuitos integrados são conhecidos como *Single Event Effects* (SEE). SEEs são pulsos de voltagem transiente, causados pela incidência de partículas de radiação no circuito, e podem ter efeitos destrutivos ou não destrutivos. Um exemplo de efeito destrutivo é o *Single Event Latchup* (SEL), que resulta em uma alta corrente de operação, acima das especificações do dispositivo. Este tipo

de falha deve ser corrigido através da reinicialização do sistema. Efeitos não destrutivos, também conhecidos como *soft-erros*, são efeitos transitórios provocados pela interação de uma partícula energizada com a junção PN de um transistor em estado de não condução, ou *off-state* (DODD et al., 2004). Quando o pulso transiente ocorre em um elemento de memória, este efeito é denominado *Single Event Upset*, e é representado como uma inversão do valor armazenado no *flip-flop*, ou seja, um *bit-flip*. Quando um pulso transiente ocorre em uma porta lógica de um bloco combinacional, ele é chamado de *Single Event Transient* (SET). Este pulso pode propagar no sistema ocasionar um *bit-flip* em um elemento de memória.

A Figura 5 mostra exemplos de efeitos de um SEU e de um SET em um circuito. À esquerda pode ser observado o efeito de um SEU, em que uma partícula incide sobre um elemento de memória, representado como um registrador, e altera seu valor de "00"para "10". Este efeito afeta o restante do circuito, que altera o valor do registrador da direita de "0"para "1". Sobre a porta NOR do circuito, pode ser observado a incidência de uma partícula que causa um pulso de voltagem sobre a lógica combinacional, o qual se propaga no circuito até a lógica sequencial da direita que registra o valor incorreto "1", ao invés de "0".

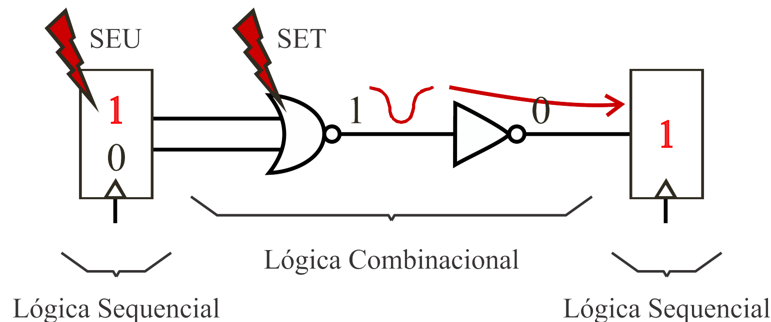

Figura 5 – Exemplo do efeito de SET e SEU em circuitos digitais

2.2.2 Definição de Falha, Erro e Defeito

Efeitos indesejados em circuitos como, por exemplo, uma perturbação causada por uma partícula de radiação, podem corromper o funcionamento do sistema e causar sérios problemas, dependendo da aplicação em execução. Porém, nem sempre isso ocorre, pois estas perturbações podem ser mascaradas através do *hardware* ou do *software* (em sistemas microprocessados), como também podem causar erros irrelevantes para a aplicação em execução. Neste sentido, para classificar os efeitos causados por estas perturbações, serão utilizadas as definições de falha, erro e defeito apresentadas por (AVIZIENIS et al., 2004).

Quando uma partícula de radiação incide sobre um circuito integrado e causa uma perturbação, como um SET, este evento é definido como uma falha. Falhas podem ser mascaradas pelo *hardware* e pelo *software*. No *hardware*, uma falha no circuito pode ser

mascarada devido à lógica combinacional, a fatores elétricos e à janela de amostragem dos registradores. Nesses casos, a falha pode não se propagar até as saídas do sistema. Quando uma falha se propaga até o nível da informação, alterando uma variável utilizada pela aplicação, ainda assim, esta falha poderá ser mascarada durante a execução do *software*. Isso ocorre, por exemplo, quando uma falha altera o valor de um dado armazenado em um registrador que não será utilizado pelo programa ou que será sobrescrito.

Porém, quando acontece uma falha que não é mascarada pelo *hardware* nem pelo *software*, alterando o resultado esperado do sistema, tanto em termos de dados quanto de controle, isso representa um erro. Este erro, por sua vez, poderá causar um defeito. Defeito, ou *failure*, é o comportamento indesejado que corresponde ao mau funcionamento do sistema, o qual geralmente pode ser observado pelo usuário. Entretanto, nem sempre erros se tornam defeitos. Pode-se tomar como exemplo de evento que não representa um defeito, a seguinte situação: uma falha que gera um erro de cálculo em um único pixel de uma imagem de televisão. Porém, se uma falha gera um erro que faz com que a imagem congele, durante um longo tempo, este fenômeno poderá ser considerado um defeito. Em aplicações que requerem alto grau de confiabilidade ou mesmo críticas, onde vidas humanas estão envolvidas, defeitos são inaceitáveis.

2.2.3 Métodos de Injeções de Falhas

Embora os efeitos de falhas existam à nível do mar, a taxa ainda não é suficiente para testar técnicas de tolerância a falhas. Além disso, muitas vezes, projetos de tolerância a falhas devem ser dimensionados para que os sistemas possam funcionar em ambientes de difícil acesso como altas altitudes e até mesmo o espaço. Sendo assim, a emulação e testes de falhas são necessários.

Uma das maneiras de testar sistemas, e a que mais se aproxima dos efeitos reais da radiação, é expor o circuito à aceleradores de partículas, os quais são capazes de acelerar partículas energizadas sobre o circuito. O equipamento é chamado de *Cyclotron* e pode ser encontrado em diferentes lugares do mundo, que utilizam diferentes tipos de partículas energizadas, como íons pesados, prótons e nêutrons. Uma desvantagem deste método de teste são os custos de operação envolvidos. Além disso, com este método não se tem um controle preciso sobre as partes do *hardware* que devem ser afetadas pelas partículas.

Outro método de testes é a simulação dos efeitos da radiação sobre os sistemas, que pode ser realizada em nível elétrico e em nível lógico. A principal vantagem deste método é o controle que se tem sobre a injeção das falhas, o que permite analisar o efeito das falhas sobre diferentes estruturas do sistema. No entanto, é necessário que exista a descrição do circuito e uma ferramenta confiável de simulação. Em nível elétrico, pode-se utilizar, por exemplo, o simulador HSPICE, da Synopsys, e a descrição Spice do circuito. Em nível lógico, pode-se utilizar, por exemplo, o simuladores como o ModelSim, da Mentor, e a

descrição RTL do circuito.

Para placas do tipo *Field Programmable Gate Arrays* (FPGAs), é possível emular falhas na configuração do *chip* através de mecanismos de reconfiguração em que a falha é inserida por meio da carga de um novo *bitstream*, que corresponde ao original mas com um ou mais *bits* invertidos (NAPOLES et al., 2007). Também é possível inserir módulos de *hardware* no circuito com a finalidade emular falhas no sistema (ENTRENA et al., 2012). Tais técnicas podem ser utilizadas em FPGAs como uma alternativa ao simulador, pois, desta maneira, as falhas são executadas muito mais rapidamente. Por outro lado, os simuladores dispensam a utilização de placas de teste e desenvolvimento, as quais devem possuir os recursos de *hardware* necessários para comportar a descrição do sistema a ser testado.

2.3 Técnicas de Tolerância a Falhas para Processadores

Diversas técnicas de tolerância a falhas têm sido propostas na literatura para a proteção de processadores, sendo apresentadas em três diferentes níveis de implementação: técnicas de *hardware*, técnicas de *software* e técnicas híbridas (estas combinam as duas técnicas anteriores) (AZAMBUJA, 2013). Técnicas de tolerância a falhas de *hardware* são aquelas que apresentam os melhores resultados em termos de desempenho. No entanto, a implementação destas técnicas envolve a alteração física do circuito, seja na concepção do circuito integrado ou na alteração da descrição de *hardware* do processador, quando se está trabalhando com FPGAs, por exemplo.

Quando existe a descrição de *hardware* de um processador para FPGA é possível combinar técnicas de *hardware* e de *software* em um único *chip*, explorando o máximo de desempenho a custos de desenvolvimento não muito elevados, porque, neste nível, todas as otimizações são feitas através da programação do FPGA. Contudo, não existem descrições em *hardware* de processadores modernos de alto desempenho, pois são arquiteturas proprietárias e fechadas. Além disso, a implementação da arquitetura desses dispositivos modernos em FPGAs certamente exigiria uma grande quantidade de recursos de *hardware*, os quais só poderiam ser encontrados em FPGAs de alto custo. Por sua vez, as técnicas de *hardware* que necessitam ser aplicadas durante a fabricação do circuito integrado apresentam os maiores custos de implementação, devido a custos Não Recorrentes de Engenharia (NRE).

Técnicas de tolerância a falhas baseadas em *software* são aquelas que despendem de menor custo de desenvolvimento, pois as proteções são feitas através da alteração do código fonte das aplicações, não necessitando de alterações no *hardware*. Entretanto, estas técnicas sempre irão ocasionar consumo extra de memória e degradação de desempenho do sistema. Isso ocorre porque a proteção é realizada através do acréscimo de linhas de

código de programa. Todavia, tais técnicas podem ser aplicadas em qualquer processador e sobre qualquer algoritmo, podendo, então, ser utilizadas para proteger os mais modernos microprocessadores comerciais, os quais apresentam alto desempenho computacional e baixo custo. Neste sentido, a utilização de técnicas em *software* é, atualmente, a melhor opção a ser explorada para proteger GPUs, visto que a principal demanda por estes dispositiem desenvolver soluções de *hardware* para esta finalidade.

As técnicas de *software* podem ser implementadas em alto ou em baixo nível de abstração. Em alto nível de abstração como, por exemplo, em linguagem C, a alteração do código fonte é facilitada pois os códigos são mais facilmente interpretados pelos projetistas. Porém, o processo de compilação do código pode prejudicar a proteção ao retirar redundâncias necessárias das técnicas implementadas. Por outro lado, em baixo nível de abstração (*assembly*), a interpretação e a modificação do código se torna mais difícil pois é necessário conhecer o Conjunto de Instruções da Arquitetura (ISA) do dispositivo e considerar suas características de *hardware* em maior profundidade. No entanto, este nível de implementação proporciona a independência do compilador, além de um maior controle sobre as variáveis e instruções envolvidas na aplicação, pois uma linha de código em alto nível pode se converter em várias linhas em baixo nível.

Conforme será apresentado na Seção 2.4, muitos trabalhos tem sido apresentados para proteger GPUs em alto nível de abstração, mas nenhum foi proposto para proteger estes dispositivos em nível *assembly*. Por esta razão, este trabalho propõe que a proteção dos bancos de registradores de GPUs seja implementada por meio de técnicas de *software* em baixo nível de abstração. Embora estas técnicas não tenham sido aplicada à GPUs, tais técnicas são propostas para proteger os dados e o controle em processadores.

As técnicas orientadas à proteção dos dados são variações do método *Error Detection by Duplicated Instructions* (EDDI) (OH; SHIRVANI; MCCLUSKEY, 2002), que visa preservar a integridade dos valores armazenados nos registradores e na memória de dados do processador, enquanto as técnicas de proteção ao controle visam a integridade do fluxo de execução. (AZAMBUJA, 2010) apresenta uma análise da eficiência de diversas técnicas do estado da arte, implementadas em *assembly*, para proteger aos dados e ao controle, demonstrando taxas de detecção de falhas de até 87%.

A seguir serão abordadas duas técnicas de tolerância a falhas que serão utilizadas como base para proteger GPUs. A primeira é a técnica Variáveis, que será utilizada como base para implementação da proteção dos bancos de registradores de dados e de endereços, e a segunda é a técnica Desvios Condicionais que será utilizada como base para a proteção dos bancos de registradores de predicado.

2.3.1 Técnica de Tolerância a Falhas em Assembly: Variáveis

A técnica Variáveis é definida por um grupo de regras de transformação, as quais são aplicadas sobre o código *assembly* do programa a fim de detectar a presença de falhas em dados. Essas falhas podem causar erros diretos nos dados do sistema, como também podem causar erros indiretos no fluxo de execução das instruções. Variáveis é apresentada na literatura em três versões: VAR1; VAR2 e VAR3 baseadas em (REBAUDENGO et al., 1999), (NICOLESCU; VELAZCO, 2003) e (REIS et al., 2005), respectivamente. Todas estas versões seguem uma metodologia de duplicação de todas as variáveis do sistema - tanto as armazenadas em memória, quanto as armazenadas em registradores - e a verificação de consistência entre as variáveis originais e suas cópias. Cada registrador utilizado pelo programa é replicado sobre registradores não utilizados. Da mesma maneira, as posições de memória ocupadas pelo programa são replicadas sobre posições não ocupadas.

A diferença de implementação entre as técnicas VAR1, VAR2 e VAR3 está relacionada ao momento em que a verificação de consistência é realizada. Em VAR1, o teste de consistência deve ser realizado antes de cada instrução de leitura de uma variável. Em VAR2, o teste de consistência deve ser realizado após cada instrução de escrita sobre um registrador. Em VAR3, a verificação de consistência deve ser realizada antes de cada instrução de escrita na memória. Em todos os casos, sempre que um teste de consistência resultar em discrepância, um erro deve ser sinalizado através de uma sub-rotina.

As técnicas VAR1 e VAR2 podem detectar as falhas com menor tempo desde a sua ocorrência, pois as checagens de consistência acontecem mais frequentemente durante a execução do programa. Além disso, estas duas técnicas possuem um maior potencial para detectar falhas que poderiam ser mascaradas antes de resultarem em um erro. No entanto, em termos de desempenho, a versão que apresenta melhor resultado é a VAR3, pois é a que realiza o menor número verificações de consistência. A Figura 6 demonstra um exemplo de utilização de implementação da técnica VAR3.

Na Figura 6, as instruções originais são mostradas na esquerda e suas respectivas transformações são apresentadas na direita. A instrução original 2 lê um dado da memória endereçado pelo registrador R4. Para proteger esta operação, a instrução 1 é acrescentada para realizar um teste de consistência entre R4 e sua réplica R4' através da instrução *Branch if Not Equal* (*bne*). A instrução 3 é acrescentada para guardar uma réplica de R1 em R1'. A instrução 5 apenas replica a operação da instrução original 4 sobre seus registradores réplicas. Por fim, a instrução original 8, que é um *store*, tem seus registradores R1 e R2, verificados com suas réplicas R1' e R2' através das instruções 6 e 7. Os *offsets*, demonstrados na figura, são utilizados para manter as réplicas da memória.

Instruções de desvio condicional também devem ter seus registradores verificados para evitar erros no fluxo de execução do programa que, indiretamente, podem prejudicar

Código Original	Código Protegido
	1: BNE R4, R4', ERROR;
2: LD R1, [R4];	2: LD R1, [R4];
	3: LD R1', [R4' + OFFSET];
4: ADD R1, R2, 1;	4: ADD R1, R2, 1;
	5: ADD R1', R2', 1;
	6: BNE R1, R1', ERROR;
	7: BNE R2, R2', ERROR;
8: ST [R1], R2;	8: ST [R1], R2;
	9: ST [R1' + OFFSET], R2';

Figura 6 – Transformações Variáveis

as instruções de escrita em memória.

2.3.2 Técnica de Tolerância a Falhas em Assembly: Desvios Condicionais

Instruções de desvio são os mecanismos de *software* que possibilitam o controle do fluxo de execução das aplicações em processadores. Os resultados dos desvios condicionais, que definem se haverá ou não um salto no fluxo de controle, baseiam-se na verificação de valores de variáveis do sistema. Se uma dessas variáveis é alterada por uma falha, poderão ocorrer erros de fluxo de controle no sistema.

Ao executar uma instrução de salto condicional, um programa pode tomar dois caminhos diferentes: se a condição for falsa, o programa segue para a instrução subsequente; se a condição for verdadeira, acontece um salto, propriamente dito, de maneira que o *Program Counter* (PC) será incrementado ou decrementado para alcançar o endereço de destino. A fim de verificar se o caminho seguido foi tomado corretamente, a técnica de desvios condicionais considera a replicação das instruções de desvio condicional nos dois possíveis fluxos de execução.

Quando o desvio não é tomado, replica-se a instrução na posição de memória seguinte do programa, modificando o endereço de desvio para uma sub-rotina que informe um erro. Desta maneira, se o desvio não for tomado na instrução original, também não deverá ser tomado na instrução replicada, a menos que um erro afete o programa. Caso o desvio seja tomado, a instrução replicada deve ser invertida, de maneira que o desvio replicado seja tomado apenas na presença de uma falha, executando, assim, uma sub-rotina que informe o erro ao sistema. É importante mencionar que a instrução replicada, invertida e inserida no fluxo de execução onde o desvio é tomado, não deve interferir no fluxo original de execução, portanto, é necessário adicionar uma instrução de desvio incondicional para contornar a instrução replicada.

A Figura 7 exemplifica o procedimento descrito. As instruções originais são mostradas na esquerda, enquanto as transformações são mostradas na direita. As instruções BEQ e BNE apresentadas na Figura 7, representam, respectivamente, as instruções de desvio condicional *Branch if Equal* e *Branch if Not Equal*. Portanto, pode-se dizer que BEQ representa a replicação inversa de BNE e vice-versa.

Código Original	**Código Protegido**
1: BEQ R1, R2, 6;	**1: BEQ R1, R2, 5;** **2: BEQ R1', R2', ERROR;**
4: ADD R2, R3, 1;	3: ADD R2, R3, 1;
	4: JMP 6; **5: BNE R1', R2', ERROR;**
6: ADD R2, R3, 9; 7: JMP DEST;	6: ADD R2, R3, 9; 7: JMP DEST;

Figura 7 – Transformações Desvios Condicionais

Conforme pode ser observado na Figura 7, para proteger o código original, as instruções 2, 4 e 5 são acrescentadas. A instrução 1 apenas tem seu endereço alterado para a instrução 5 (BNE), que é o teste inverso da instrução 1 (BEQ), de modo que, se o desvio é tomado na instrução 1 então ele não deve ser tomado na instrução 5, a menos que exista divergência entre as réplicas de R1 e R2. Por outro lado, se o desvio não é tomado na instrução 1, então ele também não deverá ser tomado na instrução 2, que é o mesmo teste da instrução 1 (BEQ), mas sobre os registradores réplicas. Caso contrário, o programa é direcionado para uma sub-rotina de erro. A instrução 4 é um salto incondicional que serve apenas para contornar a instrução 5, que somente deve ser executada quando houver salto na instrução 1.

2.4 Trabalhos Relacionados

Proposto por (HUANG; ABRAHAM, 1984), o método de tolerância a falhas *Algorithm-Based Fault Tolerance* (ABFT), ou tolerância a falhas baseada em algorítmos, tem sido bastante explorado em GPUs sob a presença de falhas induzidas por radiação para proteger operações matriciais. (BRAUN; WUNDERLICH, 2010) propõe uma metodologia de utilização da técnica ABFT em GPUs, em que a codificação é realizada no nível de pequenas sub-matrizes processadas em pequenos blocos de *threads*, em vez de toda a matriz. O autor explora a utilização de múltiplos blocos justificando que, no particionamento abordado, recursos de *hardware* como memória cache e memória local são explorados com maior intensidade, a fim de aumentar o desempenho do sistema. (RECH et al., 2013) apresenta o método extABFT, que consiste em uma estratégia otimizada

ABFT para Multiplicação de Matrizes em GPUs, capaz de detectar múltiplas distribuições de erros de saída, as quais são demonstradas pelos autores por meio experimentos que indicam que existem três tipos de distribuição de erro de saída: erros individuais, erros em uma linha/coluna e erros distribuídos aleatoriamente. Nos dois trabalhos citados, os autores demonstram que suas técnicas apresentam um baixo custo em tempo de execução, porém, para se aplicar uma técnica ABFT sobre uma programa, é necessária a reengenharia do mesmo, o que dificulta ou impossibilita a automatização da aplicação da técnica. Além disso, esta técnica é restrita a um pequeno conjunto de aplicações, não podendo ser utilizada em algoritmos genéricos.

O trabalho (OLIVEIRA; RECH; NAVAUX, 2013) apresenta a técnica denominada (*Duplication With Comparison* (DWC), ou duplicação com comparação. Esta técnica baseia-se na duplicação do trabalho realizado pelo algoritmo, reexecutando as transações com a mesma entrada e comparando os resultados. Na referida pesquisa, os autores indicam que, para GPUs, a técnica DWC pode ser implementada de várias maneiras, como através da duplicação de blocos ou *threads*. No entanto, os autores enfatizam que os processos duplicados devem ser cuidadosamente distribuídos, de modo que erros em recursos compartilhados, como memórias caches, ou recursos críticos, como o escalonador, não se propaguem para ambas as cópias, assim prejudicando a capacidade de detecção do método. Três estratégias DWC são exploradas sob diferentes filosofias de duplicação, sendo denominadas espacial, EO-espacial e temporal. Em DWC espacial, os blocos são replicados e executados paralelamente em diferentes SMs. Em DWC EO-espacial, os blocos são replicados e executados no mesmo SM, um após o outro. Em DWC temporal, cada *thread* executa as operações duas vezes e compara os resultados. As estratégias apresentadas podem ser aplicadas a qualquer algoritmo, porém os autores demonstram resultados de degradação de desempenho que variam de 90% a 151%.

(OLIVEIRA; RECH; NAVAUX, 2013) e (RECH et al., 2013) validaram suas estratégias de detecção de falhas através da execução dos algoritmos em uma GPU real sob o bombardeamento de partículas de nêutrons, realizado na instalação ISIS, em Rutherford Appleton Laboratories, Didcot, no Reino Unido. A utilização de GPUs reais sob campanhas de irradiação é imprescindível para se obter um comportamento mais realista de uma GPU sob a incidência de falhas. Porém, este método, além de ser caro, somente pode ser realizado em laboratórios especializados. Deste modo, um simulador de falhas é importante para validar as técnicas aplicadas antes da exposição do dispositivo à uma campanha de radiação. (RECH et al., 2013) também propõe um método de simulação de falhas em que uma *thread* é dedicada para alterar recursos utilizados pela aplicação do estudo de caso. Outro método de simulação de falhas é proposto em (FANG et al., 2014), que faz um estudo da resiliência de erros em GPUs, avaliando um grande número de algoritmos sob a presença de falhas injetadas. Este trabalho usa uma GPU real e um injetor de falhas que utiliza a ferramenta de depuração *cuda-gdb*, capaz de interromper a

execução do programa CUDA na GPU e fazer uso deste breve intervalo de tempo para realizar a injeção da falha. Os modelos de simulação de falhas citados alteram as instruções CUDA a fim de simular um erro. Portanto, não representam um comportamento real da arquitetura. Além disso, estas instruções possuem um potencial limitado para emulação de SEUs em GPUs.

Através dos estudos realizados, pode-se verificar que há muitos trabalhos relacionados a técnicas de tolerância a falhas aplicadas a GPUs. Existem desde técnicas que são restritas a operações matriciais, como ABFT, até técnicas que podem ser aplicadas a algoritmos genéricos, como a DWC. Dentro da pesquisa realizada, todos os trabalhos se baseiam em técnicas aplicadas em *software* de alto nível de abstração. Em (DIMITROV; MANTOR; ZHOU, 2009), entre as abordagens de tolerância a falhas propostas pelos autores, é apresentado o método denominado *R-Scatter*, que tem a finalidade de tirar proveito do Paralelismo a Nível de Instrução (ILP). No trabalho em questão, algumas análises são feitas sobre o código *assembly* da aplicação, porém a técnica é aplicada em alto nível e demonstrada em linguagem C. Uma vez que esse código passa por um processo de compilação, o compilador pode gerar um código *assembly* não otimizado em relação à utilização de recursos, o que acarreta em custos de desempenho ou de detecção de falhas.

A novidade no presente trabalho reside em mostrar que técnicas de tolerância a falhas implementadas em *software* em baixo nível de abstração são capazes de proteger os bancos de registradores de arquiteturas de alto desempenho, como GPUs, assim contribuindo com novas possibilidades para o desenvolvimento de projetos de aplicações críticas para a implementação de sistemas com menores área, custo e tempo de desenvolvimento.

3 Metodologia Proposta

Para a elaboração de uma abordagem de baixo nível para detectar SEUs em GPUs, a metodologia utilizada consistiu em quatro passos: (1) definição de uma GPU de estudo de caso, (2) seleção de um *benchmark* de aplicações a serem protegidas, (3) desenvolvimento e implementação das técnicas de tolerância a falhas para proteger os bancos de registradores de GPUs e (4) a execução de campanhas de injeção de falhas sobre a GPU definida, durante a execução das aplicações de estudo de caso em suas versões originais e protegidas, para validação das técnicas propostas. Este capítulo descreve cada um desses passos em detalhes.

3.1 FlexGrip (*FLEXible GRaphIcs Processor*)

A GPU escolhida para a realização dos estudos foi a *soft-core* GPGPU denominada FlexGrip (MERCHANT, 2013). Descrita em linguagem de descrição de *hardware* (HDL), sintetizável em circuito programável FPGA e testada em uma placa de desenvolvimento ML605 Virtex-6, esta GPGPU apresenta seu código aberto, sem restrições para seu uso ou alterações. Esta GPU foi escolhida por três motivos: o primeiro é devido à possibilidade de realizar alterações de *hardware*, o que, futuramente, permitirá que técnicas de tolerância a falhas em nível de *hardware* possam ser exploradas; o segundo é a possibilidade de explorar o comportamento desta GPGPU sob a simulação de SEUs e SETs aplicados diretamente à elementos de *hardware* desse dispositivo; e o último motivo se deve a esta GPGPU possibilitar que todas as práticas realizadas possam ser verificadas não só através de simulação, mas também por meio da execução em placa FPGA.

A GPGPU FlexGrip foi desenvolvida baseada nos recursos disponíveis da arquitetura G80, que foi a primeira arquitetura a contar com o modelo de programação CUDA. GPUs baseadas na arquitetura G80 possuem 16 SMs, onde cada SM conta com 8 SPs (*cores*). A FlexGrip, no entanto, possibilita que o sistema seja configurado para utilizar 8, 16 ou 32 *cores*, o que permite explorar o consumo de potência e desempenho. Da mesma forma, múltiplos SMs podem ser instanciados a fim de se obter uma melhor relação entre desempenho e consumo de recursos de *hardware* do FPGA. Deste modo, os recursos disponibilizados pela FlexGrip variam de acordo com o número de *cores* e SMs definidos.

A GPGPU conta com um escalonador de blocos que é responsável por escalonar blocos de *threads* através do método *round-robin* em um SM. O número máximo de blocos que podem ser escalonados para um SM é restringido pela quantidade de registradores e de memória compartilhada disponíveis. O SM é composto por uma unidade *warp* e por um *pipeline* de cinco estágios: busca, decodificação, leitura, execução e escrita. A Figura 8,

mostra o diagrama de blocos de um SM da arquitetura FlexGrip. A seguir, são descritos os componentes que compõem o SM, que é o componente responsável por efetivamente executar os blocos de *threads* na GPGPU.

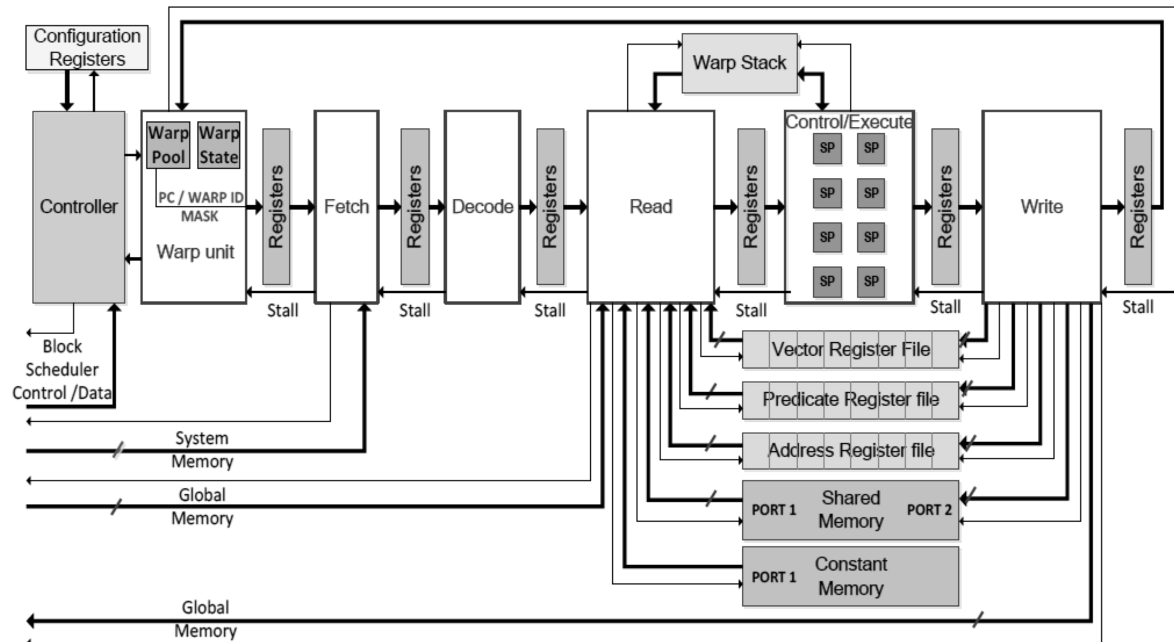

Figura 8 – Diagrama do Multiprocessador (SM) (MERCHANT, 2013)

3.1.1 Pipeline

3.1.1.1 Unidade de *Warp*

As *threads* que chegam em um SM são agrupadas em conjuntos de *warps* por similaridade de operações, sendo que cada um deles é composto por um estado e por dados associados. Na FlexGrip, a unidade *warp* é responsável pela geração destes *warps* e por escaloná-los dentro do SM através do método *round-robin*.

Os dados do *warp* são constituídos por um identificador (ID), o Contador de Programa (PC) e uma máscara de *threads*. Cada *warp* possui seu próprio PC, o que permite independência para assumir seu próprio fluxo de controle, e pode conter até 32 *threads*. O controle individual de cada *thread* é realizado através da máscara de *threads* a qual tem a finalidade de mascarar *threads* que não façam parte do fluxo corrente de execução da instrução CUDA, o que é especialmente útil para a execução condicional de instruções. O tamanho da máscara de *threads* é igual ao tamanho do *warp*, ou seja, 32 *bits*.

O estado do *warp* pode ser Pronto (*Ready*), Ativo (*Active*), Aguardando (*Waiting*) ou Terminado (*Finished*). O estado *Ready* indica que o *warp* está ocioso e pronto para ser agendado. *Active*, indica que o *warp* está sendo processado no *pipeline*. O estado *Waiting*, por sua vez, é utilizado para controle de sincronismo dos *warps* dentro de um bloco. Neste estado, o *warp* indica que encontrou uma instrução de sincronismo e está esperando que

os demais *warps* alcancem este estado. Por fim, quando todas as *threads* do *warp* finalizam a execução do *kernel*, o estado deste *warp* é sinalizado como *Finished*.

Dentro de um *warp*, as *threads* são organizadas em linhas e colunas, cuja quantidade varia de acordo com o número de SPs instanciados dentro de um SM. O *warp* possui tamanho fixo igual a 32 *threads* distribuídas entre linhas e colunas. Sendo assim, em uma configuração com 16 SPs, por exemplo, o *warp* seria organizado em 2 linhas, cada uma contendo 16 *threads*. Da mesma forma, para uma configuração com 8 SPs, o *warp* seria organizado em 4 linhas, cada uma contendo 8 *threads*. A Figura 10 demonstra esta organização, a qual será descrita posteriormente.

3.1.1.2 Estágios de Busca e de Decodificação

O estágio de busca é responsável por buscar a instrução referenciada pelo PC do *warp* associado, na memória de programa. A instrução é então registrada para todas as *threads* residentes no *warp*, sendo mapeada simultaneamente em múltiplos SPs. Instruções CUDA podem ser formadas por 4 (*short*) ou por 8 (*long*) *bytes*. Após realizar a busca da instrução, o PC é incrementado, passando a apontar para a próxima instrução a ser executada. O estágio de decodificação é responsável por decodificar a instrução e gerar diversos sinais de controle, tais como tamanho de instrução, tipo de instrução, operandos de origem e destino, tipos de dados, entre outros.

3.1.1.3 Estágios de Leitura e de Escrita

Conforme pode ser observado na Figura 8, os estágios de leitura e de escrita acessam os três bancos de registradores (dados, endereços e predicado). No estágio de leitura, operandos são lidos de memórias ou de bancos de registradores, dependendo do tipo da instrução decodificada. O banco de registradores é particionado de modo que cada *thread* possua o seu próprio conjunto de registradores. O estágio de escrita, por sua vez, acessa os bancos de registradores de dados, endereços e predicado, preenchendo-os, respectivamente, com dados temporários, *offsets* de memória e sinalizações de predicado. Este estágio também preenche as memórias compartilhada e global com dados temporários ou resultados finais. Quando este estágio termina sua execução, o *warp* retorna para a unidade de *warp*, atualizando seus dados associados e seu estado.

3.1.1.4 Estágio de Controle

O estágio de controle e execução consiste em múltiplos SPs e é responsável por realizar todo o processamento de dados, aritméticos e lógicos, por meio dos SPs. Cada *thread* de um *warp* é mapeada para um SP, possibilitando a execução paralela das instruções. Na FlexGrip, os SPs são capazes de executar apenas operações com dados do tipo inteiro, como adição, subtração, multiplicação, deslocamento de *bits* e operações lógicas

como OR, NOR, XOR, AND, entre outras. Além disso, este estágio também é responsável por executar as instruções de fluxo de controle, como instruções de sincronismo e desvios de fluxo, contando com mecanismos de controle de divergências de execução entre *threads*, que podem ocorrer durante a execução de instruções de desvio condicional.

Um *warp* é considerado divergente quando o resultado de uma instrução de desvio não é o mesmo para todas as *threads* do *warp*. A instrução de sincronismo SSY é utilizada para definir o ponto de reconvergência da instrução de desvio que será alcançada, independentemente se o desvio é tomado ou não. O controle de execução de um *warp* divergente é realizado com o auxílio da máscara de *threads* e de um estrutura de pilha, que pode ser observada na Figura 8 com o nome de *Warp Stack*.

A Figura 9 demonstra um exemplo em que ocorre a divergência de execução em uma instrução de desvio em um trecho de código. Conforme especificado, o tamanho da máscara do *warp* da GPGPU é igual a 32, mas, por simplificação, o exemplo considera como se fosse de apenas 4. O valor de um *bit* da máscara em "1"significa que a *thread* deve executar a instrução de desvio, enquanto o valor "0"significa que a instrução não deve ser executada pela *thread*. Portanto, o exemplo considera uma situação em que apenas uma *thread* ("1000") deve tomar o desvio enquanto as demais ("0111") não devem tomar o desvio. O exemplo também considerada uma situação em que o tratamento do desvio ocorre primeiro. O processo ocorre em seis passos, destacados em círculos na figura, os quais são descritos da seguinte forma:

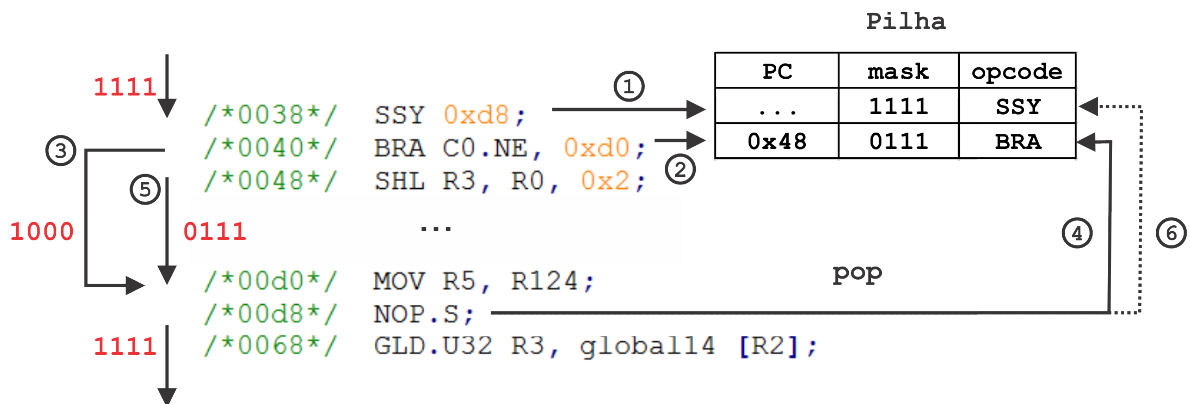

Figura 9 – Exemplo de tratamento de um *warp* divergente

1. quando a instrução SSY é alcançada, o valor da máscara corrente ("1111") e do *opcode* de SSY são armazenadas na pilha. Após, o programa segue até encontrar um ponto de possível divergência;

2. quando a instrução de desvio condicional é alcançada, a *thread* que realiza o desvio é tratada antes das demais e, assim, a máscara para as *threads* remanescentes ("0111") é armazenada na pilha juntamente com o valor do PC da instrução seguinte à instrução do salto ("0048");

3. após guardar os dados referentes ao caminho não tomado, a *thread* "1000"salta e percorre o seu caminho até encontrar a instrução de reconvergência NOP.S;

4. quando a instrução NOP.S é executada, ocorre um *pop* na pilha, cujo topo contém os dados que permitem a retomada o PC e das *threads* que devem executar o caminho não tomado;

5. neste ponto, as *threads* remanescentes "0111"executam o seu caminho até encontrar a instrução de reconvergência NOP.S;

6. quando a instrução de reconvergência é alcançada novamente, um novo *pop* ocorre na pilha, cujo topo, neste momento, conterá o *opcode* de SSY e a máscara "1111", que possibilita a retomada de todas as *threads* que estavam sendo executadas antes da divergência. Porém, neste segundo *pop*, o *opcode* contém o código de SSY e quando isso ocorre o PC é apenas incrementado para o endereço da próxima instrução, que é o endereço "0068". Neste ponto os dois caminhos foram tratados e o processo está concluído.

3.1.2 Bancos de Registradores VRF, ARF e PRF

Os bancos de registradores são divididos em três grupos: Banco de Registradores de Dados (VRF), Banco de Registradores de Endereços (ARF) e Banco de Registradores de Predicado (PRF). O banco VRF é utilizado para guardar dados intermediários para a execução das instruções, enquanto o ARF é utilizado para guardar os *offsets* de memória para operações de leitura e escrita, em que os dados são lidos em bloco ao invés de sequencialmente, e o banco PRF é utilizado para guardar sinalizações de predicado, ou *flags*, que são utilizadas para execução condicional de instruções. Estas *flags* podem indicar diversas condições, como zero, não zero, sinal, vai um, dentre outros.

Os bancos de registradores VRF, ARF e PRF são criados de acordo com o número de *cores* definido. Cada *core* possui seus próprios bancos de registradores, o que permite que todos os *cores* possam acessar os registradores de forma paralela. Os bancos VRF e ARF são compostos por registradores de 32 bits, enquanto PRF é composto por registradores de 4 bits. Para cada *thread*, são destinados apenas quatro registradores de endereço e quatro registradores de predicado, enquanto a quantidade de registradores de dados varia de acordo com a aplicação. Cada banco de registradores VRF foi inicialmente projetado para ser implementado em FPGA através de blocos de memórias SRAM de tamanho de 1152 *bytes*. Uma vez que um SM comporta até 24 *warps* e que cada registrador de VRF possui 32 *bits*, quando o número máximo de *threads* for instanciado por uma aplicação, o número máximo de registradores de dados disponíveis para cada *thread* será igual a 12. Entretanto, quanto menos *threads* forem utilizadas, mais registradores estarão disponíveis para cada uma delas. Por outro lado, cada banco ARF e PRF possui tamanhos

de 384 e 48 *bytes*, respectivamente, disponibilizando assim o número invariável de quatro registradores para cada *thread*, independentemente do número de *threads* instanciadas.

Conforme mencionado, o *warp* é organizado e executado em linhas e colunas que variam de acordo com o número de *cores* utilizados. Tal organização implica na distribuição de *threads* e de seus respectivos registradores. Neste sentido, dois aspectos são considerados: (1) quando o número de *cores* for menor que o tamanho do *warps*, ou seja, 32, mais linhas são criadas e (2) quando o número de *threads* supera o tamanho do *warps*, novos *warps* são criados. A distribuição de registradores entre as *threads* depende, portanto, do número de *cores* e do número de *threads* utilizados.

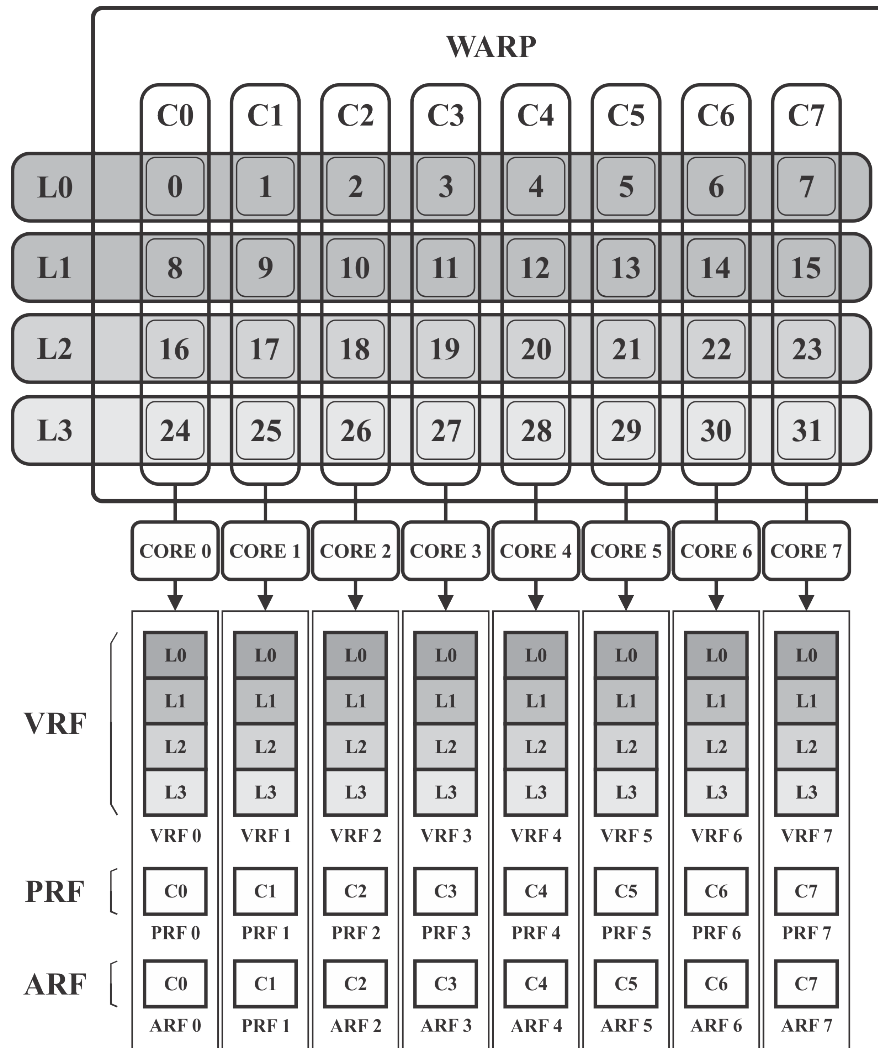

Figura 10 – Distribuição de registradores por *thread* na FlexGrip

A Figura 10 demonstra uma situação em que a FlexGrip é configurada para operar com 8 *cores*. Neste caso, 4 linhas e 8 colunas são geradas. As linhas são escalonadas sobre os *cores* para concluir a execução das 32 *threads* nos 8 processadores escalares. Independentemente se o número de *threads* instanciadas por uma aplicação é menor do que 32, todas as linhas sempre são escalonadas. O VRF é particionado em linhas, enquanto

o PRF e o ARF são acessados por meio de um *offset* igual 4xL, sendo L a linha do *warp* e 4 o número de registradores por *thread*, que é fixo para esses dois bancos.

3.1.3 Fluxo de *Software*

Conforme apresentado na Seção 2.1, em uma GPU NVIDIA, o *kernel* é invocado por um *host*, o qual geralmente é uma CPU. Esta invocação, na prática, significa enviar o respectivo código do *kernel* para a Memória de Sistema (*System Memory*) da GPU para ser executado. Porém, antes de uma aplicação ser lançada, é preciso configurar a GPU, definindo o tamanho dos *grids* de blocos e de *threads* que se deseja utilizar. Além disso, deve-se carregar na memórias global os dados a serem processados pelas *threads*. Para executar uma aplicação na FlexGrip, esses parâmetros de configuração, bem como os valores iniciais das memórias compartilhada e global são configurados diretamente no HDL da GPGPU. Da mesma forma, são incluídas as instruções do *kernel* que representam o algoritmo a ser executado.

Para o obter o código do programa a ser executado pela FlexGrip é necessário, primeiramente, que o algoritmo descrido em linguagem de alto nível seja compilado. O processo de compilação de algoritmos CUDA é demonstrado na Figura 11. Durante o tempo de compilação, o compilador recebe o *kernel* CUDA, convertendo-o em um código denominado *Parallel Thread Execution* (PTX). PTX é uma linguagem de programação *assembly* intermediária, que fica entre a linguagem de alto nível e o *assembly* que realmente é executado no dispositivo. O objetivo da PTX, ou *assembly* virtual, é disponibilizar uma linguagem de baixo nível compatível com GPUs que apresentam diferentes capacidades computacionais. Deste modo, a linguagem PTX não representa diretamente o conjunto de instruções de máquina da GPU, pois ela é compilada para instruções *assembly*, as quais dependem da capacidade computacional de cada GPU. Durante o tempo de execução, o código PTX passa pelo *driver* CUDA, que é responsável por gerar o código binário *cubin*, que é então direcionado para a GPU para ser efetivamente executado.

Conforme pode ser observado na Figura 11, no processo de compilação, o código PTX é convertido em outro formato de código denominado *Source and Assembly* (SASS) (NVIDIA, 2013). Este formato representa as instruções nativas que são interpretadas pelo *hardware* da NVIDIA. Porém, o código SASS é gerado em tempo de execução e não é visível para o usuário final. Ainda assim, pode-se obter esse código por meio da ferramenta *cuobjdump*, disponibilizada pelo *CUDA toolkit* da NVIDIA (NVIDIA, 2013). Esta ferramenta gera o código SASS a partir do código *cubin*, descrito anteriormente. Por fim, para executar um *kernel* na FlexGrip, os códigos hexadecimais resultantes do arquivo SASS devem ser copiados para FlexGrip antes de executar uma aplicação. Exemplos de código SASS são mostrados no Anexo A.

Desta maneira, para que se possa proteger os códigos de GPUs da NVIDIA em

baixo nível, é preciso alterar o código fonte SASS. É possível também alterar o código em nível PTX, mas, neste nível, as otimizações podem sofrer alterações indesejadas no processo de compilação para o *assembly* final.

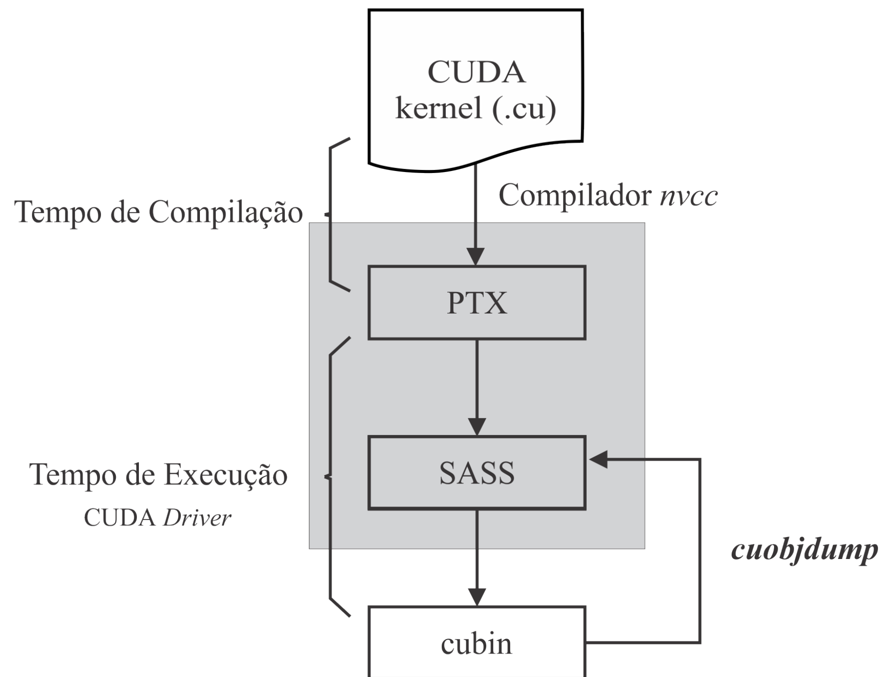

Figura 11 – Fluxo de *software* de uma GPU NVIDIA

3.2 Algoritmos de Estudo de Caso

As aplicações selecionadas compõem um conjunto de quatro algoritmos que exploram o uso intensivo do caminho de dados e do caminho de controle da GPU, o que permitirá uma avaliação mais justa das técnicas de tolerância a falhas aplicadas. Estes algoritmos são: (1) Multiplicação de Matrizes, (2) Ordenação de Vetores, (3) Autocorrelação e (4) Redução (CHANG et al., 2010) e (NVIDIA, 2009). Todos operam sobre dados do tipo inteiro, visto que a atual versão da FlexGrip pode executar apenas operações com este tipo de dado. O algoritmo de Ordenação é o que explora as operações de fluxo de controle com mais intensidade, enquanto Autocorrelação apresenta algumas instruções de fluxo de controle. Os algoritmos de Multiplicação de Matrizes e de redução são dedicados para o processamento intensivo de dados em paralelo. Todos estes algoritmos são facilmente escaláveis, o que possibilita o teste de diferentes configurações de vetores de entrada, que pode ser feito através da alteração dos parâmetros de configuração da GPU, de modo que seja utilizada a quantidade de blocos de *threads* adequada para a respectiva entrada de dados. Esta característica pode ser útil para, futuramente, verificar a eficiência das técnicas aplicadas sobre diferentes configurações de paralelismo da GPGPU.

O algoritmo de Multiplicação de Matrizes multiplica duas matrizes quadradas. O algoritmo de Ordenação de Vetores é o Bitonic Sort, um algoritmo paralelo de Ordenação de dados que se encontram dispostos em sequências bitônicas (DIZ, 2010). O algoritmo de Autocorrelação, por sua vez, realiza a correlação de um sinal consigo efetuando operações de Multiplicação e adição sobre elementos do vetor de entrada. Finalmente, o algoritmo de Redução selecionado utiliza o operador de somatório para somar todos os elementos de um *array* em paralelo.

Para a execução do algoritmo de Multiplicação de Matrizes, a FlexGrip foi configurada para multiplicar duas matrizes 8x8 utilizando 32 *cores* e 1 bloco, ou CTA, de dimensões de 8x8 *threads*. Para os algoritmos de Autocorrelação e Ordenação, a GPGPU foi configurada para autocorrelacionar e ordenar, respectivamente, 8 inteiros utilizando 8 *cores* e um bloco de 1x8 *threads*. Por fim, a aplicação de Redução foi configurada para somar 16 inteiros utilizando 8 *cores* e um bloco de 1x8 *threads*. O processamento de pequenos conjuntos de dados de entrada foi preposto a fim de reduzir os tempos de simulação sendo que, para as configurações utilizadas, o tempo médio de cada simulação é de aproximadamente um minuto.

Os códigos CUDA foram compilados através do compilador NVIDIA CUDA (*nvcc*) e os códigos de máquina, SASS, demonstrados no Anexo A, foram obtidos através do processo descrito na Seção 3.1.3. A Tabela 2 mostra o tempo de execução para cada uma das aplicações, o tamanho dos códigos de programa e a quantidade de registradores utilizados por cada aplicação. Além disso, a tabela mostra a configuração referente ao número *cores* utilizado para a execução dos algoritmos.

Tabela 2 – Tempos de execução e recursos de *hardware* utilizados pelos algoritmos originais

Algoritmo	**Tempo (*ms*)**	**Memória (*bits*)**	**VRF**	**ARF**	**PRF**	***Cores***
Multiplicação	0,33	3200	11	-	1	32
Ordenação	0,33	2944	6	2	1	8
Autocorrelação	0,27	2114	8	-	1	8
Redução	0,13	2114	6	2	1	8

As Figuras 12, 13, 14 e 15 demonstram o comportamento dinâmico para os algoritmos de Ordenação, Autocorrelação, Multiplicação e Redução, respectivamente. O eixo horizontal do gráfico apresenta o tempo de execução do programa, enquanto o eixo vertical apresenta o número da instrução em execução dos respectivos códigos mostrados no Anexo A. No gráfico, também é apresentado o valor da máscara de *threads* (em hexadecimal entre colchetes) que indica quais as *threads* que estão habilitadas em determinados caminhos do programa. Com os dados dispostos no gráfico é possível constatar a dinâmica de controle das aplicações de estudo de caso, que envolve os desvios realizados bem como as divergências de execução dentro do *warp*. A máscara de instruções, que controla

a execução condicional de instruções pelas *threads* que fazem parte dos caminhos do programa, não é mostrada no gráfico mas é utilizada mais intensivamente pelos algoritmos de Ordenação e de Redução, que apresentam várias instruções de execução condicional.

Os gráficos demonstram que as aplicações de Ordenação e Autocorrelação são as que mais utilizam o caminho de controle, no que diz respeito ao controle de divergência de execução. A aplicação de Ordenação realiza vários desvios, bem como a execução condicional de instruções. A Autocorrelação demonstra um *loop* único que é executado diversas vezes por *threads* divergentes.

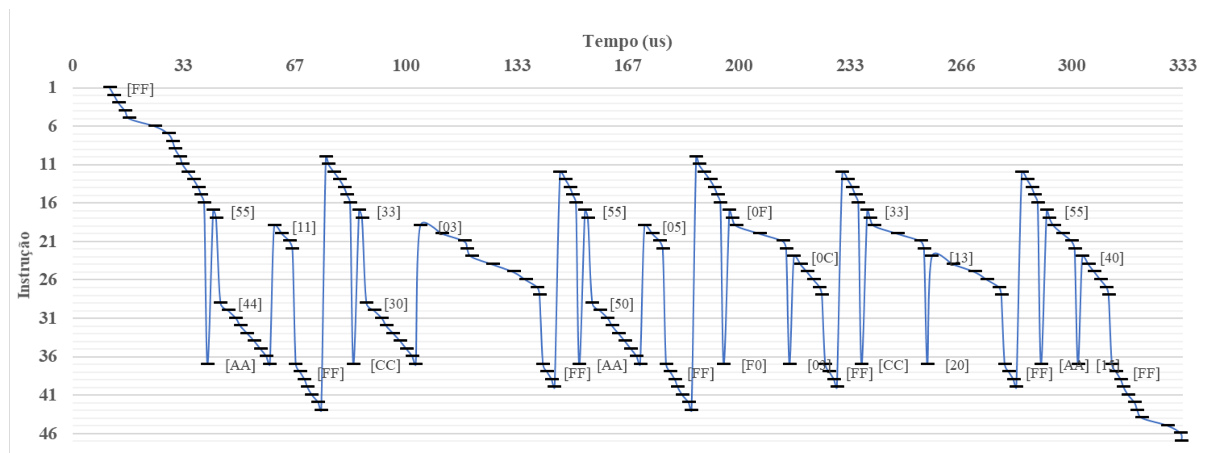

Figura 12 – Comportamento dinâmico do algoritmo de Ordenação de Vetores

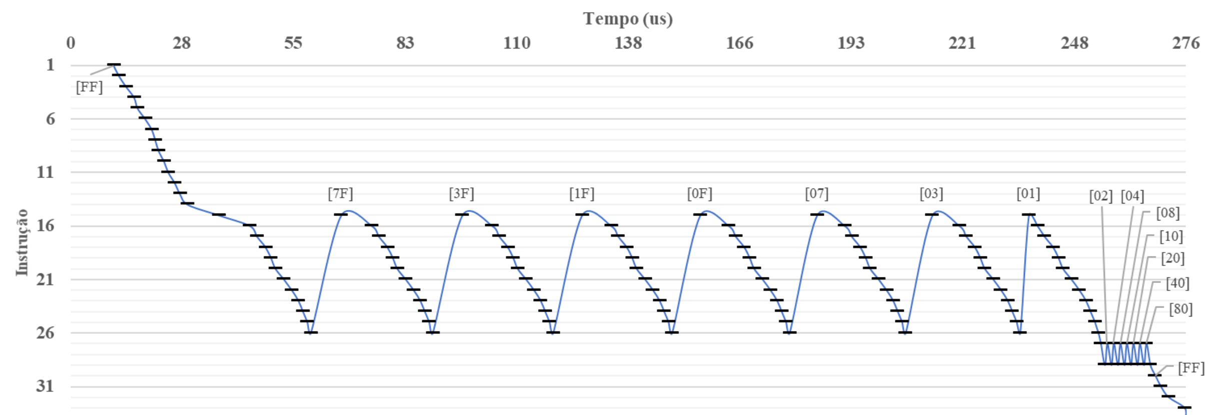

Figura 13 – Comportamento dinâmico do algoritmo de Autocorreção de Vetores

As aplicações de Multiplicação e de Redução, por outro lado, utilizam menos o caminho de controle. Na Multiplicação de Matrizes, pode-se observar que, embora o código *assembly* conte com instruções de sincronismo para controle de divergência, nenhuma divergência ocorre. Nesta aplicação, um *loop* pode ser observado, o qual é executado por todas as *threads* simultaneamente. O início de cada *loop* contém as instruções de acesso à memória global, por esta razão ocorre um maior distanciamento entre as instruções neste intervalo, considerando o eixo horizontal. A aplicação de Redução, assim como a

de Multiplicação, mantém uma mesma máscara de *threads* durante toda a execução, o que significa que nenhuma divergência ocorre. A Redução, no entanto, não conta com instruções de sincronismo e possui diversas instruções de execução condicional, as quais podem ser observadas no código do Anexo A.1.

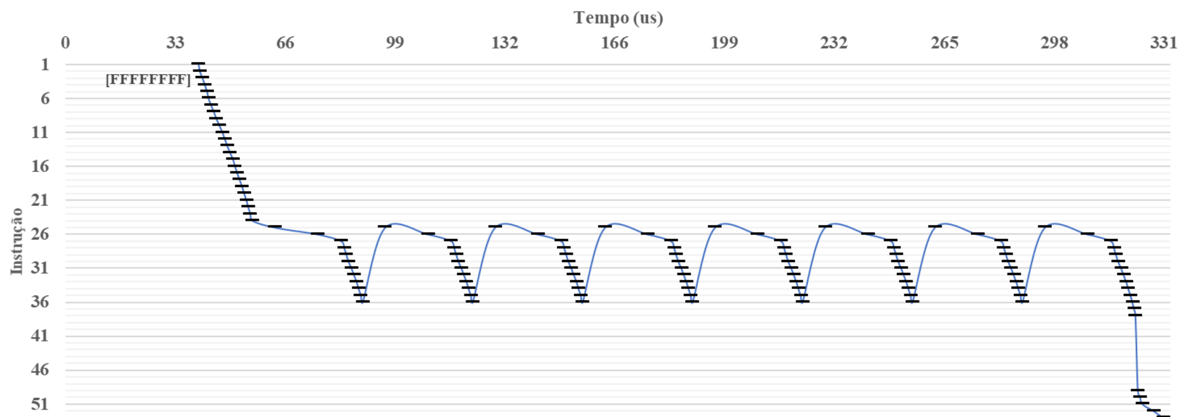

Figura 14 – Comportamento dinâmico do algoritmo de Multiplicação de Matrizes

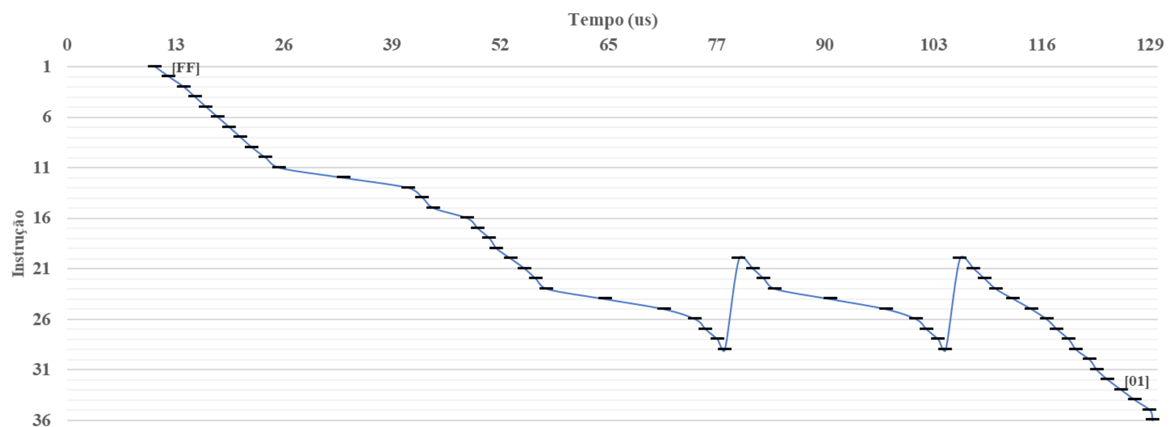

Figura 15 – Comportamento dinâmico do algoritmo de Redução de Vetores

3.3 Técnicas de Tolerância a Falhas

GPUs contam com a linguagem PTX como uma linguagem intermediária, o que possibilita a programaçao em alto nível de abstração para GPUs de diferentes capacidades computacionais. No entanto, esta linguagem é compilada para a linguagem SASS, a qual representa as instruções nativas do dispositivo, conforme descrito na Seção 3.1.3. A NVIDIA oferece uma documentação completa para trabalhar com a linguagem intermediária PTX, mas não disponibiliza nenhum tipo de documento com as especificidades da ISA dos diferentes modelos de GPUs, o que dificulta a programação em instruções nativas de baixo nível. Além disso, também não é disponibilizado um montador documentado para a

função de conversão do código SASS para o código binário *cubin*. A fim de contornar essas dificuldades, alguns trabalhos encontrados na literatura têm se dedicado à descoberta e ao entendimento da arquitetura desses dispositivos em nível de detalhes não documentado pelos fabricantes. Pode-se tomar como exemplos os trabalhos (LAAN, 2007) e (COLLANGE et al., 2010), que foram utilizados como base no desenvolvimento do estágio de decodificação do *pipeline* da FlexGrip (MERCHANT, 2013).

Isto representa um obstáculo aos projetistas interessados em desenvolver técnicas de tolerância a falhas em baixo nível de abstração, pois é necessário, primeiramente, conhecer a ISA dos dispositivos para, então, testar e automatizar a proteção dos códigos. Por essas razões, as técnicas propostas neste trabalho são implementadas manualmente, embora os estudos já tenham sido iniciados para automatizar o processo através da ferramenta *Hardening Post Compiling Translator* (AZAMBUJA et al., 2012) - que pode executar automaticamente todos os passos necessários aplicar as transformações sobre o código original - e dos projetos *asfermi* (HOU; LAI; MIKUSHIN, 2011) e *amdasm* (RÁK, 2011), que são montadores para dispositivos NVIDIA e AMD, respectivamente. Tais ferramentas permitirão automatizar o processo de proteção para a FlexGrip e para GPUs comerciais.

As técnicas de tolerância a falhas propostas por este trabalho são destinadas a proteger cada um dos bancos de registradores de uma GPU. Para isso, primeiro deve-se considerar a arquitetura de bancos de registradores de GPUs. A GPGPU definida como estudo de caso tem três tipos de bancos de registradores, que são utilizados para armazenar diferentes tipos de dados. Cada banco de registradores conta com diferentes instruções de suporte da respectiva ISA e, portanto, cada banco deve ser tratado com diferentes técnicas de proteção. A fim de proteger efetivamente os bancos de registradores da GPGPU de estudo de caso, três diferentes técnicas de tolerância a falhas baseadas em *software* são propostas.

De acordo com documentos técnicos da NVIDIA (NVIDIA, 2009), as arquiteturas de GPU, por padrão, não contam com Códigos de Correção de Erros (ECC). Entretanto, algumas têm a opção de implementar ECC de maneira externa (na memória) ou interna (na *cache* e nos registradores). O ECC, por sua vez, tem desvantagens em confiabilidade, custo em área de memória e até mesmo degradação do desempenho (PILLA et al., 2014). As técnicas propostas neste trabalho não são apenas uma alternativa ao ECC, mas também podem ser combinadas com seu uso, tanto interno quanto externo.

O conjunto de instruções CUDA suportado pela arquitetura FlexGrip, de capacidade computacional 1.0, é apresentado no Anexo B. Neste trabalho são consideradas apenas as instruções disponíveis para a FlexGrip para o desenvolvimento das técnicas de detecção de falhas mas em GPUs comerciais e mais modernas, a ISA disponibilizada pode oferecer maiores possibilidades para proteção das aplicações devido ao maior conjunto

instruções em relação ao do dispositivo de estudo de caso.

3.4 Campanha de Injeção de Falhas

A realização de campanhas de injeção de falhas permite analisar a sensibilidade de GPUs a SEUs, bem como validar as técnicas de tolerância a falhas desenvolvidas para proteger estes dispositivos contra efeitos indesejados. SEUs são falhas transientes que podem acontecer em qualquer elemento de memória, sendo que neste âmbito se incluem as memórias global, compartilhada e *cache* e os registradores. Portanto, o efeito de SEUs sobre todos estes componentes deve ser estudado.

Este trabalho tem como objetivo investigar os efeitos destas falhas apenas nos registradores ou, mais especificamente, nos três bancos de registradores e nos registradores de *pipeline* da *soft-core* GPGPU FlexGrip. Futuramente, pretende-se expandir este estudo para os demais elementos de memória da GPGPU.

A GPGPU de estudo de caso oferece a possibilidade de simular o efeito causado por falhas induzidas por partículas de radiação diretamente em elementos de *hardware* do sistema. Se fosse utilizada uma GPU comercial, isso não seria possível, ou teria-se que submeter o dispositivo a um acelerador de partículas, ou então simular falhas através de ferramentas de *debug*, por meio de métodos intrusivos, que interferem no fluxo natural da execução das instruções e que não possuem o alcance adequado para simular falhas na maior parte do *hardware* do dispositivo, conforme citado na Seção 2.4.

A FlexGrip também possibilita que as falhas sejam emuladas em um FPGA por meio de mecanismos de reconfiguração do *bitstream* e por meio da inserção de módulos de *hardware* no circuito com a finalidade de emular falhas no sistema. Além disso, o FPGA também pode ser submetido a aceleradores de partículas para testes com radiação. Estes testes são interessantes e devem ser explorados futuramente, mas em todos os casos é necessária a utilização de uma plataforma robusta para a comportar os recursos de *hardware* necessários para a sintetização da GPGPU em um FPGA, o que aumenta drasticamente o custo dos testes. A Tabela 3 demonstra uma estimativa de recursos demandados pela FlexGrip quando sintetizada em diferentes configurações para uma placa Virtex-6 XC6VLX240T-1FFG1156 (NEDEL, 2015).

Tabela 3 – Recursos de *hardware* demandados pela FlexGrip para sintetização em uma placa Virtex-6

Parâmetros	Frequência (MHz)	LUTs	flip-flops	BRAM	DSP48E
8 SP	100	71.323	103.776	120	156
16 SP	100	113.504	149.297	132	300
32 SP	100	231.436	240.230	156	588

Portanto, este trabalho também propõe um injetor de falhas para simular falhas

em Nível de Transferência entre Registradores (RTL) nesta GPGPU, que foi utilizado em campanhas de injeção de falhas para simular SEUs em todos os bancos de registradores, com os objetivos de analisar o comportamento da GPGPU e de validar as técnicas de tolerância a falhas propostas. Além disso, SEUs foram simulados nos registradores do *pipeline* da GPGPU com a finalidade de explorar os efeitos destas falhas neste componente - até então inexplorados na literatura com este nível de controle de injeção das falhas - como também de verificar se as técnicas de tolerância a falhas implementadas possuem algum potencial de redução de erros causados por falhas no *pipeline*.

4 Implementação

4.1 Técnicas de Tolerância a Falhas

GPUs e CPUs possuem arquiteturas muito diferentes entre si, mas os dois dispositivos executam suas tarefas através de um determinado conjunto de instruções. Um código *assembly*, resultante do processo de compilação CUDA, pode ser observado no Anexo 3.2. Este código representa as instruções de um *kernel*, ou seja, as instruções que serão executadas por todas as *threads* instanciadas na GPU. Dessa maneira, ao proteger este código, todas as *threads* serão protegidas. A divisão de registradores entre as *threads* é feita através do *hardware* da GPU, de acordo com os parâmetros obtidos da compilação do código de programa. Por esta razão, no código do programa, todas as *threads* chamam os registradores pelo mesmo nome, pois o *hardware* faz com que cada *thread* acesse seus respectivos registradores. Assim, para replicar os registradores de dados, é preciso configurar o número de registradores que serão utilizados por cada *thread*. Já o número de registradores de endereços e de predicado é fixo em quatro registradores por *thread*, conforme descrito anteriormente. É importante mencionar que, para criar réplicas em recursos que podem ser acessados por um conjunto de *threads*, como, por exemplo, memória compartilhada e memória global, é necessário que tais réplicas não sejam criadas sobre endereços de memória que são utilizados por outras *threads*.

É importante ressaltar que existe um limite de registradores disponibilizados por uma GPU, e que a distribuição desses registradores é feita de acordo com o número de registradores utilizado pela aplicação e pelo número de *threads* em uso, conforme descrito na Seção 3.1.2. Portanto, para aplicar as técnicas em *software* de tolerância a falhas, deve-se garantir que existem recursos disponíveis no sistema, e isso vai depender do número de registradores utilizado pela aplicação e da quantidade de *threads* utilizada. Se o número de registradores disponíveis for insuficiente para que todos os registradores originais sejam replicados, três possibilidades podem ser consideradas: (1) selecionar variáveis para realização de *register spilling*, processo no qual algumas variáveis temporárias deixam de ser armazenadas em registradores e passam a ser armazenadas em memória, assim liberando registradores, mas acarretando em grande perda de desempenho, visto que diversas instruções de acesso à memória são necessárias para chavear o contexto das variáveis, (2) reduzir o número de registradores em alto nível, reduzindo o número de *threads* ou diminuindo o número de registradores disponíveis ao compilador, e consequentemente acarretando em perda de desempenho e (3) realizar uma proteção seletiva de um sub-grupo de registradores, o que reduz a confiabilidade do sistema, mas não acarreta em perdas de desempenho (além do custo original das técnicas de tolerância a falhas).

A proteção seletiva, ou *selective hardening*, além de possibilitar a otimização de recursos de *hardware*, é uma abordagem que possibilita que as técnicas propostas sejam implementadas em diferentes graus de proteção, de modo que registradores específicos possam ser escolhidos para serem protegidos, o que possibilita uma troca entre confiabilidade e desempenho. Isto é uma grande vantagem das técnicas propostas pois a implementação em *assembly* permite o acesso direto aos registradores do sistema, o que proporciona uma proteção seletiva mais específica, quando comparado à implementação em alto nível.

A seguir serão apresentadas as práticas utilizadas para proteger os bancos de registradores de GPUs contra SEUs. As técnicas de tolerância a falhas são aplicadas sobre o código *assembly* SASS, apresentado na Seção 3.1.3, do *kernel* dos algoritmos. As transformações aplicadas sobre os códigos são baseadas nas técnicas Variáveis e Desvios Condicionais. Sendo que a primeira é aplicada para proteger os bancos VRF e ARF, enquanto a segunda é utilizada como base para proteger os bancos PRF.

4.1.1 Técnica de Detecção para VRF (VRF Hard)

VRF é o maior banco de registradores da GPUs, tanto em quantidade de registradores quanto em ocupação de memória, e compreende os registradores mais utilizados por uma aplicação pois a maior parte das instruções de um programa operam sobre esses registradores. Sendo assim, a probabilidade de ocorrência e propagação de SEUs em VRF será maior do que em PRF e ARF para a maioria das aplicações. Desta maneira, pode-se concluir que, entre os bancos de registradores de uma GPU, VRF é o banco mais importante a ser protegido.

A técnica utilizada como base para proteger o banco VRF é a Variáveis. Desta forma, são criadas cópias para todas as variáveis utilizadas e verificações de consistência entre a variável original e sua cópia são realizadas em duas ocasiões: (1) em instruções que acessam a memória e (2) em instruções de desvio. As Figuras 16, 17, 18, e 19 ilustram trechos de códigos em suas versões original e protegida. As Figuras cobrem instruções de operação, de acesso à memória (*load* e *store*) e de desvio (*branch*) em código *assembly*. O código original pode ser visto no lado esquerdo da Figura, enquanto o código protegido, com instruções adicionais em negrito, é mostrado à direita.

Para proteger instruções de operação, replica-se a instrução original, mas sobre os registradores replicados. Neste caso, não é necessário realizar a verificação de consistência, pois estas instruções não acessam a memória. A Figura 16 mostra que, para replicar a instrução original 1, que realiza a operação sobre o registrador R1, a instrução 2 é adicionada, porém esta é modificada para operar sobre a réplica de R1, ou seja, sobre o registrador R1'.

As instruções de *load* e *store* fazem acessos à memória e, portanto, é preciso que

Código Original	Código Protegido
1: IADD32I R1, R1, 1;	1: IADD32I R1, R1, 1;
	2: IADD32I R1', R1', 1;

Figura 16 – VRF Hard: transformações para instruções de operação

sejam realizadas verificações de consistência entre os registradores originais e suas réplicas. Para realizar estas transformações, duas instruções são inseridas: a primeira, chamada de ISET, para comparar as variáveis e setar uma *flag* em caso de discrepância, e a segunda, chamada BRA, para saltar para uma sub-rotina de erro, caso a *flag* indique que ocorreu discrepância. Estas transformações podem ser vistas na Figura 17, em que a instrução original 5, é uma instrução de *store* cujas variáveis R0 e R6 são verificadas antes de efetivar a movimentação dos dados.

Código Original	Código Protegido
1: GST [R0], R6;	1: GST [R0], R6;
	2: ISET.C1 R0, R0', NE;
	3: BRA C1.NE, ERROR;
	4: ISET.C1 R6, R6', NE;
	5: BRA C1.NE, ERROR;

Figura 17 – VRF Hard: transformações para instruções de *store*

O mesmo padrão pode ser visto na Figura 18, em que a instrução original 3 tem sua variável, armazenada no registrador 8, verificada através das instruções 1 e 2. A instrução GLD, ou *load*, realiza uma leitura da memória global e armazena o valor no registrador 3. De acordo com a técnica Variáveis, as variáveis armazenadas na memória também deveriam ser replicadas. No entanto, a fim de diminuir a degradação de desempenho que uma instrução extra de *load* causaria, esta instrução foi substituída por uma instrução *mov*, que pode ser observada na instrução 4. Dessa maneira, é introduzido um ponto de falha no sistema, porque se ocorrer uma falha durante a instrução de *load* sobre o registrador utilizado por esta instrução, tal falha não poderá ser detectada pois o valor incorreto que será carregado através da instrução *load* será copiado para o registrador réplica com a próxima instrução *mov*.

As transformações aplicadas para proteger instruções de desvio são mostradas na Figura 19. Apesar desse tipo de instrução não acessar diretamente a memória, os desvios condicionais se baseiam em comparações realizadas sobre variáveis. Portanto, é necessário proteger estas instruções pois um desvio incorreto poderia levar o programa a executar *loads* ou *stores* imprevistos, o que poderia corromper a memória, ou fazer com que uma *thread* perdesse o sincronismo com as demais, assim causando um erro no sistema. A

Código Original	**Código Protegido**
1: GLD R3, [R6];	1: GLD R3, [R6];
	2: MOV R3', R3;
	4: ISET.C1 R6, R6', NE;
	5: BRA C1.NE, ERROR;

Figura 18 – VRF Hard: transformações para instruções de *load*

Figura 19 ilustra as transformações para proteger uma instrução de desvio. A instrução original 1 realiza uma comparação entre os valores armazenados nos registradores R1 e R7 e armazena o resultado da comparação no registrador de predicado C0, enquanto a instrução original 6 realiza um salto, ou não, para o endereço DEST, de acordo com o valor C0, resultante da comparação. Por sua vez, as instruções 2, 3, 4 e 5 são utilizadas para verificar a consistência entre os registradores R1 e R7 com suas réplicas antes que a instrução de salto condicional seja executada. Caso ocorra discrepância entre as variáveis o programa é desviado para uma rotina de erro.

Código Original	**Código Protegido**
1: ISET.C0 R1, R7, NE;	1: ISET.C0 R1, R7, NE;
	2: ISET.C1 R1, R1', NE;
	3: BRA C1.NE, ERROR;
	4: ISET.C1 R7, R7', NE;
	5: BRA C1.NE, ERROR;
6: BRA C0.NE, DEST;	6: BRA C0.NE, DEST;

Figura 19 – VRF Hard: transformações para instruções de *branch*

Uma maneira de reduzir a quantidade de saltos que direcionam o programa para uma rotina de erro pode ser feita através do armazenamento do resultado das verificações de consistência e adiamento deste resultado para algum ponto estratégico do programa. Este ponto pode ser, por exemplo, antes de instruções de desvios condicionais do programa original, sendo que, assim, o programa pode ser dividido em blocos de execução que serão definidos pelas instruções de controle de fluxo. Tal organização pode vir a ser considerada, futuramente, na automatização da aplicação das técnicas.

A Figura 20 mostra um trecho do código da Multiplicação de Matrizes, referente ao laço observado no comportamento dinâmico do programa, em que as verificações de consistência são realizadas conforme as regras da técnica Variáveis, mas o teste que leva a *thread* para uma rotina de erro é feito somente no final do bloco, antes da instrução original de desvio condicional. As instruções originais são mostradas em negrito, enquanto suas transformações inseridas são mostradas abaixo.

Conforme pode ser observado na Figura 20, as instruções de acesso à memória e de controle de fluxo 1, 4 e 17 têm suas respectivas verificações de consistência nas linhas 3, 6, 18 e 19, enquanto o teste da *flag* C2, que indica se houve erro, somente é testada na penúltima instrução. Conforme pode ser observado, as instruções de verificação de consistência mudam um pouco em relação ao apresentado anteriormente. Neste caso, elas foram convertidas em instruções condicionais que também dependem do valor da *flag* de erro C2 (C2.EQ). Esta *flag* garante que as verificações de consistência somente serão executadas quando nenhum erro prévio foi detectado, o que impede que C2 seja sobrescrito, caso uma falha já tenha sido encontrada anteriormente. Desta maneira, neste código, foi possível utilizar apenas uma verificação da *flag* C2 para quatro verificações de consistência.

```
01: GLD.U32 R3, global14 [R8];
02: MOV R14, R3;
03: ISET.S32.C2 (C2.EQ) o [0x7f], R8, R19, NE;
04: GLD.U32 R4, global14 [R9];
05: MOV R15, R4;
06: ISET.S32.C2 (C2.EQ) o [0x7f], R9, R20, NE;
07: IMUL.U16.U16 R10, R3L, R4H;
08: IMUL.U16.U16 R21, R14L, R15H;
09: IMAD.U16 R10, R3H, R4L, R10;
10: IMAD.U16 R21, R14H, R15L, R21;
11: SHL R10, R10, 0x10;
12: SHL R21, R21, 0x10;
13: IADD32I R1, R1, 0x1;
14: IADD32I R12, R12, 0x1;
15: IMAD.U16 R3, R3L, R4L, R10;
16: IMAD.U16 R14, R14L, R15L, R21;
17: ISET.S32.C0 o [0x7f], R1, R7, NE;
18: ISET.S32.C2 (C2.EQ) o [0x7f], R1, R12, NE;
19: ISET.S32.C2 (C2.EQ) o [0x7f], R7, R18, NE;
20: IADD R6, R3, R6;
21: IADD R17, R14, R17;
22: IADD32I R9, R9, 0x4;
23: IADD32I R20, R20, 0x4;
24: IADD R8, R8, R2;
25: IADD R19, R19, R13;
26: BRA C2.NE, ERRO;
27: BRA C0.NE, 01;
```

Figura 20 – Exemplo de VRF Hard sobre o algoritmo de Multiplicação de Matrizes

A Tabela 4 mostra os tempos de execução e os recursos de memória de programa utilizados pela técnica VRF Hard quando aplicada sobre os algoritmos de estudo de caso. Os dados da tabela mostram que os tempos de execução são de 30% a 77% superiores aos tempos de execução dos algoritmos originais. A Multiplicação de Matrizes apresenta o menor aumento, devido ao percentual de acessos à memória global realizado (suas réplicas são substituídas por instruções *mov*, como mostrado na Figura 18, que são mais rápidas do que as instruções *load*), e o fato das instruções de *load* levarem os tempos de execução

mais longos para serem concluídas.

Tabela 4 – VRF Hard: desempenho, recursos de memória e registradores utilizados

Algoritmo	**Tempo (*ms*)**	**Memória (*bits*)**	**VRF**	**ARF**	**PRF**
Multiplicação	0,43 (1,30x)	6656 (2,08x)	22 (2x)	-	2 (+1)
Ordenação	0,55 (1,67x)	6144 (2,09x)	12 (2x)	2 (-)	2 (+1)
Autocorrelação	0,47 (1,74x)	4352 (2,06x)	16 (2x)	-	2 (+1)
Redução	0,23 (1,77x)	4544 (2,15x)	12 (2x)	2 (-)	2 (+1)

Em relação à ocupação da memória de programa, os resultados mostram que a quantidade de memória ocupada pelos algoritmos protegidos com a técnica VRF Hard supera o dobro da quantidade utilizada pelos algoritmos originais, variando de 106% a 115% em relação aos valores originais. Isso ocorre devido às instruções adicionais que são inseridas. Estes dados, entretanto, não representam um impacto real no sistema, uma vez que a memória de programa representa um pequeno percentual da memória e total do sistema, normalmente tem baixo custo e pode ser protegida através da utilização de tecnologias mais resilientes, tais como memórias FLASH ou EEPROM.

Ao considerar a quantidade de registradores, visto que todas as variáveis de VRF são replicadas, VRF Hard usa o dobro de registradores quando comparado com os algoritmos originais. As técnicas de tolerância a falhas propostas sempre utilizam um registrador de PRF para manter o resultado das verificações de consistência e, por esta razão PRF é acrescido em um registrador.

4.1.2 Técnica de Detecção para PRF (PRF Hard)

A maior parte das instruções de uma GPU pode ter sua execução condicionada por uma *flag* de predicado, enquanto apenas algumas instruções são capazes de gerar estas *flags*. Um exemplo de instrução que pode manipular *flags* de predicado é a ISET (*Integer Set*). Esta instrução compara dois valores inteiros e define o valor de um registrador de predicado (*flag*), de acordo com a comparação realizada. Entre os dois operandos da instrução, essa comparação pode ser Menor ou Igual (LE), Igual (EQ), Diferente (NE), Maior ou Igual (GE), Menor (LT) e Maior (GT). Cada *thread* conta com até quatro *flags* de predicado que, em código *assembly*, são representadas como C0, C1, C2 e C3.

Entre as instruções que podem ter sua execução determinada por *flags* de predicado, estão as instruções de desvio condicional. Portanto, se uma falha corromper um registrador de predicado, poderá ocorrer a indevida execução, ou a não execução, de instruções, além de indesejados desvios, ou não desvios. Desta maneira, uma falha neste banco de registradores poderá provocar erros de dados na saída do sistema e causar problemas de sincronismo entre as *threads* de um *warp*, pois uma *thread* que não possui instruções

de sincronismo, por não considerar erros de fluxo de execução, pode tomar um caminho indevido e não encontrar as instruções de retorno que faça com que o *warp* seja finalizado corretamente. Além disso, visto que cada *warp* comporta 32 *threads* que podem divergir em relação à execução de determinadas instruções, a predicação é utilizada para habilitar ou suprimir a execução de instruções por *thread* em sequências curtas de instruções, o que dispensa o uso desvios condicionais e, portanto, anula a sobrecarga do gerenciamento de *threads* que divergem em relação ao caminho de execução, como o demonstrado na Seção 3.1.1.4.

O conjunto de instruções *assembly* disponível para a GPGPU de estudo de caso não possibilita a comparação direta entre duas *flags* de predicado e, desse modo, não é possível comparar diretamente essas *flags* por meio da técnica Variáveis. Porém, é possível realizar uma comparação indireta entre duas *flags* através da técnica de Desvios Condicionais. As Figuras 21 e 22 mostram dois trechos de códigos utilizados para detectar falhas em PRF, enquanto a Tabela 5 mostra o tempo de execução e o consumo de memória resultantes da implementação de PRF Hard sobre os algoritmos de estudo de caso.

Código Original	**Código Protegido**
`1: ISET.C0 R1, R10, EQ;`	`1: ISET.C0 R1, R10, EQ;`
	`2: ISET.C0’ R1, R10, EQ;`
`3: BRA C0.NE, 8;`	`3: BRA C0.NE, 7;`
	`4: BRA C0’.NE, ERROR;`
`5: IADD R6, 2;`	`5: IADD R6, 2;`
	`6: BRA 8;`
	`7: BRA C0’.EQ, ERROR;`
`8: IADD R6, 1;`	`8: IADD R6, 1;`

Figura 21 – PRF Hard: transformações para instruções de desvio condicional

A Figura 21 mostra a aplicação da PRF Hard sobre as instruções de desvio condicional ISET e BRA, nas linhas 1 e 3. Inicialmente, a instrução adicional da linha 2 é utilizada para replicar a *flag* C0 em C0’. Em seguida, as instruções 4, 6 e 7 são adicionadas para replicar a instrução de desvio BRA. Para realizar a análise da aplicação da técnica, temos que considerar os dois possíveis fluxos de controle que podem ser tomados após a execução da instrução de desvio, que são: (1) o desvio não é tomado, e portanto a instrução seguinte a ser executada é a da linha 4 e (2) o desvio é tomado, e portanto a próxima instrução executada é a da linha 7.

No primeiro caso, onde o desvio não é tomado, uma réplica da instrução original, porém sobre a *flag* replicada, também não deveria desviar o fluxo de controle. Sendo assim, a PRF Hard apenas replica a instrução original sobre a *flag* replicada, porém, ao invés

de ter como destino o destino da instrução original, adiciona o endereço da sub-rotina de tratamento de erro. Desta maneira, a instrução replicada da linha 4 apenas é executada quando a instrução original da linha 3 não toma o desvio condicional, e, portanto, esta também só toma o desvio em caso de falha.

No segundo caso, em que o desvio da instrução original é tomado, sua réplica sobre a *flag* replicada também deveria desviar o fluxo em caso de correto funcionamento. Desta maneira, a fim de detectar um erro através de um salto para a sub-rotina de tratamento de erro, a instrução replicada é invertida e seu destino é apontado para a sub-rotina de erro. Por esta razão, a instrução da linha 7, além de operar sobre a réplica da *flag* orignal C0, tem sua comparação invertida, de C0.NE para C0'.EQ, e como destino do desvio a sub-rotina de tratamento d erro.

Por fim, a instrução da linha 8 é adicionada para evitar que qualquer outro fluxo de controle execute a instrução adicional da linha 7. É importante reparar que em aplicações reais, diversas outras instruções podem aparecer entre as instruções das linhas 5 e 6 e, portanto, é importante que a instrução replicada da linha 7 apenas seja executada após a instrução da linha 4, assim evitando falsas detecções de falhas.

Em casos de desvio condicional, como apresentado na Figura 21, a capacidade de detecção de falhas desta técnica é reduzida para GPUs, quando comparado com GPPs. Isso é esperado porque GPUs processam múltiplas *threads* agrupadas em *warps*. Assim, considerando um exemplo em que todas as *threads* de um *warp* devessem realizar um desvio, mas uma falha em PRF faz com que apenas uma *thread* não realize este desvio, podem ocorrer duas situações: (1) se o sistema priorizar à execução da *thread* afetada, o erro será detectado, visto que, na técnica de Desvios Condicionais, o desvio que a leva para a rotina de tratamento de erro será a próxima instrução a ser executada e (2) se o sistema priorizar a execução das *threads* remanescentes, a *thread* afetada poderá não ser retomada pelo sistema, pois ela não conta com instruções de sincronismo que garantem o seu retorno. A forma mais simples de resolver esta questão seria acrescentar as instruções de sincronismo para retomada da *thread* afetada pela falha, porém, devido à pequena probabilidade de ocorrência de falhas no PRF quando comparado aos outros bancos, e ao alto custo de desempenho que essas instruções resultariam, optou-se por não adicioná-las.

Na Figura 22, à esquerda, pode-se observar um trecho de código em que não ocorrem saltos, mas apresenta instruções (linhas 3 e 4) que têm suas execuções condicionadas aos valores de *flags* de predicado. As instruções 3 e 4 somente são executadas se o valor de C0, resultado da instrução 1, for falso. Portanto, é preciso que a *flag* C0 seja protegida para garantir que uma falha no sistema não se propague a partir da indevida execução, ou não execução, das instruções 3 e 4. As transformações aplicadas para proteger este código podem ser observadas nas linhas 2, 5, 6, 7 e 8. Na instrução 2, uma réplica do valor de C0 é criada em C0'. Após as instruções 3 e 4 serem executadas, é realizada a

Código Original	Código Protegido
1: ISET.C0 R1, R10, EQ;	1: ISET.C0 R1, R10, EQ; **2: ISET.C0’ R1, R10, EQ;**
3: R2A A1 (C0.EQ), R3; 4: MOV R1 (C0.EQ), R12;	3: R2A A1 (C0.EQ), R3; 4: MOV R1 (C0.EQ), R12;
	5: BRA C0.NE, 8; **6: BRA C0’.NE, ERROR;** **7: BRA 9;** **8: BRA C0’.EQ, ERROR;**

Figura 22 – PRF Hard: transformações para instruções de execução condicional

verificação de consistência através das linhas 5, 6, 7 e 8. Quando o valor de C0 for verdadeiro na instrução 5, ocorrerá um salto para a instrução 8, que possui o teste inverso sobre a réplica C0’. Se nesta instrução C0’ for divergente de C0, o programa é desviado para uma rotina de indicação de erro. Se na instrução 5 não ocorrer o desvio, o programa segue para a instrução 6, que contém o mesmo teste sobre a réplica. Se nesta instrução C0’ divergir de C0, o programa é desviado para a sub-rotina de tratamento de erro. Se não houver divergência, a próxima instrução, na linha 7, contorna o teste da linha 8 e desvia o programa para o seu fluxo original.

Tabela 5 – PRF Hard: desempenho, recursos de memória e registradores utilizados

Algoritmo	**Tempo (*ms*)**	**Memória (*bits*)**	**VRF**	**ARF**	**PRF**
Multiplicação	0,34 (1,03x)	3840 (1,20x)	11 (-)	-	3 (2x+1)
Ordenação	0,45 (1,36x)	4352 (1,48x)	6 (-)	2 (-)	3 (2x+1)
Autocorrelação	0,31 (1,15x)	2880 (1,36x)	8 (-)	-	3 (2x+1)
Redução	0,16 (1,23x)	3072 (1,45x)	6 (-)	2 (-)	3 (2x+1)

A Tabela 5 demonstra que o tempo de execução para a proteção de PRF é proporcional ao número de instruções de desvios condicionais e de outras instruções que dependem de *flags* de predicado para serem executadas. Por esta razão, os dois algoritmos Bitonic Sort e Redução apresentaram os maiores custos em tempo de execução, variando de 23% a 36% em comparação ao tempo dos algoritmos originais. Em, relação à ocupação de memória de programa, PRF Hard utilizou de 20% a 48% mais memória que os algoritmos originais.

A aplicação da PRF Hard requer o dobro da quantidade de registradores originalmente utilizados pela aplicação, além de um registrador para a sinalização da falha. Embora este custo possa parecer alto para proteger um banco de registradores que disponibiliza apenas 4 registradores por *thread*, normalmente não é. Os algoritmos utilizam

poucos registradores de predicado, visto que o intervalo entre escrita e leitura desses registradores geralmente é pequeno, o que permite que PRF seja facilmente reutilizado. Portanto, a PRF Hard deve ser capaz de lidar com a maioria dos algoritmos. Entretanto, nos casos onde o número de registradores de PRF for insuficiente, ainda seria possível utilizar registradores do VRF e aplicar a mesma abordagem utilizada em ARF Hard, que será descrita a seguir.

4.1.3 Técnica de Detecção para ARF (ARF Hard)

O principal objetivo em proteger o ARF é prevenir a GPU de acessar endereços de memória incorretamente através de instruções de *load* e *store*. Tais acessos podem fazer com que o sistema corrompa a memória e cause erros que nem mesmo técnicas como ECC sejam capazes de detectar ou corrigir. A fim de proteger a GPU, a técnica aplicada utiliza o mesmo conceito da técnica de proteção aplicada à VRF.

Assim como na técnica de proteção de VRF, o método aplicado à ARF replica os registradores de endereço de ARF e realiza as verificações de consistência dos dados armazenados entre os registradores originais e protegidos. As principais diferenças estão em dois aspectos que surgem ao lidar com ARF: (1) a ISA da FlexGrip não apresenta instruções que dão suporte para comparar diretamente registradores de ARF, e (2) o número de registradores de ARF é limitado, sendo disponibilizado um número fixo de apenas quatro registradores por *thread*.

A fim de solucionar estes dois aspectos, ao invés de usar o banco de registradores ARF para armazenar os dados replicado, a técnica ARF Hard utiliza registradores de VRF. Desta maneira é possível ter mais registradores disponíveis à um pequeno custo de ocupação, uma vez que VRF possui mais registradores do que ARF, bem como instruções que permitem comparar registradores diretamente. O principal inconveniente, por outro lado, é a degradação de desempenho e ocupação de memória de programa causados pelas instruções extras de movimentação de dados entre ARF e VRF.

As transformações aplicadas ao código original consideram duas operações: (1) quando uma instrução transfere dados de um registrador de VRF para um registrador de ARF ou, em outras palavras, quando um registrador de ARF é escrito, e (2) quando uma instrução de acesso à memória lê o valor de um registrador de ARF para endereçar um elemento da memória global ou compartilhada. As Figuras 23 e 24 mostram as transformações utilizadas para proteger estas operações, enquanto a Tabela 6 mostra o tempo de execução e o consumo de memória resultantes da implementação de ARF Hard sobre os algoritmos de estudo de caso.

No primeiro caso, a replicação do registrador de ARF é muito simples, uma vez que o valor original é armazenado em VRF. Como pode ser observado na Figura 23, a

instrução 1 move o dado do registrador R3 de VRF, com deslocamento à esquerda (*shift left*) em 0x2, para o registrador A1 de ARF, enquanto a instrução 2 cria uma réplica de A1 no registrador R12 de VRF, ao mover R3, deslocado à esquerda em 0x2, para R12.

Código Original	Código Protegido
`1: R2A A1, R3, 0x2;`	`1: R2A A1, R3, 0x2;`
	`2: SHL R12, R3, 0x2;`

Figura 23 – ARF Hard: transformações para operação de leitura

O segundo caso, em que registradores de ARF são lidos, é um pouco mais complicado. Como todas as operações de leitura de registradores do ARF ocasionam num acesso à memória, sempre é preciso realizar checagens de consistência entre o registrador no ARF com sua réplica armazenada no VRF. Para fazer isso, primeiro é necessário mover o dado contido no registrador de ARF para um registrador de VRF, para então realizar a checagem de consistência e notificar o *host* em caso de discrepância. A Figura 24 mostra um exemplo do código de programa protegido que considera este caso. A instrução original 1 carrega um dado da memória compartilhada em R0 através do endereçamento realizado pelo valor contido em A1 adicionado do valor 0x8. A instrução adicional 2 move o dado do registrador A1, do ARF, para um registrador temporário de VRF, R14, enquanto as instruções 3 e 4 realizam a verificação de consistência e entre o registrador temporário R14 e a réplica de A1, R12.

Código Original	Código Protegido
`1: MOV R0, g [A1+0x8];`	`1: MOV R0, g [A1+0x8];`
	`2: A2R R14, A1;`
	`3: ISET.C1 R14, R12, NE;`
	`4: BRA C1.NE, ERROR;`

Figura 24 – ARF Hard: transformações para operação de escrita

Tabela 6 – ARF Hard: desempenho, recursos de memória e registradores utilizados

Algoritmo	Tempo (*ms*)	Memória (*bits*)	VRF	ARF	PRF
Ordenação	0,52 (1,57x)	4544 (1,54x)	8 (1,3x)	2 (-)	2 (+1)
Redução	0,16 (1,23x)	2944 (1,39x)	8 (1,3x)	2 (-)	2 (+1)

Apenas duas das aplicações de estudo de caso utilizam ARF, cujos resultados são mostrados na Tabela 6. Tais resultados indicam que o custo em desempenho de ARF Hard é proporcional à utilização dinâmica de registradores de ARF, que é maior no Bitonic Sort do que na Redução, demonstrando um tempo de execução de 23% a 57% superior ao

tempo dos algoritmos originais. É interessante observar que o *overhead* entre diferentes aplicações pode ser maior que o dobro. Em relação ao consumo de memória de programa, ARF Hard utilizou de 39% a 54% mais memória do que a quantidade de utilizada pelos algoritmos originais.

Ao considerar a utilização de registradores da técnica ARF Hard, a quantidade de registradores de ARF permanece inalterada, visto que suas réplicas são armazenadas no VRF. Entretanto, o VRF tem uma ocupação extra igual ao número de registradores de ARF. A Tabela 6 demonstra um aumento em VRF de 30% em relação aos valores originais pois os algoritmos desprotegidos fazem uso de 2 registradores de ARF e 6 registradores de VRF. Portanto, nestes casos, a ocupação de VRF é aumentada em 30%. O PRF, por sua vez, apenas é acrescido de um registrador devido à flag de indicação de erro.

É importante ressaltar que o objetivo das técnicas apresentadas é a detecção de falhas no sistema, que é o primeiro passo em qualquer técnica de tolerância à falhas. A estratégia final de proteção ou correção das falhas dependerá do projeto. As técnicas aplicadas para detecção seguem as regras de transformações mostradas no capítulo 2.3 em que, uma vez detectada uma falha, o código é direcionado para uma sub-rotina de tratamento de erro através de uma instrução de salto. Em um FPGA, o tratamento deste erro pode ser feito de diversas maneiras, até mesmo por meio da inclusão de mecanismos de *hardware* para lidar com a falha. Em uma GPU comercial pode-se, por exemplo, utilizar a instrução *trap*, que aborta a execução do *kernel* e gera uma interrupção para o *host*, no lugar da instrução de desvio. Para estratégias de proteção que tratam o erro diretamente na GPU, é necessário que sejam consideradas instruções de sincronismo para cada bloco do código, como o demonstrado na figura 20, a fim de garantir que as *threads* remanescentes sejam retomadas após o tratamento da *thread* errônea.

4.2 Injetor de Falhas (GPUFI)

Para simular falhas na FlexGrip, foi desenvolvido um *script* em linguagem *Tool Command Language* (TCL), denominado GPUFI (GPU *Fault Injector*), que modela um injetor de falhas que reproduz SEUs nos bancos de registradores e nos registradores de *pipeline* da GPGPU FlexGrip, durante a execução de uma aplicação no simulador ModelSim (GRAPHICS, 2005).

Uma falha do tipo SEU representa um *bit-flip* em um elemento de memória. Assim, o injetor de falhas deve selecionar um registrador (VRF, PRF, ARF ou registrador do *pipeline*) e alterar o estado de um de seus *bits* durante a execução de um algoritmo qualquer. Além de injetar falhas, GPUFI também possui uma função para geração dos resultados *golden* (resultados sem falhas) que pode ser utilizada antes da etapa de injeção de falhas, propriamente dita. Ao ser executada durante a simulação de uma aplicação na

FlexGrip, esta função monitora uma série de sinais da GPGPU e realiza uma coleta de dados, os quais são utilizados pelo algoritmo injetor de falhas, gerando como saída um arquivo que contém informações sobre os parâmetros da aplicação executada como, por exemplo, tempo de execução, resultado esperado, parâmetros de configuração da GPGPU e da aplicação. Por sua vez, o algoritmo injetor utiliza essas informações para injetar falhas em um registrador de um banco de registradores ou em um registrador do *pipeline*, de acordo com a configuração do usuário.

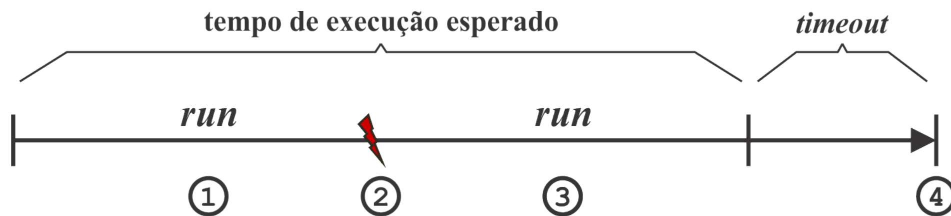

Figura 25 – Passos para injeção de uma falha

O processo de injeção ocorre em quatro passos, que estão apontados na Figura 25. No primeiro passo, a simulação é executada através do comando *run*, disponibilizado pelo simulador, até um momento aleatório dentro do tempo de execução da aplicação, determinado pela função *rand()*. No segundo passo, a falha é injetada, ou seja, um *bit* arbitrário é selecionado e tem seu valor invertido. Depois da injeção de uma falha, no quarto passo, a simulação é continuada através do comando *run* até concluir a execução da aplicação, conforme o tempo esperado acrescido de um tempo de *timeout*. Este *timeout* é necessário para garantir que a aplicação seja concluída e possibilitar a verificação de um erros que fazem com que a aplicação entre em laço infinito. Por fim, no quarto passo, o algoritmo injetor de falhas faz uma comparação entre o resultado obtido e o resultado esperado e, então, define o tipo de falha classificando-a de acordo com efeito que ela causa no sistema. Então, a execução é concluída e os resultados da injeção de falhas são gerados em um arquivo de saída do tipo *Comma-Separated Values* (CSV).

O modo como um *bit* é selecionado e invertido é diferente para os bancos de registradores e para os registradores do *pipeline* devido à diferença de implementação desses componente. A seguir são descritos o modo como os SEUs são simulados em cada um desses componentes, bem como a classificação dos efeitos que as falhas causam nesses componentes.

4.2.1 SEUs nos Bancos de Registradores

Para arbitrar um *bit* de um banco de registradores (ARF, PRF ou VRF), o algoritmo ampara-se nos resultados *golden* e na sistemática de distribuição de registradores utilizada pela FlexGrip, abordada na Seção 3.1.2, para selecionar aleatoriamente (1) uma *thread*, (2) um registrador utilizado por esta *thread* e (3) um *bit* deste registrador. A alea-

toriedade das operações é determinada pela função *rand()*, disponibilizada pela linguagem TCL. Para efetuar o *bit-flip* sobre o *bit* selecionado, o algoritmo utiliza o comando *change*, que permite que um valor seja escrito em um módulo de memória em qualquer momento da simulação. Uma vez que os bancos de registradores são implementados em memória, este comando é utilizado. A Figura 26 demonstra um exemplo de injeção de falhas para a aplicação de Multiplicação de Matrizes.

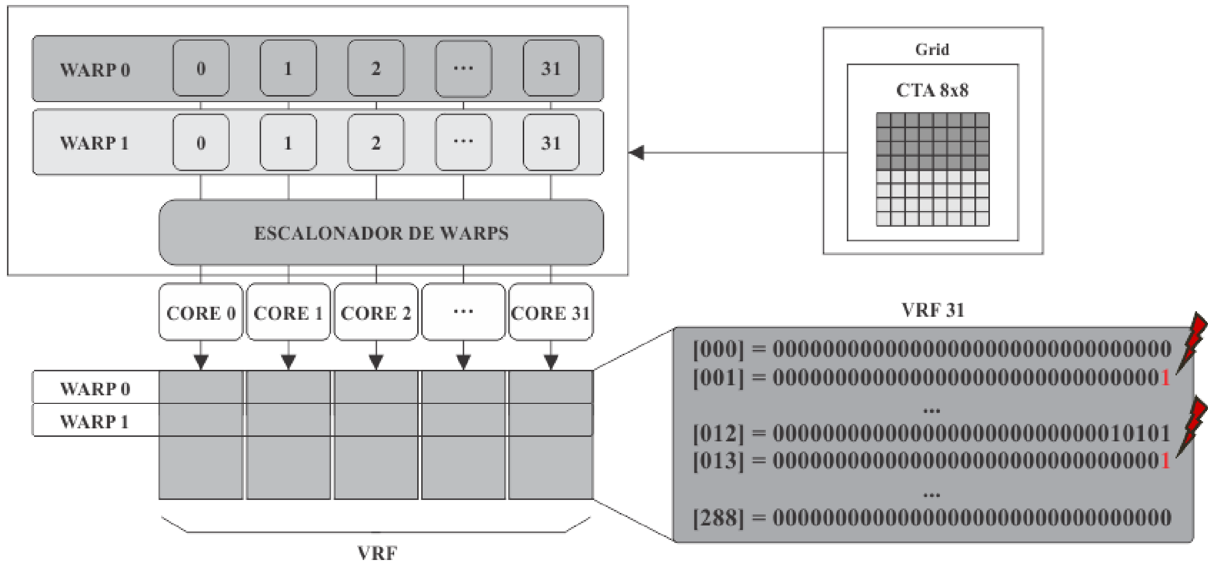

Figura 26 – Exemplo de simulação de um SEU em um banco de registradores

Conforme descrito na na Seção 3.2, para o processamento da Multiplicação de Matrizes, a FlexGrip foi configurada para utilizar 32 cores - o que resulta em *warps* de 32 colunas e apenas uma linha - utilizando um bloco de 8x8, o que resulta em 64 *threads*. O bloco é, então, dividido em 2 *warps* na unidade de *warp*, sendo que o *warp* 0 contém as *threads* de 0 a 31 enquanto o *warp* 1 contém as *threads* de 32 a 63. O exemplo da Figura 26 demonstra uma situação em que duas falhas ocorrem nos registradores 1 e 13 do banco VRF do *core* 31. Uma vez que o algoritmo de Multiplicação de Matrizes utiliza 11 registradores, a falha no registrador 1 afeta a *thread* 31 enquanto a falha no registrador 13 de VRF atinge o registrador 2 da *thread* 63. Isso ocorre porque as *threads* 31 e 63 compartilham o mesmo banco, mas há um *offset* entre seus registradores igual ao número de registradores utilizado pelo algoritmo, que, neste caso, é igual a 11.

O tipo de banco de registradores (VRF, ARF ou PRF) no qual se deseja aplicar as falhas deve ser determinado pelo usuário antes da execução do *script* GPUFI. A faixa de registradores onde será injetada uma falha também pode ser definida pelo usuário. Isso é útil, pois permite limitar os registradores de um banco onde as falhas deverão ser injetadas, o que é interessante para simulações que exigem grandes tempos de simulação. Dessa forma, pode-se, por exemplo, excluir registradores que não são utilizados pela aplicação a fim de acelerar campanhas de injeção de falhas.

4.2.2 SEUs nos Registradores do Pipeline

Para selecionar o *bit* que deve ser invertido, o injetor utiliza uma lista que contém os nomes dos registradores do *pipeline* e calcula a razão de probabilidade de ocorrência de uma falha para cada um dos sinais que compõem esses registradores, considerando o número de *bits* de cada registrador, onde registradores maiores têm probabilidade linearmente maior do que registradores menores. A arbitrariedade para seleção do sinal que receberá o SEU é determinada pela função *rand()*. Na Tabela 7 pode-se observar o tamanho dos registradores que compõem o *pipeline*. A coluna *estágio* indica os blocos de registradores que conectam dois estágios do *pipeline* enquanto as outras colunas indicam o número de registradores e de *bits* por bloco. Cada um desses blocos de registradores conta com um sinal *enable* que habilita a transferência dos dados da entrada para a saída do bloco.

Tabela 7 – Tamanho dos blocos de registradores de pipeline

Estágio	Registradores	Total em Bits
Warp to Fetch	9	139
Fetch to Decode	11	235
Decode to Read	67	504
Read to Execute	42	321
Execute to Write	27	305
Write to Warp	9	132
Total	165	1636

Para inverter o valor do *bit* selecionado, o processo é um pouco mais complicado do que o utilizado para os bancos de registradores. Isso ocorre pois, diferente dos bancos de registradores - que são implementados em módulos de memória, o que possibilita a utilização do comando *change* - os registradores do *pipeline* são implementados em processos HDL.

Uma maneira de modificar o valor de um sinal HDL durante uma simulação no ModelSim é utilizando o comando *force*. Este comando permite que um sinal tenha seu valor alterado durante um tempo determinado pelo usuário. Após este tempo, o estímulo aplicado é cancelado e o valor é retomado ao original. Logo, para simular um *bit-flip* em um registrador através do comando *force*, é necessário que o valor do *bit* seja alterado em momentos específicos que garantam a propagação do sinal da entrada para a saída do registrador.

Deste modo, para simular um SEU em registradores do *pipeline*, o algoritmo GPUFI monitora o sinal de *enable* dos blocos de registradores que conectam dois estágios do *pipeline*. Quando o momento de uma falha ocorre, primeiramente é verificado a qual bloco pertence o *bit* que deve ser afetado, e então a simulação é continuada até que o

enable de seu respectivo bloco esteja em nível alto. Quando o sinal de *enable* é ativado, o *bit* é forçado, através do comando *force* para o seu valor complementar durante o período de um *clock* e meio, assim garantindo que a falha seja registrada pelo circuito sequencial e seja mantida até que este valor seja sobrescrito ou propagado no *pipeline*.

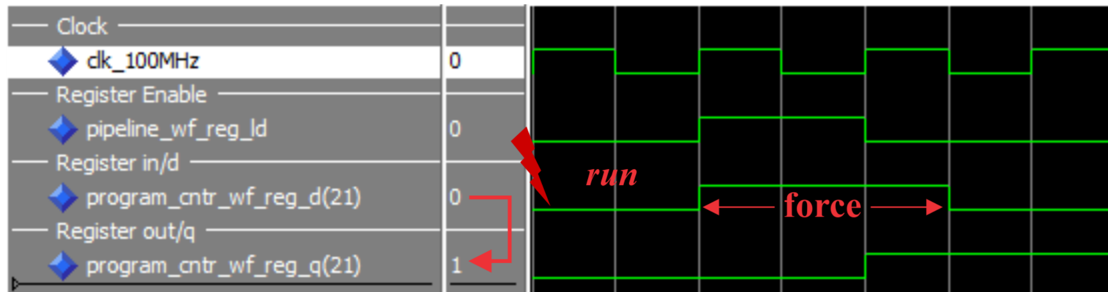

Figura 27 – Exemplo de simulação de um SEU em um registrador do *pipeline*

A Figura 27 demonstra um exemplo deste processo em que uma falha incide sobre o sinal 21 do registrador *program_cntr_wf_reg_d* e se propaga para a sua respectiva saída *program_cntr_wf_reg_q*, conforme indicado pela seta na figura. No nome do sinal, *d* e *q* indicam, respectivamente, entrada e saída, enquanto *wf* indica que o registrador pertence ao bloco *Warp to Fetch*, cujo sinal de *enable* pode ser observado na figura com o nome de *pipeline_wf_reg_ld*.

Nessas simulações, o usuário pode direcionar as falhas para qualquer um dos blocos de registradores entre dois estágios do *pipeline*. Isso é útil para analisar a suscetibilidade de estágios separadamente e para agilizar campanhas de injeção, direcionando-as para um ou mais blocos de registradores de interesse.

4.2.2.1 Classificação de Falhas

A fim de avaliar a confiabilidade do sistema, as falhas são classificadas de acordo com os efeitos que elas causam no sistema. Quando o programa termina sua execução normalmente e apresenta os resultados esperados, a falha é classificada como *unnecessary for Architecturally Correct Execution* (unACE). Em todos os outros casos, é classificada como *Architecturally Correct Execution* (ACE). Dentre as falhas do tipo ACE, duas classificações ainda são encontradas: (1) quando o programa termina sua execução normalmente, mas apresenta um resultado errôneo, a falha é classificada como *Silent Data Corruption* (SDC) e (2) quando uma falha faz com que o programa seja encerrado anormalmente ou faz com que o programa entre em laço infinito, ela é classificada como Hang.

Sendo assim, após a execução da aplicação, o algoritmo injetor de falhas faz uma comparação entre o resultado obtido e o resultado esperado e, então, define o tipo de falha classificando-a como (1) unACE, quando os resultados são iguais, (2) SDC, quando os resultados são diferentes, (3) Hang, quando a máquina de estados que controla o anda-

mento da aplicação na FlexGrip continua em estado de execução após o tempo de *timeout* e (4) Detected, o qual representa que houve a detecção de uma falha através da técnica de proteção aplicada sobre o algoritmo simulado.

É importante ressaltar que existe distinção entre falha e erro, uma vez que uma falha pode ou não causar um erro no sistema. SDC e Hang são considerados efeitos indesejados e devem ser evitados. Neste trabalho, tais efeitos serão considerados erros. O objetivo deste trabalho é detectar falhas a fim de prevenir a ocorrência de erros.

5 Resultados e Análise

Neste capítulo são apresentados os resultados obtidos a partir da execução das campanhas de injeção de falhas sobre a *Soft-Core* GPGPU FlexGrip e sua análise.

5.1 Campanhas de Injeção de Falhas

A fim de validar a suscetibilidade de GPUs a SEUs e a eficácia das técnicas de detecção de falhas, foram executadas duas distintas campanhas de injeção de falhas sobre a GPGPU FlexGrip através de simulação em nível RTL no simulador ModelSim SE 10.1c. Primeiramente, as falhas foram injetadas sobre os três bancos de registradores da FlexGrip. Após, as falhas foram injetadas sobre os registradores do *pipeline* da GPGPU. As campanhas foram realizadas automaticamente por meio do injetor GPUFI, e cada uma delas foi realizada em duas etapas. Na primeira, as falhas foram injetadas durante a execução dos algoritmos de estudo de caso originais, com o objetivo de verificar a suscetibilidade da GPGPU a SEUs. Na segunda etapa, as falhas foram injetadas durante a execução das aplicações protegidas, para verificar a eficácia das técnicas de proteção. Para cada aplicação de caso de uso, foi injetada uma falha por execução, num total de 10.000 falhas.

Visando uma análise completa sobre as técnicas propostas, as falhas foram injetadas em cada um dos três bancos de registradores, bem como nos registradores do *pipeline*, durante a execução das aplicações protegidas com VRF Hard, PRF Hard e ARF Hard. Desta maneira, pôde-se verificar se uma técnica de tolerância, desenvolvida para detectar falhas em um banco de registradores específico, é capaz de identificar falhas originadas nos outros bancos. Ademais, pode-se constatar se as técnicas desenvolvidas para detectar falhas nos bancos de registradores de GPUs possuem algum potencial de redução de erros causados por falhas no *pipeline*.

Para os bancos de registradores, as falhas foram injetadas somente nos registradores endereçados pelo compilador, ou seja, os registradores não escritos ou lidos pela aplicação foram removidos das campanhas de injeção de falhas. Deste modo, ao não injetar falhas em registradores não utilizados, é possível acelerar o processo de injeção.

É importante ressaltar que a quantidade de *bits* de cada banco de registradores e dos registradores do *pipeline* é diferente, o que significa que, embora a quantidade absoluta de falhas tenha sido exatamente a mesma para os três bancos e para o *pipeline*, os efeitos da radiação em aplicações reais tendem a aparecer em maior quantidade no VRF, seguido do ARF, PRF e *pipeline*, em uma relação linear com seus tamanhos absolutos de

aproximadamente 72/24/3/1, respectivamente.

5.1.1 Banco de Registradores de Dados

Nesta campanha de injeção de falhas, as falhas foram injetadas apenas no VRF. A Tabela 8 mostra o resultado das injeções para as aplicações de estudo de caso originais, enquanto a Tabela 9 apresenta os resultados para VRF Hard e a tabela 16 apresenta os resultados das injeções para as técnicas PRF Hard e ARF Hard. Os gráficos das Figuras 28, 30, 32 e 34 mostram a distribuição de falhas sobre os registradores das aplicações e o número de erros causados no sistema devido às falhas em cada um destes registradores enquanto os gráficos das Figuras 29, 31, 33 e 34 mostram a suscetibilidade a erros de acordo com intervalo de instruções executadas.

Conforme pode ser observado na Tabela 8, de todas as falhas injetadas, 71,9% resultaram em unACE, o que significa que elas não causaram erro no resultado final da aplicação. Por outro lado, os resultados demonstram que 28,1% das falhas injetadas causaram erros no sistema (ACE), sendo que 15,1% das falhas resultaram em SDC e 13,0% resultaram em Hang.

Tabela 8 – Injeções em VRF - resultados para os algoritmos originais

Algoritmo	**Detected**	**unACE**	**SDC**	**Hang**	***ACE***
Multiplicação	-	70,8%	17,9%	11,3%	29,2%
Ordenação	-	72,6%	6,2%	21,2%	27,4%
Autocorrelação	-	72,5%	21,4%	6,1%	27,5%
Redução	-	71,6%	14,8%	13,5%	28,4%
Total	**-**	**71,9%**	**15,1%**	**13,0%**	**28,1%**

Na Multiplicação de Matrizes, que resultou em 29,2% de ACEs, os Hangs ocorreram apenas por falhas nos registradores R1 e R7, no intervalo entre as instruções 25 e 37, como pode ser observado nos gráficos 28 e 29, respectivamente. Neste intervalo, a *flag C0* que determina o controle do laço, que pode ser observado na Figura 14, é gerada justamente pela comparação dos registradores R1 e R7 (vide Anexo A.3), instrução 32), que são responsáveis pelo controle de fluxo da aplicação. Uma falha em um destes registradores pode fazer com que uma *thread* se mantenha neste laço indeterminadamente. Nesta aplicação, por exemplo, R7 é o registrador que armazena o número de vezes que o laço deve ser executado, que neste caso é igual a 8. Se uma falha inverte um *bit* n deste registrador de 0 para 1, este laço será executado 2^n vezes a mais do que o esperado, o que pode levar o programa a um estado de Hang.

Além da quantidade de Hangs, o gráfico da Figura 28 mostra a quantidade de SDCs por registrador, o qual demonstra que uma falha em qualquer registrador pode provocar

um SDC no sistema. O número destes erros variou de acordo com os tempos de vida dos registradores, sendo que a maioria dos SDCs ocorreram devido a falhas no registrador R6, que possui o maior tempo de vida entre todos os registradores. A probabilidade de ocorrência de erros em relação ao intervalo de instruções do programa está diretamente relacionado com o comportamento dinâmico da aplicação, que determina em quais os trechos do programa a GPGPU estará mais suscetível a falhas. Por esta razão, tanto os SDCs quanto Hangs tenderam a ocorrer em maior frequência no mesmo intervalo do programa.

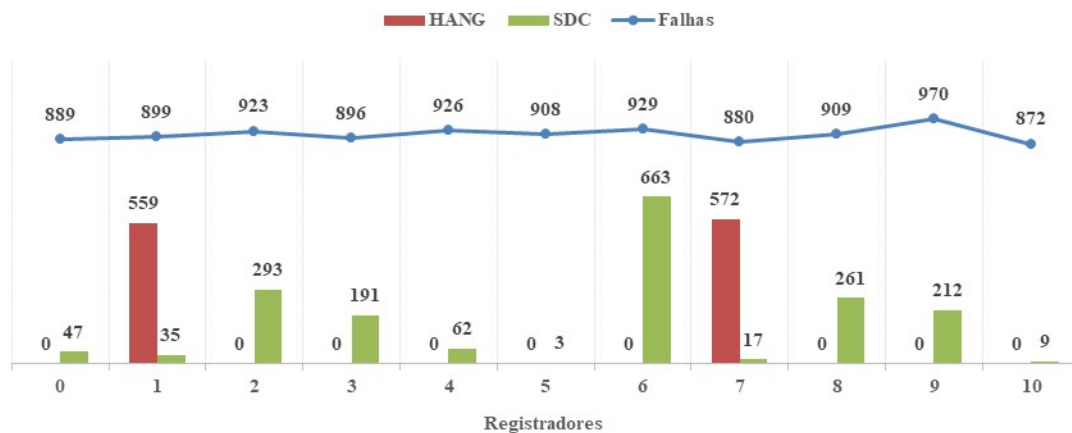

Figura 28 – Multiplicação de Matrizes: ocorrência de erros em VRF

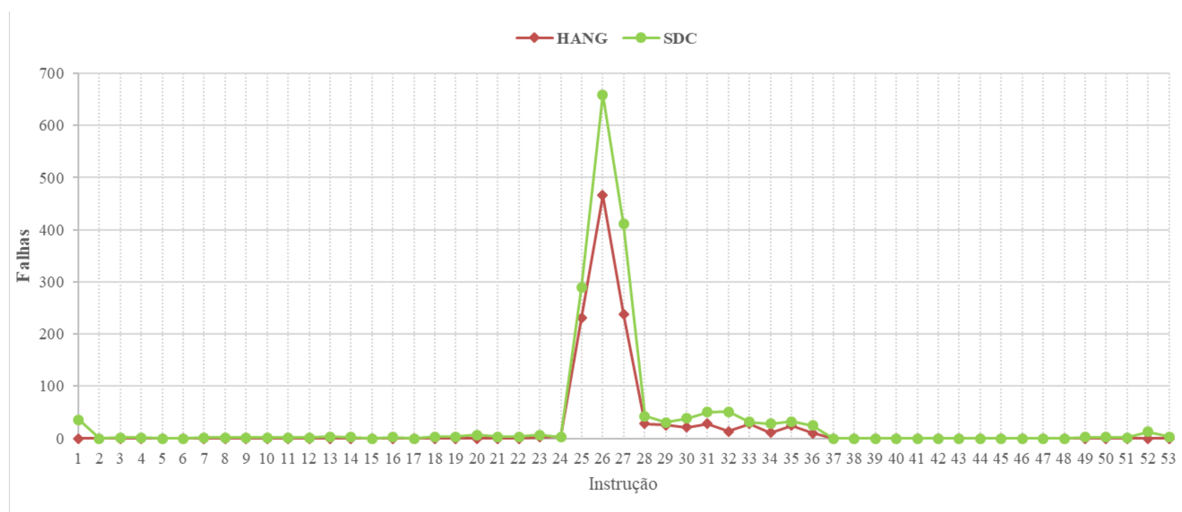

Figura 29 – Multiplicação de Matrizes: ocorrência de erros em VRF x intervalo de instruções

Um comportamento semelhante ao observado na Multiplicação de Matrizes ocorre na Autocorrelação. Os Hangs somente foram causados principalmente devido a falhas no registrador R1, que é o responsável pelo controle do laço de execução que pode ser visto na Figura 13. Os SDCs ocorreram em sua maioria devido a falhas no registrador R5, que possui o maior tempo de vida entre todos os registradores, e é utilizado para acumular o resultado final da *thread*. Este, por sua vez, é armazenado na memória na penúltima

instrução do código. É interessante observar que, para estas duas aplicações, o sistema foi mais suscetível a SDCs do que a Hangs que, nestes casos, acontece devido às instruções de sincronismo que contam com a divergência de execução entre as *threads* durante a execução dos laços mencionados, o que será discutido a seguir.

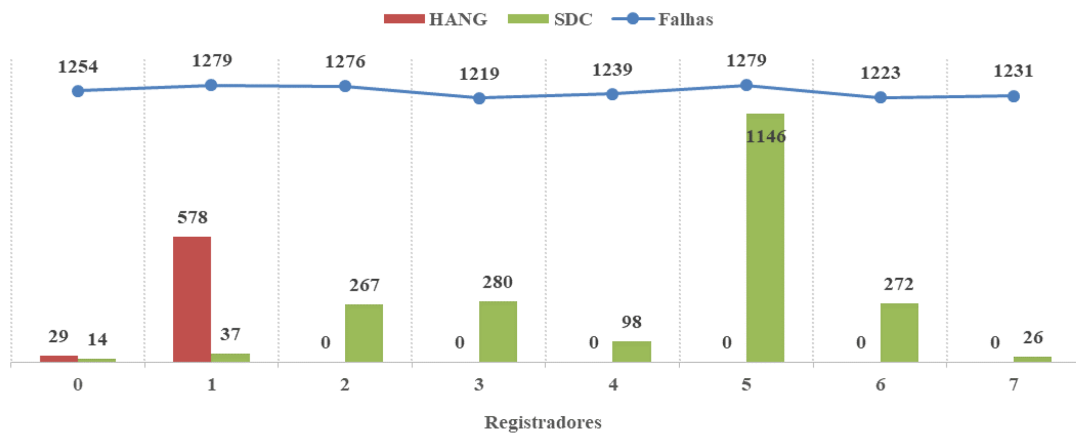

Figura 30 – Autocorrelação de Vetores: ocorrência de erros por registrador em VRF

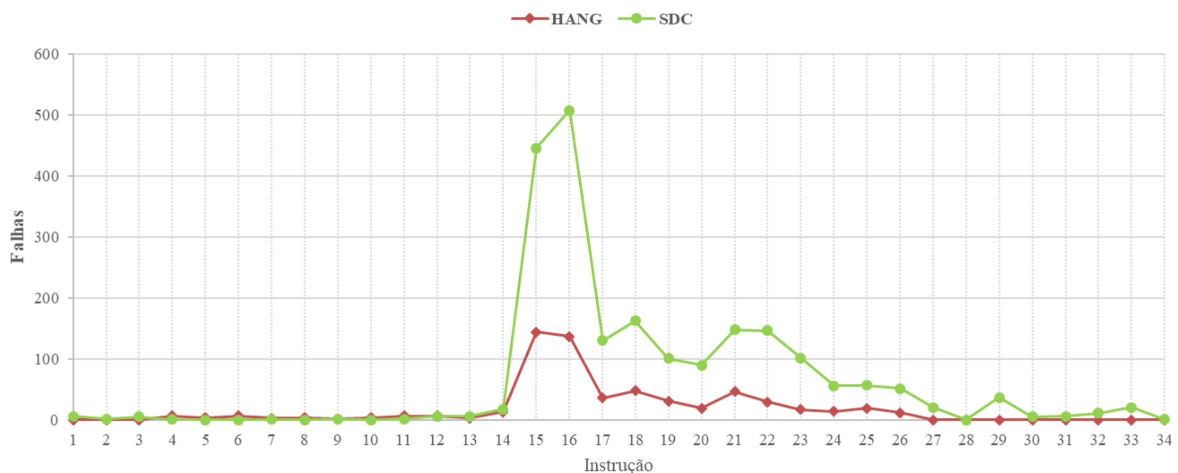

Figura 31 – Autocorrelação de Vetores: ocorrência de erros em VRF x intervalo de instruções

Diferentemente dos algoritmos anteriores, a execução do algoritmo de Ordenação de Vetores, que é algoritmo que utiliza o caminho de controle com maior intensidade, tornou a FlexGrip mais suscetível a Hangs. Estes erros foram provocados devido a falhas nos registradores R4 e R5, sendo que o tempo de vida de R5 é maior do que o tempo de R4. R5 e R4 são escritos pela primeira vez no início do programa (Anexo A.4, instruções 9 e 10), antes da instrução de sincronismo SSY, mas também são utilizados para controle de fluxo antes da instrução SSY e após instrução NOP.S. Isso significa que o algoritmo não considera a ocorrência de divergência de execução em desvios condicionais que dependem dos registradores R4 e R5 (instruções 11, 40 e 43). Portanto, quando uma falha ocorre nestes registradores durante o intervalo de execução entre as instruções 9 e

43, demonstrado na Figura 33, as *threads* podem perder o sincronismo facilmente. Assim como na Multiplicação de Matrizes, uma falha em qualquer registrador pode causar um SDC, sendo que a maior parte destes erros ocorreram devido a falhas em R3. Isso ocorre pois o valor de R3 é utilizado por várias instruções, assim a falha é facilmente propagada no sistema causando um resultado incorreto na saída.

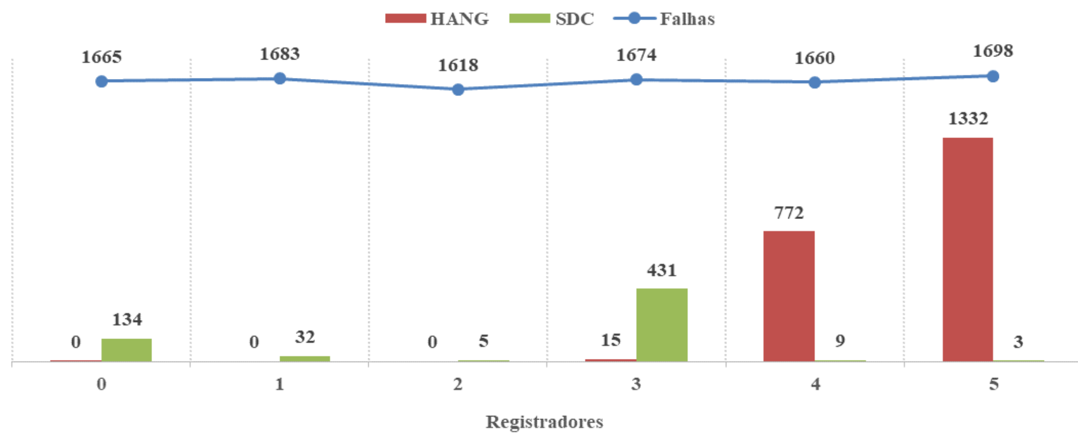

Figura 32 – Ordenação de Vetores: ocorrência de erros por registrador em VRF

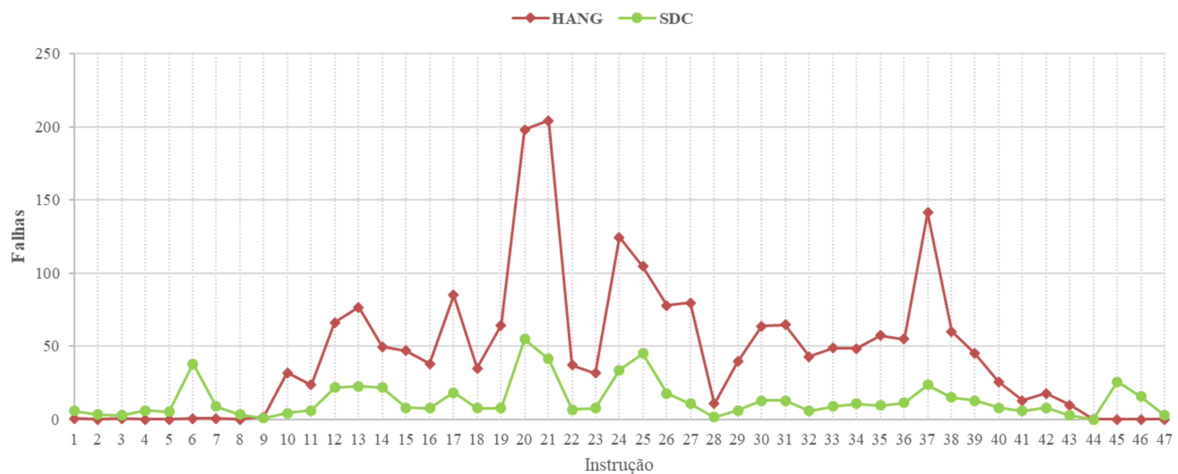

Figura 33 – Ordenação de Vetores: ocorrência de erros em VRF x intervalo de instruções

Um comportamento semelhante ao observado no Bitonic Sort também ocorre no algoritmo de Redução. Todos os Hangs ocorreram devido a falhas no registrador R2, que é o único registrador utilizado para gerar as *flags* de controle de fluxo. R2 é escrito logo na primeira instrução com um valor da memória compartilhada. Após isso, R2 é apenas atualizado com instruções de *shift* que utilizam seu próprio valor como operando (instruções 18 e 28) que geram as *flags* para o controle de fluxo de execução. Diferentemente da Multiplicação e da Autocorrelação, esta aplicação não conta com instruções de sincronismo para controle de divergência de execução, de modo que um desvio indevido de uma *thread* leva o sistema para um estado de Hang. Por estas razões, esta aplicação torna a GPGPU

muito sensível a Hangs, se mostrando ainda mais sensível que a Autocorrelação, que é uma aplicação que explora mais o caminho de controle. Tais resultados demonstram que a inserção de instruções de sincronismo podem reduzir consideravelmente a quantidade de Hangs, embora estas falhas sejam convertidas em SDCs. A suscetibilidade a SDCs para a Redução variou de acordo com o esperado, sendo maior para os registradores com maior tempo de vida.

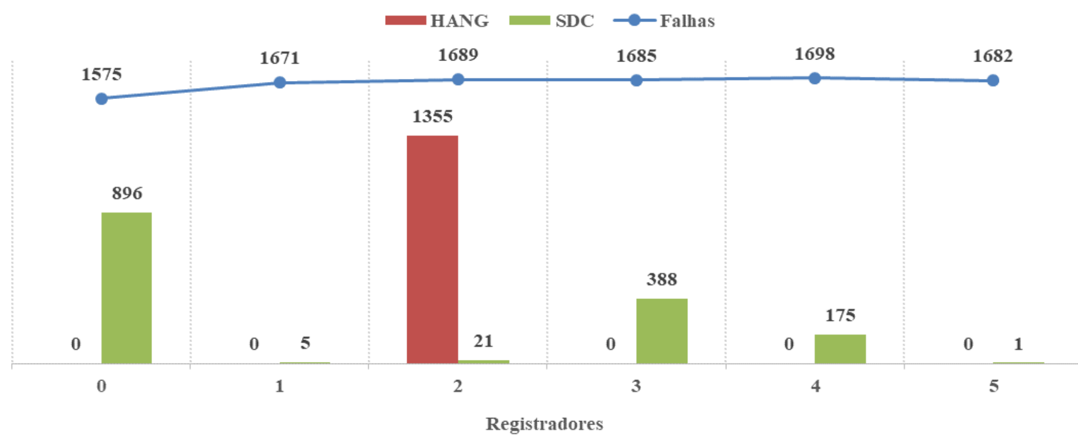

Figura 34 – Redução de Vetores: ocorrência de erros por registrador em VRF

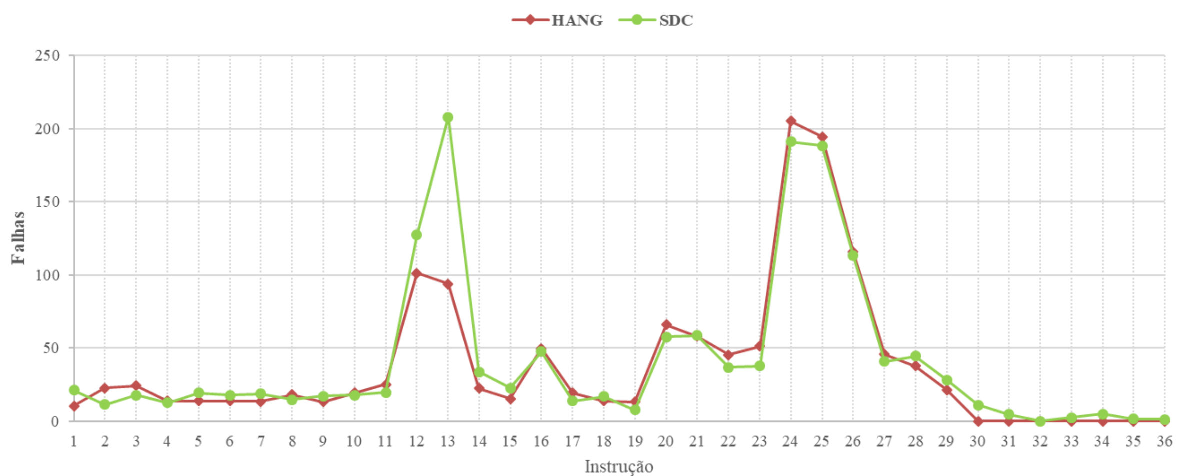

Figura 35 – Redução de Vetores: ocorrência de erros em VRF x intervalo de instruções

Nos algoritmos de Ordenação e de Redução, além dos Hangs causados devido às falhas que causam problemas de sincronismo, Hangs também puderam ser observados quando os SEUs foram simulados em um dos 16 *bits* mais significativos dos registradores R4 e R2 para a Ordenação e Redução, respectivamente, que são utilizados como operandos na instrução SHR.C0. Quando um destes 16 *bits* se encontra em nível alto durante a execução da instrução SHR.C0, é gerada uma exceção no simulador que finaliza abruptamente a simulação. O levantamento desta exceção poderia ser tratado como um mecanismo de detecção de falhas, porém, dependendo do projeto, esta interrupção abrupta na GPGPU

pode ser considerada como um comportamento indesejado no sistema. Portanto, estas exceções foram tratadas como Hangs. Se tais exceções não tivessem sido tratadas como Hangs, mas como um mecanismo de detecção de falhas, a quantidade total destes erros teria sido inferior ao demonstrado na Tabela 8.

A Tabela 9 demonstra os resultados para a técnica VRF Hard que, quando implementada sobre os algoritmos originais, reduziu o número total de erros de 28,1% para 0,7%, o que representa uma taxa de redução de 97,5%. Conforme pode ser observado, os SDCs e Hangs foram reduzidos em 95,6% e 99,3%, respectivamente. As taxas de redução de erros foram elevadas para todas as aplicações e, considerando que VRF é o maior banco de registradores e o mais frequentemente acessado pelas aplicações, tais resultados demonstram que a técnica VRF Hard possui um grande potencial para detectar falhas em GPUs. Ressalta-se que VRF Hard replica todos os registradores do algoritmo e, em média, gerou um aumento em tempo de execução de 62% com relação ao código original. Considerando isto, as elevadas taxas de redução de erros e o fato das análises realizadas sobre os algoritmos originais terem demonstrado os trechos de código e os registradores mais sensíveis a falhas, VRF Hard poderá oferecer uma excelente troca entre desempenho e capacidade de detecção de falhas em uma replicação seletiva.

Tabela 9 – Injeções em VRF - resultados para VRF Hard

			ACE		***Taxa de Redução***		
Algoritmo	**Detected**	**unACE**	**SDC**	**Hang**	***ACE***	**SDC**	**Hang**
Multiplicação	62,8%	36,6%	0,6%	0,0%	97,9%	96,6%	100%
Ordenação	64,1%	35,5%	0,3%	0,1%	98,6%	95,2%	99,5%
Autocorrelação	44,3%	55,3%	0,4%	0,0%	98,5%	98,1%	100%
Redução	47,3%	51,3%	1,1%	0,3%	95,1%	92,6%	97,8%
Total	**54,6%**	**44,7%**	**0,6%**	**0,1%**	**97,5%**	**95,6%**	**99,3%**

Os erros que não puderam ser evitados foram causados principalmente devido ao ponto de falha de VRF Hard, em que a replicação das instrução de *load* é feita utilizando uma simples instrução de *mov* ao invés de uma segunda instrução de *load*. Alguns erros também são causados por falhas no registrador R0, pois este registrador contém o *id* da *thread* em execução, o qual é utilizado como referência pelos algoritmos de estudo de caso, cujo valor é inicializado pelo *hardware* antes das instruções começarem a ser processadas. Assim, mesmo VRF Hard replicando R0 logo na primeira instrução, uma falha em R0 que ocorre antes da execução da primeira instrução também será propagada para o registrador réplica.

Os Hangs que apareceram no Bitonic Sort e na Redução ocorreram por falhas nos registradores R4 e R2, os quais são utilizados pela instrução SHR.C0 para controle de desvios condicionais. Os erros ocorreram porque os testes de consistência foram inseridos

antes das instruções SHR.C0, a fim de garantir a integridade de C0 e prevenir as exceções que podem ocorrer quando uma falha atinge um dos 16 *bits* mais significativos dos operandos da instrução SHR.C0. Entretanto, a instrução de desvio acabou inserindo um ponto de falha no código, conforme é apontado pela seta na Figura 36. Se a exceção mencionada não tivesse sido tratada como um Hang, mas como um mecanismo de detecção de falhas, os testes de consistência poderiam ser inseridos após a instrução SHR.C0 e, assim, a quantidade de Hangs seria reduzida para zero.

```
ISET.S32.C2 o [0x7f], R4, R10, NE;
BRA C2.NE, ERRO; ←
SHR.C0 R5, R4, 0x1;
SHR R11, R10, 0x1;
BRA C0.EQ, 0x130;
```

Figura 36 – Ponto de Falha de VRF Hard

Os resultados para as técnicas PRF Hard e ARF Hard, quando submetidas à campanha de injeções em VRF, são apresentados a 10 e indicam valores apenas um pouco melhores em relação aos algoritmos originais. As taxas de redução são baixas, na ordem de 2%, pois ARF Hard somente é capaz de detectar falhas em VRF que se propagam para os registradores de endereço, enquanto PRF Hard somente é capaz de detectar falhas em registradores envolvidos na geração das *flags* de predicado do programa.

Tabela 10 – Injeções em VRF - resultados para PRF Hard e ARF Hard

	PRF Hard				***ARF Hard***			
Algoritmo	**Detect.**	**unACE**	**SDC**	**Hang**	**Detect.**	**unACE**	**SDC**	**Hang**
Multiplic.	62,8%	36,6%	0,6%	0,0%	62,8%	36,6%	0,6%	0,0%
Ordenação	64,1%	35,5%	0,3%	0,1%	64,1%	35,5%	0,3%	0,1%
Autocorr.	44,3%	55,3%	0,4%	0,0%	44,3%	55,3%	0,4%	0,0%
Redução	47,3%	51,3%	1,1%	0,3%	47,3%	51,3%	1,1%	0,3%
Total	**54,6%**	**44,7%**	**0,6%**	**0,1%**	**54,6%**	**44,7%**	**0,6%**	**0,1%**

5.1.2 Banco de Registradores de Predicado

Nesta campanha de injeção de falhas, as falhas foram injetadas apenas no PRF. A Tabela 11 mostra o resultado das injeções para as aplicações de estudo de caso originais, enquanto a Tabela 12 apresenta os resultados para PRF Hard e a tabela 13 apresenta os resultados das injeções para as técnicas VRF Hard e ARF Hard. Os gráficos das Figuras37 e 38 mostram a suscetibilidade a erros de acordo com o intervalo de instruções executadas.

Conforme pode ser observado na Tabela 11, apenas 2,9% de todas as injeções resultaram em ACEs (0,7% Hangs e 2,1% SDC). Este banco de registradores possui uma

baixa suscetibilidade a erros pois, quando comparado com VRF, as instruções que utilizam as *flags* de predicado geralmente são poucas e esses registradores normalmente apresentam curtos intervalos entre a escrita e a leitura destas *flags*, visto que os testes de predicado normalmente são realizados próximos às instruções que geram as *flags*. Ainda assim, tais falhas devem ser levadas em consideração no projeto de técnicas de tolerância a falhas para a proteção de GPUs.

Tabela 11 – Injeções em PRF - resultados para os algoritmos originais

Algoritmo	**Detected**	**unACE**	**SDC**	**Hang**	***ACE***
Multiplicação	-	99,3%	0,5%	0,2%	0,7%
Ordenação	-	97,4%	1,5%	1,1%	2,6%
Autocorrelação	-	97,4%	2,2%	0,4%	2,6%
Redução	-	94,3%	4,6%	1,1%	5,7%
Total	**-**	**97,1%**	**2,2%**	**0,7%**	**2,9%**

A Tabela 11 mostra que as injeções de falhas nos registradores de predicado causaram mais SDCs do que Hangs no sistema, mesmo sendo estes os registradores, ou *flags*, que determinam o fluxo de execução dos programas. O comportamento observado para estes registradores, quando submetidos a falhas, é semelhante ao observado em VRF, pois estas *flags* são geradas a partir de instruções que realizam suas operações sobre VRF. A ocorrência de SDCs ou Hangs depende de como estas *flags* são utilizadas e onde elas se encontram no código.

SDCs tendem a ocorrer com maior facilidade em dois casos: (1) quando as *flags* são utilizadas para execução condicional de instruções, que não sejam de desvio, e (2) quando as falhas incidem sobre *flags* utilizadas para controle de fluxo que contam com instruções precedentes de sincronismo. Os dois casos podem ser observados no gráfico 37 e no Anexo A.4, sendo que o primeiro caso ocorre no intervalo do código entre as instruções 30-36, onde instruções condicionais são executadas, e o segundo caso ocorre entre os intervalos 12-19 e 20-23, onde ocorrem desvios do fluxo de execução esperado por causa de falhas em C0, que causam SDCs no saída.

Os Hangs, por sua vez, tendem a ocorrer em dois casos: (1) quando a falha afeta uma instrução de desvio condicional que não conta com instruções precedentes de sincronismo e (2) quando a falha interfere em um laço de repetição, de modo que a execução indevida do laço altera uma variável de controle, o que pode fazer com que este seja executado por um tempo indeterminado. O primeiro caso pode ser facilmente constatado na Multiplicação de Matrizes, no primeiro intervalo mostrado no gráfico da Figura 38, em que apenas Hangs ocorrem devido à instrução de desvio condicional 6 (Anexo A.3) não ser precedida por instruções de sincronismo. Este caso também pode ser observado no primeiro e nos dois últimos intervalos que geraram Hangs no Bitonic Sort. O segundo caso,

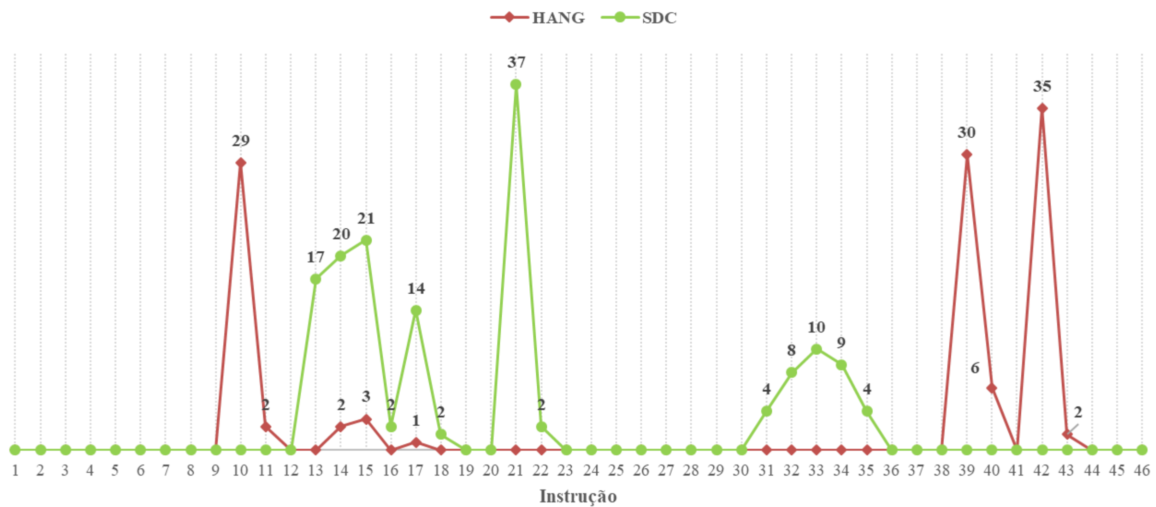

Figura 37 – Ordenação de Vetores: ocorrência de erros em PRF x intervalo de instruções

por sua vez, pode ser observado no laço da Multiplicação, no intervalo 31-36, bem como no Bitonic Sort, no intervalo 13-18, que, apesar de possuírem instruções precedentes de sincronismo, apresentam a ocorrência de Hangs, porém em menor percentual que SDCs.

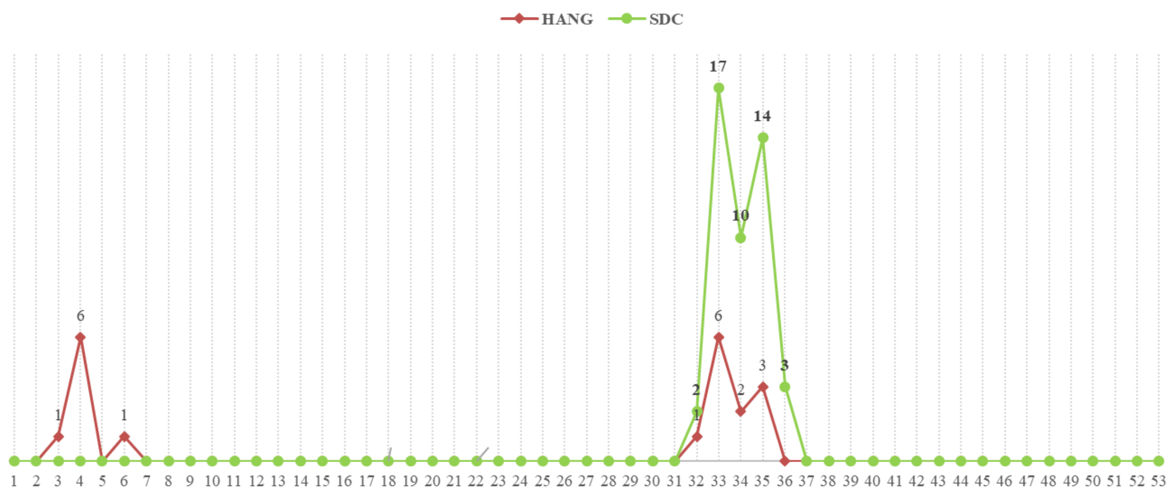

Figura 38 – Multiplicação de Matrizes: ocorrência de erros em PRF x intervalo de instruções

A Tabela 12 demonstra os resultados para PRF Hard que, quando aplicado sobre os algoritmos originais, mostra uma redução de SDCs para 0, exceto no algoritmo de Redução que mostrou 0,6% de erros (87,7% de redução). A redução média em SDCs e Hangs foram de 96,9% e 58,4%, respectivamente. Embora a proteção tenha sido capaz de eliminar a maior parte dos erros do tipo SDC, a técnica não foi capaz de alcançar as mesmas taxas de detecção para os erros do tipo Hang. Isso ocorre, principalmente, devido a problemas de sincronização entre as *threads*, que não são tratados pela técnica de tolerância a falhas PRF Hard, conforme descrito na Seção 4.1.2. Ainda assim, considerando o custo de aumento no tempo de execução médio de 19% do PRF Hard e as elevadas taxas de redução de SDCs

obtidas, esta técnica poderia ser utilizada em conjunto com outras técnicas de tolerância a falhas para proteger trechos específicos do código de execução condicional.

Tabela 12 – Injeções em PRF - resultados para PRF Hard

			ACE		***Taxa de Redução***		
Algoritmo	**Detected**	**unACE**	**SDC**	**Hang**	***ACE***	**SDC**	**Hang**
Multiplicação	1,0%	98,8%	0,0%	0,2%	73,9%	100%	12,9%
Ordenação	4,6%	94,8%	0,0%	0,6%	77,8%	100%	47,5%
Autocorrelação	3,1%	96,9%	0,0%	0,0%	100%	100%	100%
Redução	21,3%	77,9%	0,6%	0,3%	84,9%	87,7%	73,2%
Total	**7,5%**	**92,1%**	**0,1%**	**0,3%**	**84,2%**	**96,9%**	**58,4%**

A Tabela 13 mostra que as técnicas VRF Hard e ARF Hard causaram, respectivamente, um pequeno aumento e uma pequena redução na quantidade total de erros, de 2,9% para 3,8%, no caso da VRF Hard e de 4,15% para 3,3% em ARF Hard. O aumento provocado pela VRF Hard ocorreu pelo fato das instruções inseridas por esta técnica distanciar as instruções originais que escrevem nas *flags* de predicado das instruções que efetivamente utilizam estas *flags*, o que aumenta o tempo de vida destes registradores e, consequentemente, a probabilidade de propagação de uma falha. Além disso, nas campanhas de injeção de falhas sobre PRF, as falhas foram injetadas apenas sobre os registradores originais, se também tivessem sido injetadas nas variáveis réplica, o número de detecções teria aumentado, e a probabilidade de ocorrência de erros seria reduzida, pois falhas nas variáveis réplicas não devem causar erros no sistema. ARF Hard, por outro lado, foi capaz de reduzir levemente o número de erros devido à instruções condicionais que fazem com que algumas falhas se propaguem para os registradores de VRF, utilizados para armazenar as réplicas do ARF.

Tabela 13 – Injeções em PRF - resultados para VRF Hard e ARF Hard

	VRF Hard				***ARF Hard***			
Algoritmo	**Detect.**	**unACE**	**SDC**	**Hang**	**Detect.**	**unACE**	**SDC**	**Hang**
Multiplic.	0,0%	98,5%	1,3%	0,3%	-	-	-	-
Ordenação	0,4%	95,7%	1,5%	2,4%	0,0%	98,0%	1,0%	0,9%
Autocorr.	0,0%	97,3%	2,1%	0,6%	-	-	-	-
Redução	2,8%	90,2%	5,5%	1,5%	1,7%	93,8%	3,9%	0,7%
Total	**0,8%**	**95,4%**	**2,6%**	**1,2%**	**0,9%**	**95,9%**	**2,5%**	**0,8%**

5.1.3 Banco de Registradores de Endereço

Nesta campanha de injeção de falhas, as falhas foram injetadas apenas em ARF. A Tabela 14 mostra o resultado das injeções para as aplicações de estudo de caso originais,

enquanto a Tabela 16 apresenta os resultados para ARF Hard e a tabela 13 apresenta os resultados das injeções para as técnicas VRF Hard e PRF Hard. Os gráficos das Figuras39 e 40 mostram a suscetibilidade a erros de acordo com intervalo de instruções executadas.

Como pode ser observado na Tabela 14, o ARF não é tão sensível a falhas como VRF. De todas as falhas injetadas, apenas 2,8% causaram SDCs e apenas 1 falha, ou 0,00005% das falhas (não mostrado na Tabela 14), causou um Hang no sistema. Esta diferença entre SDCs e Hangs acontece, principalmente, por causa da leitura de dados incorretos, sendo que, para estas aplicações, SDCs são mais propensos a ocorrer por utilizarem estes dados, principalmente para computar os valores de saída do sistema. O armazenamento de dados incorretos poderia causar mais Hangs se o código de programa pudesse ser sobrescrito, o que não é comum em arquiteturas de memória *Harvard*.

Tabela 14 – Injeções em ARF - resultados para os algoritmos originais

Algoritmo	**Detected**	**unACE**	**SDC**	**Hang**	***ACE***
Ordenação	-	96,1%	3,9%	0,0%	3,9%
Redução	-	98,2%	1,8%	0,0%	1,8%
Total	**-**	**97,1%**	**2,9%**	**0,0%**	**2,9%**

Conforme pode ser observado no gráfico da Figura 39 e no código fonte do Bitonic Sort (Anexo A.4), os erros causados por falhas no ARF coincidem com os intervalos entre a escrita e a leitura dos registradores de ARF e da probabilidade de ocorrência de falhas nestes intervalos, que depende do tempo do comportamento dinâmico do programa. Além disso, os resultados demonstraram que 83,0% dos erros ocorreram devido a falhas no registrador A1, que é o registrador mais utilizado. De todas as falhas injetadas, 393 falhas causaram SDCs e apenas uma falha em A2, que incidiu durante a execução da instrução de controle 21, provocou um Hang no sistema. Nesta instrução, A2 é utilizado como operando em uma comparação para controle de fluxo.

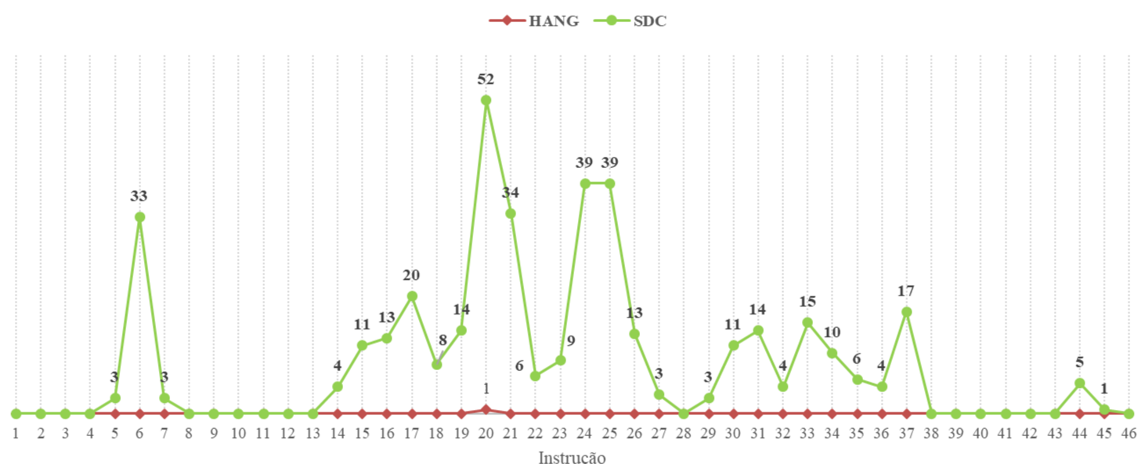

Figura 39 – Ordenação de Vetores: ocorrência de erros em ARF x intervalo de instruções

O mesmo comportamento pode ser observado no algoritmo de Redução, sendo que os erros coincidem com o tempo entre a escrita e a leitura de ARF, que estão entre o intervalo de instruções 14-17 e no intervalo 21-27, conforme pode ser observado no código do anexo A.1 e no gráfico da Figura 40. No intervalo 21-27, o número de erros é maior devido à maior probabilidade de ocorrência de falhas neste intervalo. Esta aplicação é menos suscetível a erros do que a Ordenação, pois seus registradores apresentam, proporcionalmente, menor tempo médio de vida.

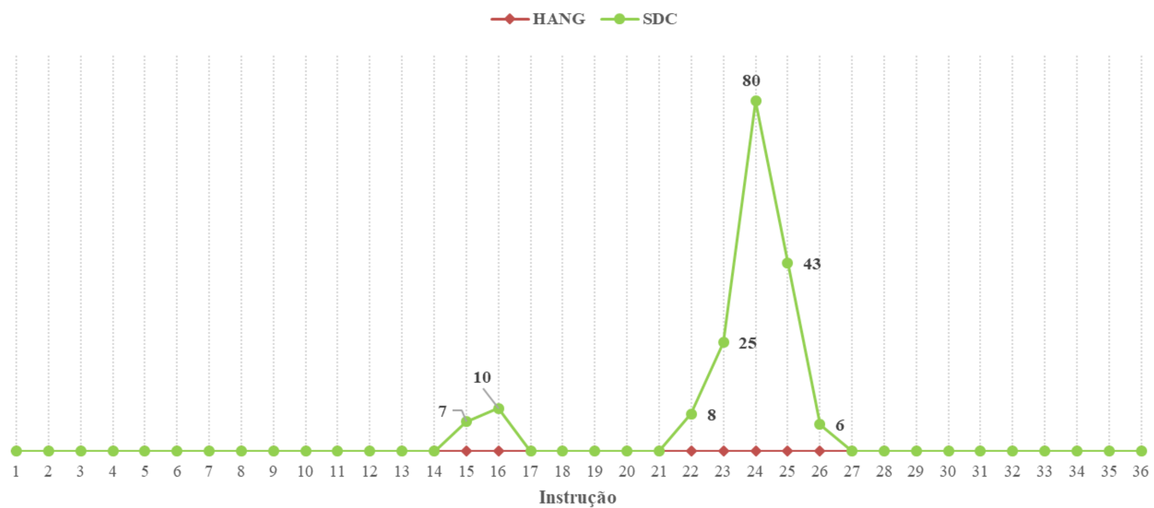

Figura 40 – Redução de Vetores: ocorrência de erros em ARF x intervalo de instruções

A Tabela 15 apresenta os resultados para ARF Hard que, quando aplicado sobre os algoritmos originais, foi capaz de detectar 100% das falhas injetadas que causaram erros no sistema, reduzindo o número de erros para zero. Em termos de detecção de falhas, estes resultados são excelentes, mesmo considerando o pequeno percentual de erros observados.

Tabela 15 – Injeções em ARF - resultados para ARF Hard

			ACE		***Taxa de Redução***		
Algoritmo	**Detected**	**unACE**	**SDC**	**Hang**	***ACE***	**SDC**	**Hang**
Ordenação	41,6%	58,4%	0,0%	0,0%	100%	100%	100%
Redução	73,5%	26,5%	0,0%	0,0%	100%	100%	-
Total	**57,6%**	**42,4%**	**0,0%**	**0,0%**	**100%**	**100%**	**100%**

VRF Hard e PRF Hard, por outro lado, não foram capazes de reduzir os erros causados por falhas no ARF. Conforme mostrado na Tabela 16, embora as técnicas tenham sido capazes de detectar uma pequena quantidade falhas, a diferença total entre os erros das aplicações originais e protegidas por estas técnicas é inferior a 0,5%, o que pode ser considerado o mesmo resultado obtido para as aplicações originais. Estes resultados mostram que, para proteger ARF, a técnica ARF Hard deve ser utilizada.

Tabela 16 – Injeções em ARF - resultados para VRF Hard e PRF Hard

	VRF Hard				***PRF Hard***			
Algoritmo	**Detect.**	**unACE**	**SDC**	**Hang**	**Detect.**	**unACE**	**SDC**	**Hang**
Ordenação	0,0%	95,8%	4,2%	0,0%	0,3%	95,8%	3,9%	0,0%
Redução	0,5%	97,9%	1,6%	0,0%	0,0%	98,2%	1,8%	0,0%
Total	**0,3%**	**96,9%**	**2,9%**	**0,0%**	**0,1%**	**97,0%**	**2,9%**	**0,0%**

5.1.4 Registradores de Pipeline

Nesta campanha de injeção de falhas, as falhas foram injetadas somente nos registradores de *pipeline* da GPGPU. A Tabela 17 mostra os resultados de injeção de falhas para as aplicações de estudo de caso originais enquanto que a Tabela 18 apresenta os resultados para as técnicas de tolerância a falhas VRF Hard, PRF Hard e ARF Hard, respectivamente.

Tabela 17 – Injeções no *pipeline* - resultados para os algoritmos originais

Algoritmo	**Detected**	**unACE**	**SDC**	**Hang**	***ACE***
Multiplicação	-	75,8%	14,2%	10,0%	24,2%
Ordenação	-	91,5%	4,1%	4,4%	8,5%
Autocorrelação	-	92,7%	2,8%	4,5%	7,3%
Redução	-	88,0%	5,3%	6,8%	12,0%
Total	**-**	**87,0%**	**6,6%**	**6,4%**	**13,0%**

A Tabela 17 mostra que, de todas as falhas injetadas, 13% resultaram em ACEs no sistema, sendo que 6,6% das falhas causaram SDCs e 6,4% causaram Hangs. Estes resultados demonstram que o *pipeline* é menos sensível a falhas que o banco de registradores VRF, porém é mais sensível que os bancos PRF e ARF.

O algoritmo de estudo de caso mais afetado pelas falhas foi a Multiplicação de Matrizes. Isto acontece pelo fato das *threads* serem executadas em *warps* que comportam 32 *threads*, e a Multiplicação de Matrizes utiliza 64 *threads*, as quais são escalonadas em 2 *warps* que, por sua vez, utilizam todo o *hardware* da GPU. As outras aplicações, por outro lado, foram configuradas para utilizar somente 8 *threads*, o que resulta em um *warp* em que apenas 25% de sua capacidade é utilizada pelas aplicações. Assim, quando uma falha afeta uma *thread* que não está sendo utilizada pela aplicação, esta falha tende a não se propagar no sistema com a mesma facilidade das falhas que afetam as *threads* em uso. Outra característica que pode ser observada na Tabela 17 é que as falhas tenderam a provocar mais Hangs do que SDCs, exceto na Multiplicação de Matrizes, que apresentou mais SDCs. Isto ocorre porque, para as três aplicações que utilizam apenas 8 *threads*, falhas no *pipeline* que afetam as *threads* não utilizadas pelas aplicações também podem

provocar Hangs no sistema. Isto se dá pelo fato de que uma *thread* não utilizada pode ser indevidamente habilitada e, assim, entrar em dessincronismo com as *threads* utilizadas.

Além dos Hangs que ocorrem por falhas que afetam mecanismos de controle do *pipeline*, como os sinais responsáveis pelo mascaramento de execução de *threads*, bem como os sinais de controle de escalonamento de *warp* e de bloco, Hangs também são provocados por falhas que se propagam para instruções ou dados utilizados no controle da de fluxo da aplicação. SDCs, por sua vez, decorrem, principalmente, de falhas que se propagam no sistema alterando valores de registradores ou causando o mau funcionamento de instruções que não interferem no fluxo de controle da aplicação. Sendo assim, a suscetibilidade a Hangs ou SDCs devido à falhas no *pipeline* depende das instruções que compõem o algoritmo em execução e da configuração da GPU. Por estas razões, os algoritmos não demonstraram a mesma tendência a SDCs e Hangs observada quando as falhas foram injetadas sobre os bancos de registradores.

Tabela 18 – Injeções nos registradores do *pipeline* - resultados para as técnicas de detecção de falhas

Técnica	**Algoritmo**	**Detected**	**unACE**	**SDC**	**Hang**	***ACE***
VRF Hard	Multiplicação	5,3%	74,4%	10,9%	9,4%	20,3%
	Ordenação	2,6%	90,8%	2,4%	4,2%	6,6%
	Autocorrelação	4,4%	90,4%	1,2%	4,0%	5,2%
	Redução	5,4%	86,9%	2,1%	5,6%	7,7%
	Total	**4,4%**	**85,6%**	**4,1%**	**5,8%**	**9,9%**
PRF Hard	Multiplicação	0,3%	75,8%	13,7%	10,2%	23,9%
	Ordenação	2,0%	90,7%	3,0%	4,2%	7,2%
	Autocorrelação	1,1%	92,7%	2,3%	4,0%	6,2%
	Redução	1,1%	88,8%	4,0%	6,1%	10,1%
	Total	**1,1%**	**87,0%**	**5,7%**	**6,1%**	**11,9%**
ARF Hard	Ordenação	1,9%	91,0%	3,1%	4,1%	7,1%
	Redução	1,5%	88,0%	3,7%	6,8%	10,5%
	Total	**1,7%**	**89,5%**	**3,4%**	**5,4%**	**8,8%**

Quando as técnicas VRF Hard e PRF Hard foram aplicadas aos algoritmos originais, o total de erros foi reduzido de 13% para 9,9% e 11,2%, respectivamente. Ao considerar apenas os algoritmos em que ARF Hard se aplica, o total de erros foi reduzido de 10,5% para 8,8% quando ARF Hard foi aplicado sobre os algoritmos originais. Embora os resultados médios não representem valores tão expressivos, ao considerar a taxa de redução de erros, principalmente de SDCs, para cada aplicação, alguns resultados significativos podem ser observados. A Figura 41 demonstra as taxas de redução de *ACE*, Hang e SDC para cada aplicação.

É importante salientar que as técnicas propostas não foram projetadas para pro-

teger o *pipeline*. Este componente foi submetido à campanha de injeção de falhas com o intuito de analisar o seu comportamento na presença de falhas e verificar se as técnicas propostas apresentam algum impacto, positivo ou negativo, na redução de erros causados por SEUs.

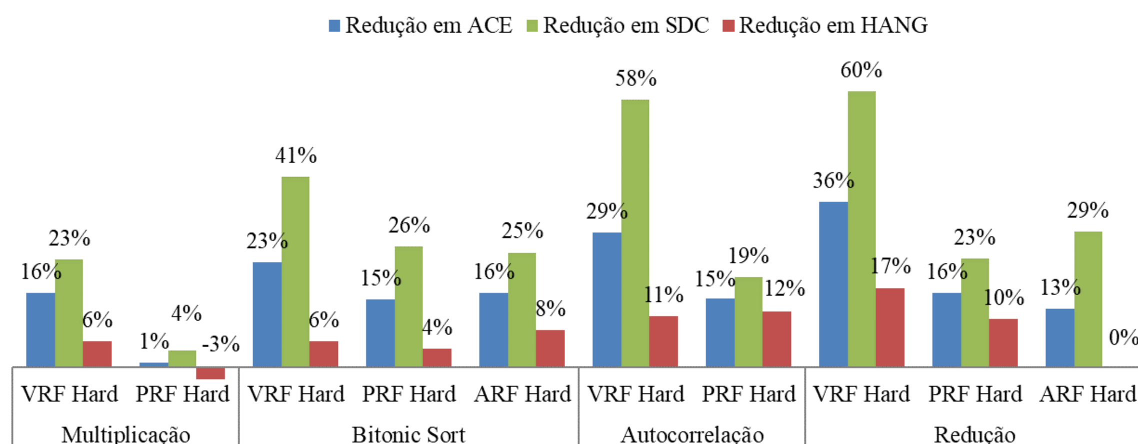

Figura 41 – Taxas de redução de erros no *pipeline* obtidas a partir das técnicas de detecção de falhas

Conforme pode ser observado na Figura 41, as taxas de redução de falhas são maiores para SDCs, sendo que VRF Hard é a técnica com maior potencial de detecção de falhas do *pipeline*. As técnicas de *software* implementadas não alcançam elevadas taxas de redução de Hangs porque falhas em registradores de controle podem causar erros que fogem do alcance das técnicas utilizadas. Como exemplo, pode-se citar as falhas que corrompem o PC, o que pode levar o programa para um endereço de execução além do último endereço do programa, e não é considerado pelas técnicas implementadas. Outro exemplo, que ocorre em maior frequência na GPGPU, são as falhas em *current_mask*, sinal que transcorre todo *pipeline* e que é utilizado para mascarar a execução das *threads*. Uma *thread* indevidamente desabilitada, além de não executar as instruções do código (protegido ou não), não encontra instrução de finalização e leva o sistema para um estado de Hang. No entanto, Hangs causados por falhas que se propagam até os registradores da aplicação responsáveis pelo fluxo de controle podem ser detectados pelas técnicas propostas.

SDCs, normalmente, correspondem aos resultados obtidos a partir das falhas que se propagam para o código do programa e, portanto, o esperado é que sejam reduzidos em maior quantidade, conforme pode ser constatado na Figura 41. O potencial de redução deste tipo de erro depende dos pontos de falhas de cada algoritmo e é proporcional ao percentual de variáveis replicadas e das verificações de consistências realizadas pelas técnicas. Por estas razões, as taxas de redução são diferentes para cada técnica. VRF Hard, técnica que utiliza mais réplicas e realiza o maior número de verificações de consistência, foi capaz de reduzir SDCs de 23%, para a Multiplicação, até 60%, para a Redução.

Ao analisar o porquê desta diferença na redução de erros pôde-se observar que ela decorre das instruções utilizadas por cada algoritmo, o que pode adicionar pontos de falhas no sistema, bem como do seu comportamento dinâmico. As Figuras 42, 43, 44 e 44 mostram apenas os sinais da GPGPU que, quando submetidos à campanha de injeção, provocaram SDCs durante a execução de VRF Hard aplicado aos algoritmos de Multiplicação, Ordenação, Autocorrelação e Redução, respectivamente. Os sinais estão organizados, conforme percentual de SDCs obtido, em ordem decrescente e numerados de acordo com Anexo C.

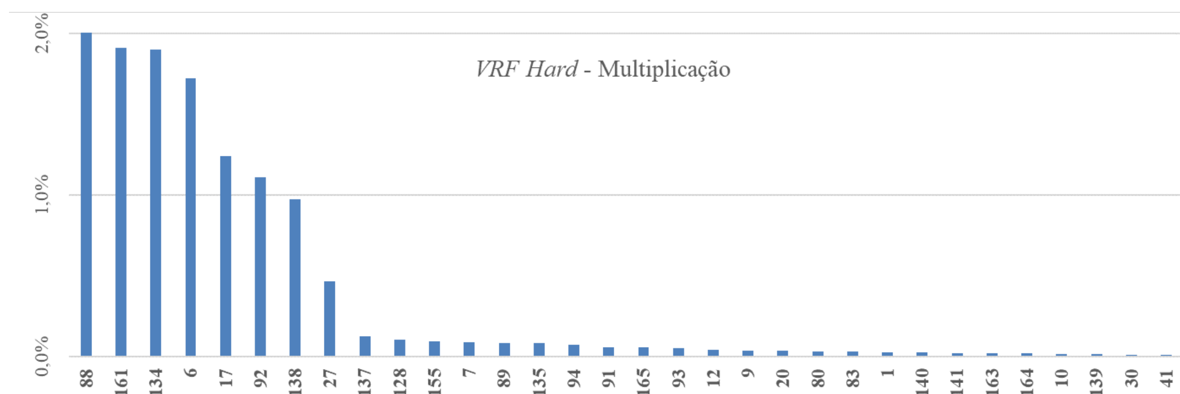

Figura 42 – Multiplicação de Matrizes: SDCs x registradores do *pipeline*

Para a Multiplicação, o gráfico demonstra que, diferentemente das outras aplicações, falhas em *current_mask* foram as principais causas de SDCs que não puderam ser detectados por VRF Hard. Isso ocorre porque na Multiplicação de Matrizes todas as *threads* devem estar habilitadas durante toda a execução do laço, que pode ser identificado na Figura 14. Este laço é precedido por uma instrução de sincronismo SSY, que armazena o valor da máscara corrente. A instrução de reconvergência NOP.S é executada após este laço. Sendo assim, quando uma falha corrompe o mascaramento de execução das *threads*, uma *thread* é desabilitada durante a execução do laço de repetição. Porém, após o laço de repetição, a máscara de *threads* armazenada antes do laço é retomada e todas as *threads* seguem a execução. Como nenhuma instrução é executada pela *thread* retomada que havia sido desabilitada, não ocorre divergência entre os registradores originais e réplica e, assim, a técnica VRF Hard não detecta o problema. Se não houvessem as instruções de sincronismo, a *thread* afetada seguiria desabilitada e ocorreria um Hang no programa. Da mesma forma que ocorreram SDCs na Multiplicação, também ocorrem nos algoritmos de Ordenação e Autocorrelação, que possuem trechos de código que contam com instruções de sincronismo. No entanto, estas duas aplicações são menos afetadas, pois apenas falhas nos 8 *bits* menos significativos de current_mask, que correspondem ao mascaramento das 8 *threads* em uso, podem causar SDCs nestas aplicações. Falhas nos outros *bits*, que causariam Hangs nos algoritmos originais, são detectadas pela VRF Hard.

No algoritmo de Ordenação, a maioria dos SDCs foram causados por falhas no sinal

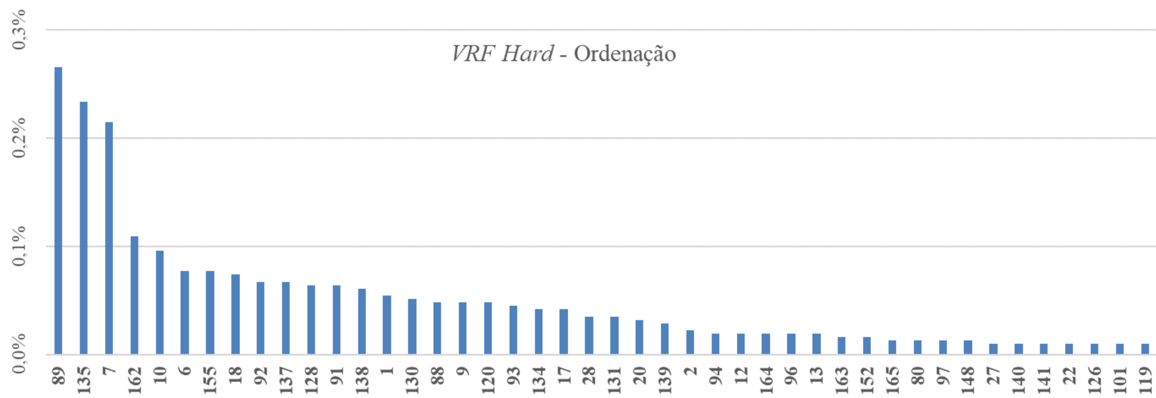

Figura 43 – Ordenação de Vetores - SDCs x registradores do *pipeline*

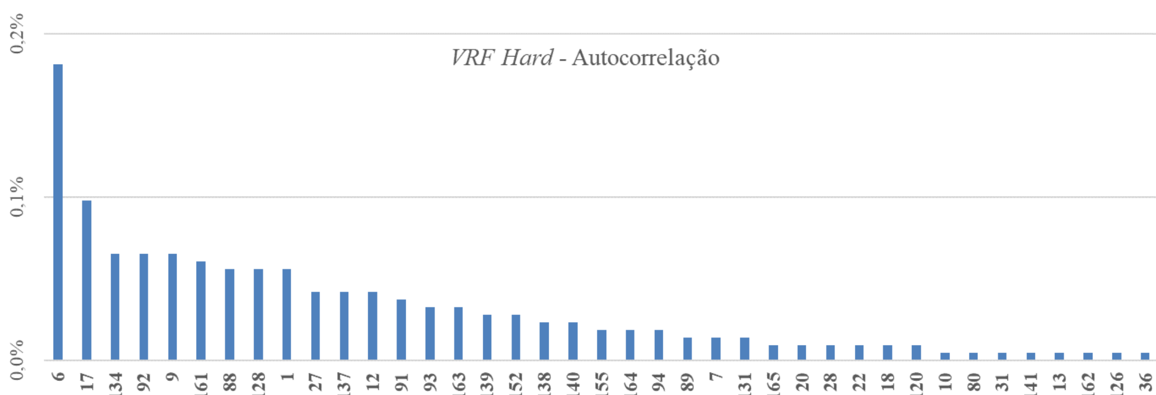

Figura 44 – Autocorrelação de Vetores - SDCs x registradores do *pipeline*

shmem_base_addr, que mantém o endereço base para acesso à memória compartilhada e, de fato, esta é a aplicação que mais acessa esta memória. Uma vez que VRF Hard substitui as instruções de leitura desta memória por um simples *mov*, estas instruções se tornam pontos de falha no programa, pois não podem ser detectadas. Uma forma de detectar estes erros seria replicando estas instruções a um custo de perda desempenho.

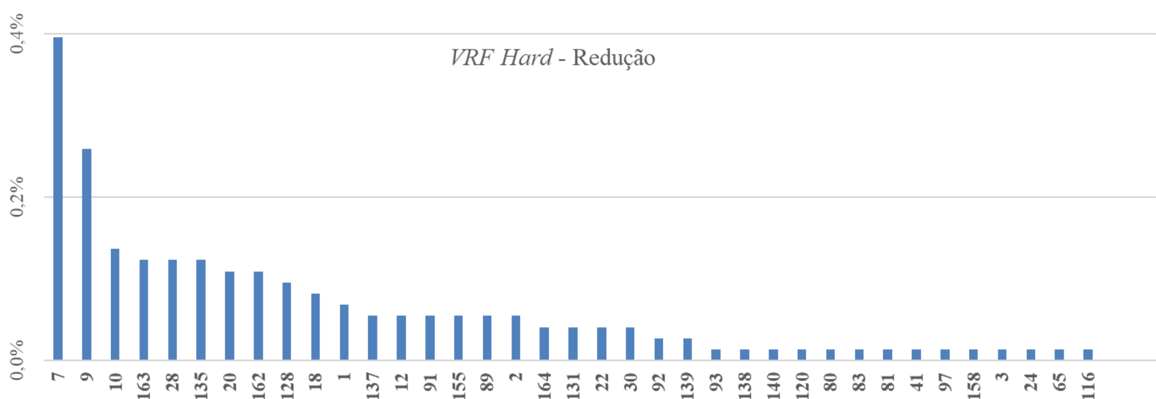

Figura 45 – Redução de Vetores - SDCs x registradores do *pipeline*

Por fim, na Redução os sinais mais sensíveis também são os sinais para acesso à

memória compartilhada, assim como no Bitonic Sort, pois esta aplicação possui algumas instruções de acessos a esta memória, que são pontos de falhas para VRF Hard. Estes resultados demonstram que para aumentar as taxas de redução de SDCs causados por falhas no *pipeline*, deve-se eliminar os pontos de falha replicando as instruções de acesso à memória. No entanto, a probabilidade de ocorrência de falhas no *pipeline*, bem como a suscetibilidade a erros, é muito baixa quando comparado com o banco VRF e, portanto, tal abordagem não é relevante para uma técnica cujo objetivo é proteger os bancos de registradores de dados.

Em síntese, os resultados demonstram que as técnicas desenvolvidas para detectar falhas nos bancos de registradores de GPUs oferecem alguma capacidade de detecção de falhas no *pipeline*. Os melhores resultados foram obtidos através da técnica VRF Hard que foi capaz de reduzir até 60% SDCs que ocorreram nas aplicações originais. Portanto, VRF Hard além de apresentar elevada taxa de redução de erros que ocorrem devido às falhas em VRF, também apresenta benefícios para redução de erros causados por falhas no *pipeline*.

6 Conclusão e Trabalhos Futuros

Esse trabalho apresentou um conjunto de técnicas de tolerância a falhas baseadas em software em baixo nível de abstração desenvolvidas para proteger os bancos de registradores de uma *soft-core* GPGPU contra SEUs. Três técnicas de detecção de falhas, denominadas VRF Hard, PRF Hard e ARF Hard, foram utilizadas para proteger, respectivamente, os bancos de registradores de dados, de endereço e de predicados da GPGPU. As técnicas propostas foram implementadas sobre quatro aplicações de estudo de caso que exploram o uso intensivo do caminho de dados e do caminho de controle da GPGPU, e os resultados em termos de desempenho, ocupação de memória e o potencial para detecção de falhas foram avaliados.

Os resultados experimentais demonstraram que os custos, em termos de tempo de execução com relação às aplicações originais desprotegidas, variaram de 30% e 77% para VRF Hard, de 3% a 36% para PRF Hard e de 23% a 57% para ARF Hard. Ao considerar a utilização de memória de programa, os resultados mostraram-se até 115% superior aos valores originais para a VRF Hard, até 48% superior para PRF Hard e até 54% superior para a técnica de proteção ARF Hard. Tais resultados de implementação demostram que técnicas de tolerância a falhas baseadas em *software* de baixo nível podem ser utilizadas em GPUs a custos de tempo de execução e consumo de memória de programa aceitáveis.

Para verificar a eficácia das técnicas implementadas, foi desenvolvido um injetor de falhas para simular SEUs em nível RTL utilizando o simulador ModelSim. Campanhas de injeção de falhas foram realizadas nos bancos de registradores e nos registradores do *pipeline* da GPGPU executando os algoritmos originais e protegidos. As campanhas de injeção foram concluídas com um total de 360.000 falhas injetadas nos bancos de registradores e 140.000 falhas injetadas nos registradores do *pipeline*. Quando as falhas foram injetadas nos bancos de registradores, as técnicas VRF Hard e PRF Hard demonstraram uma redução média de erros de 97,5% e 84,2%, respectivamente, enquanto ARF Hard foi capaz de reduzir os erros em 100%, o que significa que nenhuma falha foi capaz de causar erro no sistema. Por fim, quando as falhas foram injetadas sobre os registradores de *pipeline*, os resultados para as aplicações protegidas demonstraram alguns benefícios, embora as técnicas não tenham sido desenvolvidas com a finalidade de proteger o *pipeline*. Os resultados para VRF Hard apontaram uma redução média de erros de 26%, enquanto PRF Hard e ARF Hard apresentaram uma redução média de erros de 11,7% e 14,4%, respectivamente.

Este é primeiro trabalho da literatura a aplicar técnicas de tolerância a falhas de *software* implementadas em *assembly* (SASS) para proteger GPUs, bem como proteger

todos os bancos de registradores e realizar campanhas de injeção de falhas em nível RTL em uma GPGPU. Além disso, os resultados são promissores para auxiliar os desenvolvedores a implementar projetos de tolerância a falhas para arquiteturas de alto desempenho de processamento paralelo.

Em continuidade a este trabalho, existem diversas possibilidades a serem seguidas, tais como: (1) a realização de campanhas de injeções de falhas sobre outros componentes da GPGPU, como a memória constante e outros componentes do *pipeline*, (2) a exploração da proteção seletiva de registradores a fim de investigar a troca entre desempenho e capacidade de detecção de falhas, com o objetivo principal de reduzir a degradação de desempenho sem prejudicar a capacidade de detecção de falhas, (3) o desenvolvimento da ferramenta para automatizar a implementação das técnicas na FlexGrip e em GPUs comerciais, e (4) a proteção de uma GPU comercial com as técnicas propostas e a exposição tanto da GPU como do FPGA a um feixe de nêutrons para verificar o potencial de detecção de falhas das técnicas estudadas.

Referências

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, IEEE, v. 1, n. 1, p. 11–33, 2004. Citado na página 22.

AZAMBUJA, J. R. et al. A fault tolerant approach to detect transient faults in microprocessors based on a non-intrusive reconfigurable hardware. *IEEE Transactions on Nuclear Science*, IEEE, v. 59, n. 4, p. 1117–1124, 2012. Citado na página 42.

AZAMBUJA, J. R. F. d. Análise de técnicas de tolerância a falhas baseadas em software para a proteção de microprocessadores. 2010. Citado na página 25.

AZAMBUJA, J. R. F. D. *Designing and Evaluating Hybrid Techniques to Detect Transient Faults in Processors Embedded in FPGAs*. Tese (Doutorado) — UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL, 2013. Citado 3 vezes nas páginas 15, 21 e 24.

BARTH, J. L.; DYER, C.; STASSINOPOULOS, E. Space, atmospheric, and terrestrial radiation environments. *IEEE Transactions on Nuclear Science*, IEEE, v. 50, n. 3, p. 466–482, 2003. Citado na página 21.

BAUMANN, R. C. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *IEEE Transactions on device and materials reliability*, v. 1, n. 1, p. 17–22, 2001. Citado 2 vezes nas páginas 14 e 21.

BINDER, D.; SMITH, E.; HOLMAN, A. Satellite anomalies from galactic cosmic rays. *IEEE Transactions on Nuclear Science*, IEEE, v. 22, n. 6, p. 2675–2680, 1975. Citado na página 21.

BRAUN, C.; WUNDERLICH, H.-J. Algorithm-based fault tolerance for many-core architectures. In: IEEE. *2010 15th IEEE European Test Symposium*. [S.l.], 2010. p. 253–253. Citado na página 28.

CHANG, D. W. et al. Ercbench: An open-source benchmark suite for embedded and reconfigurable computing. In: IEEE. *2010 International Conference on Field Programmable Logic and Applications*. [S.l.], 2010. p. 408–413. Citado na página 38.

COLLANGE, S. et al. Barra: A parallel functional simulator for gpgpu. In: IEEE. *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*. [S.l.], 2010. p. 351–360. Citado na página 42.

DICELLO, J. F.; PACIOTTI, M.; SCHILLACI, M. An estimate of error rates in integrated circuits at aircraft altitudes and at sea level. *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, Elsevier, v. 40, p. 1295–1299, 1989. Citado na página 21.

DIMITROV, M.; MANTOR, M.; ZHOU, H. Understanding software approaches for gpgpu reliability. In: ACM. *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. [S.l.], 2009. p. 94–104. Citado na página 30.

DIXIT, A.; WOOD, A. The impact of new technology on soft error rates. In: IEEE. *Reliability Physics Symposium (IRPS), 2011 IEEE International*. [S.l.], 2011. p. 5B–4. Citado na página 14.

DIZ, G. *Ordenação bitônica*. 2010. Disponível em: <https://kuniga.wordpress.com/2010-/12/17/ordenacao-bitonica/>. Citado na página 39.

DODD, P. E.; MASSENGILL, L. W. Basic mechanisms and modeling of single-event upset in digital microelectronics. *IEEE Transactions on Nuclear Science*, IEEE, v. 50, n. 3, p. 583–602, 2003. Citado na página 15.

DODD, P. E. et al. Production and propagation of single-event transients in high-speed digital logic ics. *IEEE Transactions on Nuclear Science*, IEEE, v. 51, n. 6, p. 3278–3284, 2004. Citado na página 22.

ENTRENA, L. et al. Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection. *IEEE Transactions on Computers*, IEEE, v. 61, n. 3, p. 313–322, 2012. Citado na página 24.

FANG, B. et al. Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications. In: IEEE. *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. [S.l.], 2014. p. 221–230. Citado na página 29.

GRAPHICS, M. *ModelSim User Manual*. [S.l.]: Version, 2005. Citado na página 56.

HOU, Y.; LAI, J.; MIKUSHIN, D. *Asfermi: An assembler for the NVIDIA Fermi instruction set*. 2011. `https://github.com/hyqneuron/asfermi`. Citado na página 42.

HU, Q.; XIAO, B.; FRISWELL, M. Robust fault-tolerant control for spacecraft attitude stabilisation subject to input saturation. *IET Control Theory & Applications*, IET, v. 5, n. 2, p. 271–282, 2011. Citado na página 14.

HUANG, K.-H.; ABRAHAM, J. A. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, IEEE, v. 100, n. 6, p. 518–528, 1984. Citado na página 28.

KRÜGER, J.; WESTERMANN, R. Linear algebra operators for gpu implementation of numerical algorithms. In: ACM. *ACM Transactions on Graphics (TOG)*. [S.l.], 2003. v. 22, n. 3, p. 908–916. Citado na página 14.

LAAN, W. J. van der. *Cubin utilities*. 2007. `https://github.com/laanwj/decuda`. Citado na página 42.

LI, S.; FAROOQUI, N.; YALAMANCHILI, S. Software reliability enhancements for gpu applications. In: *Sixth Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2013)*. [S.l.: s.n.], 2013. Nenhuma citação no texto.

MAY, T. C.; WOODS, M. H. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, IEEE, v. 26, n. 1, p. 2–9, 1979. Citado na página 21.

MERCHANT, M. Testing and validation of a prototype gpgpu design for fpgas. 2013. Citado 4 vezes nas páginas 7, 31, 32 e 42.

NAPOLES, J. et al. Radiation environment emulation for vlsi designs: A low cost platform based on xilinx fpga's. In: IEEE. *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on*. [S.l.], 2007. p. 3334–3338. Citado na página 24.

NEDEL, W. M. Analise dos efeitos de falhas transientes no conjunto de banco de registradores em unidades gráficas de processamento. 2015. Citado na página 43.

NICOLESCU, B.; VELAZCO, R. Detecting soft errors by a purely software approach: method, tools and experimental results. In: *Embedded Software for SoC*. [S.l.]: Springer, 2003. p. 39–51. Citado na página 26.

NVIDIA. *CUDA Toolkit*. 2013. Disponível em: <https://developer.nvidia.com/cuda-toolkit>. Citado na página 37.

NVIDIA. *NVIDIA DRIVE™ PX*. 2016. Disponível em: <http://www.nvidia.com-/object/drive-px-request.html>. Citado 2 vezes nas páginas 14 e 20.

NVIDIA, C. Programming guide version 2.3. 1. *Nvidia Corporation*, 2009. Citado na página 38.

NVIDIA, C. *Programming Guide :: CUDA Toolkit Documentation*. 2017. Disponível em: <http://docs.nvidia.com/cuda/cuda-c-programming-guide>. Acesso em: 26 jun. 2017. Citado na página 16.

NVIDIA, F. Nvidia's next generation cuda compute architecture. *NVidia, Santa Clara, Calif, USA*, 2009. Citado na página 42.

OH, N.; SHIRVANI, P. P.; MCCLUSKEY, E. J. Control-flow checking by software signatures. *IEEE transactions on Reliability*, IEEE, v. 51, n. 1, p. 111–122, 2002. Citado na página 25.

OLIVEIRA, D. A. de; RECH, P.; NAVAUX, P. O. Experimental evaluation of hardening strategies for radiation-induced errors. 2013. Citado na página 29.

PILLA, L. et al. Software-based hardening strategies for neutron sensitive fft algorithms on gpus. *IEEE Transactions on Nuclear Science*, IEEE, v. 61, n. 4, p. 1874–1880, 2014. Citado na página 42.

RÁK, Á. *AMD-GPU-Asm-Disasm, 2011*. 2011. `https://github.com/rakadam/AMD-GPU-Asm-Disasm`. Citado na página 42.

REBAUDENGO, M. et al. Soft-error detection through software fault-tolerance techniques. In: IEEE. *Defect and Fault Tolerance in VLSI Systems, 1999. DFT'99. International Symposium on*. [S.l.], 1999. p. 210–218. Citado na página 26.

RECH, P. et al. Neutron radiation test of graphic processing units. In: IEEE. *2012 IEEE 18th International On-Line Testing Symposium (IOLTS)*. [S.l.], 2012. p. 55–60. Citado na página 15.

RECH, P. et al. An efficient and experimentally tuned software-based hardening strategy for matrix multiplication on gpus. *IEEE Transactions on Nuclear Science*, IEEE, v. 60, n. 4, p. 2797–2804, 2013. Citado 3 vezes nas páginas 14, 28 e 29.

REIS, G. A. et al. Swift: Software implemented fault tolerance. In: IEEE COMPUTER SOCIETY. *Proceedings of the international symposium on Code generation and optimization*. [S.l.], 2005. p. 243–254. Citado na página 26.

RHOD, E. L. et al. Hardware and software transparency in the protection of programs against seus and sets. *Journal of Electronic Testing*, Springer, v. 24, n. 1-3, p. 45–56, 2008. Citado na página 15.

SLAYMAN, C. Soft errors—past history and recent discoveries. In: IEEE. *2010 IEEE International Integrated Reliability Workshop Final Report*. [S.l.], 2010. p. 25–30. Citado na página 14.

STRANO, A. et al. Exploiting structural redundancy of simd accelerators for their built-in self-testing/diagnosis and reconfiguration. In: IEEE. *ASAP 2011-22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*. [S.l.], 2011. p. 141–148. Citado na página 14.

WILT, N. *The cuda handbook: A comprehensive guide to gpu programming*. [S.l.]: Pearson Education, 2013. Nenhuma citação no texto.

YIM, K. S. et al. Hauberk: Lightweight silent data corruption error detector for gpgpu. In: IEEE. *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. [S.l.], 2011. p. 287–300. Nenhuma citação no texto.

ZIEGLER, J. F. Terrestrial cosmic rays. *IBM journal of research and development*, IBM, v. 40, n. 1, p. 19–39, 1996. Citado na página 21.

ANEXO A – Códigos Fonte Utilizados

A.1 CUDA Kernel em Assembly: Reduction.sass

```
1     /*0000*/   /*0xa000420904200780*/   I2I.U32.U16 R2, g [0x1].U16;
2     /*0008*/   /*0xa0004c0504200780*/   I2I.U32.U16 R1, g [0x6].U16;
3     /*0010*/   /*0x4004041100000780*/   IMUL.U16.U16 R4, R1L, R2L;
4     /*0018*/   /*0xa000000d04000780*/   I2I.U32.U16 R3, R0L;
5     /*0020*/   /*0x30010801c4100780*/   SHL R0, R4, 0x1;
6     /*0028*/   /*0x20008600        */   IADD32 R0, R3, R0;
7     /*002c*/   /*0x20008410        */   IADD32 R4, R2, R0;
8     /*0030*/   /*0x30020001c4100780*/   SHL R0, R0, 0x2;
9     /*0038*/   /*0x30020811c4100780*/   SHL R4, R4, 0x2;
10    /*0040*/   /*0x2100e800        */   IADD32 R0, g [0x4], R0;
11    /*0044*/   /*0x2104e814        */   IADD32 R5, g [0x4], R4;
12    /*0048*/   /*0xd00e001180c00780*/   GLD.U32 R4, global14 [R0];
13    /*0050*/   /*0xd00e0a0180c00780*/   GLD.U32 R0, global14 [R5];
14    /*0058*/   /*0x2000080104000780*/   IADD R0, R4, R0;
15    /*0060*/   /*0x00020605c0000780*/   R2A A1, R3, 0x2;
16    /*0068*/   /*0x04001001e4200780*/   R2G.U32.U32 g [A1+0x8], R0;
17    /*0070*/   /*0x861ffe0300000000*/   BAR.ARV.WAIT b0, 0xfff;
18    /*0078*/   /*0x30010409e41007c0*/   SHR.C0 R2, R2, 0x1;
19    /*0080*/   /*0x1001b00300000100*/   BRA C0.EQ, 0xd8;
20    /*0088*/   /*0x300305fd6400c7c8*/   ISET.C0 o [0x7f], R2, R3, LE;
21    /*0090*/   /*0x2000060104008500*/   IADD R0 (C0.EQU), R3, R2;
22    /*0098*/   /*0x00020009c0000500*/   R2A A2 (C0.EQU), R0, 0x2;
23    /*00a0*/   /*0x00020605c0000500*/   R2A A1 (C0.EQU), R3, 0x2;
24    /*00a8*/   /*0x1800d0010423c500*/   MOV R0 (C0.EQU), g [A2+0x8];
25    /*00b0*/   /*0x2400d00104200500*/   IADD R0 (C0.EQU), g [A1+0x8], R0;
26    /*00b8*/   /*0x04001001e4200500*/   R2G.U32.U32 g [A1+0x8] (C0.EQU), R0;
27    /*00c0*/   /*0x861ffe0300000000*/   BAR.ARV.WAIT b0, 0xfff;
28    /*00c8*/   /*0x30010409e41007c0*/   SHR.C0 R2, R2, 0x1;
29    /*00d0*/   /*0x1001100300000280*/   BRA C0.NE, 0x88;
30    /*00d8*/   /*0x307c07fd640147c8*/   ISET.C0 o [0x7f], R3, R124, NE;
31    /*00e0*/   /*0x3000000300000280*/   RET C0.NE;
32    /*00e8*/   /*0x30020205c4100780*/   SHL R1, R1, 0x2;
33    /*00f0*/   /*0x1100f000        */   MOV32 R0, g [0x8];
34    /*00f4*/   /*0x2101ea04        */   IADD32 R1, g [0x5], R1;
35    /*00f8*/   /*0xd00e0201a0c00781*/   GST.U32 global14 [R1], R0;
36    /*0100*/   /*0x3000000300000780*/   RET;
```

A.2 CUDA Kernel em Assembly: Autocorr.sass

```
1   /*0000*/  /*0x100042050023c780*/  MOV.U16 R0H, g [0x1].U16;
2   /*0008*/  /*0xa000000504000780*/  I2I.U32.U16 R1, R0L;
3   /*0010*/  /*0x60014c0100204780*/  IMAD.U16 R0, g [0x6].U16, R0H, R1;
4   /*0018*/  /*0x3000cdfd6c20c7c8*/  ISET.S32.C0 o [0x7f], g [0x6], R0, LE;
5   /*0020*/  /*0x3000000300000280*/  RET C0.NE;
6   /*0028*/  /*0x2040cc0504200780*/  IADD R1, g [0x6], -R0;
7   /*0030*/  /*0x307c03fd6c00c7c8*/  ISET.S32.C0 o [0x7f], R1, R124, LE;
8   /*0038*/  /*0xa001b00300000000*/  SSY 0xd8;
9   /*0040*/  /*0x1001a00300000280*/  BRA C0.NE, 0xd0;
10  /*0048*/  /*0x3002000dc4100780*/  SHL R3, R0, 0x2;
11  /*0050*/  /*0x10008004        */  MOV32 R1, R0;
12  /*0054*/  /*0x1100e808        */  MOV32 R2, g [0x4];
13  /*0058*/  /*0x1000f8150403c780*/  MOV R5, R124;
14  /*0060*/  /*0x2000c8190420c780*/  IADD R6, g [0x4], R3;
15  /*0068*/  /*0xd00e040d80c00780*/  GLD.U32 R3, global14 [R2];
16  /*0070*/  /*0xd00e0c1180c00780*/  GLD.U32 R4, global14 [R6];
17  /*0078*/  /*0x40090c1d00000780*/  IMUL.U16.U16 R7, R3L, R4H;
18  /*0080*/  /*0x60080e1d0001c780*/  IMAD.U16 R7, R3H, R4L, R7;
19  /*0088*/  /*0x30100e1dc4100780*/  SHL R7, R7, 0x10;
20  /*0090*/  /*0x2001820500000003*/  IADD32I R1, R1, 0x1;
21  /*0098*/  /*0x60080c0d0001c780*/  IMAD.U16 R3, R3L, R4L, R7;
22  /*00a0*/  /*0x3001cdfd6c2147c8*/  ISET.S32.C0 o [0x7f], g [0x6], R1, NE;
23  /*00a8*/  /*0x2000061504014780*/  IADD R5, R3, R5;
24  /*00b0*/  /*0x20048c1900000003*/  IADD32I R6, R6, 0x4;
25  /*00b8*/  /*0x2004840900000003*/  IADD32I R2, R2, 0x4;
26  /*00c0*/  /*0x1000d00300000280*/  BRA C0.NE, 0x68;
27  /*00c8*/  /*0x1001b00300000780*/  BRA 0xd8;
28  /*00d0*/  /*0x1000f8150403c780*/  MOV R5, R124;
29  /*00d8*/  /*0xf0000001e0000002*/  NOP.S;
30  /*00e0*/  /*0x861ffe0300000000*/  BAR.ARV.WAIT b0, 0xfff;
31  /*00e8*/  /*0x30020001c4100780*/  SHL R0, R0, 0x2;
32  /*00f0*/  /*0x2000ca0104200780*/  IADD R0, g [0x5], R0;
33  /*00f8*/  /*0xd00e0015a0c00781*/  GST.U32 global14 [R0], R5;
34  /*0100*/  /*0x3000000300000780*/  RET;
```

A.3 CUDA Kernel em Assembly: MulMat.sass

```
1
2    /*0000*/    /*0xd080020500410780*/    LOP.AND.U16 R0H, R0H, c [0x1] [0x0];
3    /*0008*/    /*0x307ccffd6c20c7c8*/    ISET.S32.C0 o [0x7f], g [0x7], R124, LE;
4    /*0010*/    /*0xa000000904000780*/    I2I.U32.U16 R2, R0L;
5    /*0018*/    /*0xa000020504000780*/    I2I.U32.U16 R1, R0H;
6    /*0020*/    /*0x1002300300000280*/    BRA C0.NE, 0x118;
7    /*0028*/    /*0x10004e010023c780*/    MOV.U16 R0L, g [0x7].U16;
8    /*0030*/    /*0x6000440d00204780*/    IMAD.U16 R3, g [0x2].U16, R0L, R1;
9    /*0038*/    /*0x1000ce050423c780*/    MOV R1, g [0x7];
10   /*0040*/    /*0x4007040100000780*/    IMUL.U16.U16 R0, R1L, R3H;
11   /*0048*/    /*0x6006061100000780*/    IMAD.U16 R4, R1H, R3L, R0;
12   /*0050*/    /*0x10004c010023c780*/    MOV.U16 R0L, g [0x6].U16;
13   /*0058*/    /*0x30100811c4100780*/    SHL R4, R4, 0x10;
14   /*0060*/    /*0x6000421500208780*/    IMAD.U16 R5, g [0x1].U16, R0L, R2;
15   /*0068*/    /*0x6006040500010780*/    IMAD.U16 R1, R1L, R3L, R4;
16   /*0070*/    /*0xa002100300000000*/    SSY 0x108;
17   /*0078*/    /*0x30020a01c4100780*/    SHL R0, R5, 0x2;
18   /*0080*/    /*0x3002ce09c4300780*/    SHL R2, g [0x7], 0x2;
19   /*0088*/    /*0x1000f8190403c780*/    MOV R6, R124;
20   /*0090*/    /*0x3002020dc4100780*/    SHL R3, R1, 0x2;
21   /*0098*/    /*0x2100ea20        */    IADD32 R8, g [0x5], R0;
22   /*009c*/    /*0x10008200        */    MOV32 R0, R1;
23   /*00a0*/    /*0x2101ee1c        */    IADD32 R7, g [0x7], R1;
24   /*00a4*/    /*0x2103e824        */    IADD32 R9, g [0x4], R3;
25   /*00a8*/    /*0xd00e100d80c00780*/    GLD.U32 R3, global14 [R8];
26   /*00b0*/    /*0xd00e121180c00780*/    GLD.U32 R4, global14 [R9];
27   /*00b8*/    /*0x40090c2900000780*/    IMUL.U16.U16 R10, R3L, R4H;
28   /*00c0*/    /*0x60080e2900028780*/    IMAD.U16 R10, R3H, R4L, R10;
29   /*00c8*/    /*0x30101429c4100780*/    SHL R10, R10, 0x10;
30   /*00d0*/    /*0x2001820500000003*/    IADD32I R1, R1, 0x1;
31   /*00d8*/    /*0x60080c0d00028780*/    IMAD.U16 R3, R3L, R4L, R10;
32   /*00e0*/    /*0x300703fd6c0147c8*/    ISET.S32.C0 o [0x7f], R1, R7, NE;
33   /*00e8*/    /*0x2000061904018780*/    IADD R6, R3, R6;
34   /*00f0*/    /*0x2004922500000003*/    IADD32I R9, R9, 0x4;
35   /*00f8*/    /*0x2000102104008780*/    IADD R8, R8, R2;
36   /*0100*/    /*0x1001500300000280*/    BRA C0.NE, 0xa8;
37   /*0108*/    /*0xf0000001e0000002*/    NOP.S;
38   /*0110*/    /*0x1002d00300000780*/    BRA 0x168;
39   /*0118*/    /*0x10004e010023c780*/    MOV.U16 R0L, g [0x7].U16;
40   /*0120*/    /*0x6000440d00204780*/    IMAD.U16 R3, g [0x2].U16, R0L, R1;
41   /*0128*/    /*0x1000ce010423c780*/    MOV R0, g [0x7];
42   /*0130*/    /*0x4007000500000780*/    IMUL.U16.U16 R1, R0L, R3H;
43   /*0138*/    /*0x6006020500004780*/    IMAD.U16 R1, R0H, R3L, R1;
44   /*0140*/    /*0x30100211c4100780*/    SHL R4, R1, 0x10;
45   /*0148*/    /*0x10004c090023c780*/    MOV.U16 R1L, g [0x6].U16;
46   /*0150*/    /*0x6006000100010780*/    IMAD.U16 R0, R0L, R3L, R4;
47   /*0158*/    /*0x6002421500208780*/    IMAD.U16 R5, g [0x1].U16, R1L, R2;
48   /*0160*/    /*0x1000f8190403c780*/    MOV R6, R124;
49   /*0168*/    /*0x2000000104014780*/    IADD R0, R0, R5;
50   /*0170*/    /*0x30020001c4100780*/    SHL R0, R0, 0x2;
51   /*0178*/    /*0x2000cc0104200780*/    IADD R0, g [0x6], R0;
52   /*0180*/    /*0xd00e0019a0c00781*/    GST.U32 global14 [R0], R6;
53   /*0188*/    /*0x3000000300000780*/    RET;
```

A.4 CUDA Kernel em Assembly: BitonicSort.sass

```

        /*0000*/     /*0xa000000d04000780*/     I2I.U32.U16 R3, R0L;
        /*0008*/     /*0x30020601c4100780*/     SHL R0, R3, 0x2;
        /*0010*/     /*0x2000c80904200780*/     IADD R2, g [0x4], R0;
        /*0018*/     /*0x00020605c0000780*/     R2A A1, R3, 0x2;
        /*0020*/     /*0xd00e040180c00780*/     GLD.U32 R0, global14 [R2];
        /*0028*/     /*0x04001001e4200780*/     R2G.U32.U32 g [A1+0x8], R0;
        /*0030*/     /*0x861ffe0300000000*/     BAR.ARV.WAIT b0, 0xfff;
        /*0038*/     /*0x1002801100000003*/     MVI R4, 0x2;
        /*0040*/     /*0x30010815e41007c0*/     SHR.C0 R5, R4, 0x1;
        /*0048*/     /*0x1002700300000100*/     BRA C0.EQ, 0x130;
        /*0050*/     /*0xd0030a0104008780*/     LOP.XOR R0, R5, R3;
        /*0058*/     /*0x300301fd6400c7c8*/     ISET.C0 o [0x7f], R0, R3, LE;
        /*0060*/     /*0x00020005c0000780*/     R2A A1, R0, 0x2;
        /*0068*/     /*0xA002300300000000*/     SSY 0x110;
        /*0070*/     /*0x1002300300000280*/     BRA C0.NE, 0x110;
        /*0078*/     /*0xd00309fd040007c8*/     LOP.AND.C0 o [0x7f], R4, R3;
        /*0080*/     /*0x1001b00300000280*/     BRA C0.NE, 0xd0;
        /*0088*/     /*0x00020609c0000780*/     R2A A2, R3, 0x2;
        /*0090*/     /*0x1400d0010423c780*/     MOV R0, g [A1+0x8];
        /*0098*/     /*0x3800d1fd6c20c7c8*/     ISET.S32.C0 o [0x7f], g [A2+0x8], R0, LE;
        /*00a0*/     /*0x1002300300000280*/     BRA C0.NE, 0x110;
        /*00a8*/     /*0x00020609c0000780*/     R2A A2, R3, 0x2;
        /*00b0*/     /*0x1500f004        */     MOV32 R1, g [A1+0x8];
        /*00b4*/     /*0x1900f000        */     MOV32 R0, g [A2+0x8];
        /*00b8*/     /*0x08001001e4204780*/     R2G.U32.U32 g [A2+0x8], R1;
        /*00c0*/     /*0x04001001e4200780*/     R2G.U32.U32 g [A1+0x8], R0;
        /*00c8*/     /*0x1002300300000780*/     BRA 0x110;
        /*00d0*/     /*0x00020609c0000780*/     R2A A2, R3, 0x2;
        /*00d8*/     /*0x1400d0010423c780*/     MOV R0, g [A1+0x8];
        /*00e0*/     /*0x3800d1fd6c2187c8*/     ISET.S32.C0 o [0x7f], g [A2+0x8], R0, GE;
        /*00e8*/     /*0x00020609c0000500*/     R2A A2 (C0.EQU), R3, 0x2;
        /*00f0*/     /*0x1400d0050423c500*/     MOV R1 (C0.EQU), g [A1+0x8];
        /*00f8*/     /*0x1800d0010423c500*/     MOV R0 (C0.EQU), g [A2+0x8];
        /*0100*/     /*0x08001001e4204500*/     R2G.U32.U32 g [A2+0x8] (C0.EQU), R1;
        /*0108*/     /*0x04001001e4200500*/     R2G.U32.U32 g [A1+0x8] (C0.EQU), R0;
        /*0110*/     /*0xf0000001e0000002*/     NOP.S;
        /*0118*/     /*0x861ffe0300000000*/     BAR.ARV.WAIT b0, 0xfff;
        /*0120*/     /*0x30010a15e41007c0*/     SHR.C0 R5, R5, 0x1;
        /*0128*/     /*0x1000b00300000280*/     BRA C0.NE, 0x50;
        /*0130*/     /*0x30010811c4100780*/     SHL R4, R4, 0x1;
        /*0138*/     /*0x308009fd6440c7c8*/     ISET.C0 o [0x7f], R4, c [0x1] [0x0], LE;
        /*0140*/     /*0x1000900300000280*/     BRA C0.NE, 0x40;
        /*0148*/     /*0x00020605c0000780*/     R2A A1, R3, 0x2;
        /*0150*/     /*0x1400d0010423c780*/     MOV R0, g [A1+0x8];
        /*0158*/     /*0xd00e0401a0c00781*/     GST.U32 global14 [R2], R0;
        /*0160*/     /*0x3000000300000780*/     RET;
```

ANEXO B – Conjunto de Instruções

Tabela 19 – Conjunto de instruções suportadas e testadas na FlexGrip

Opcode	**Descrição**
I2I	Copia valor inteiro com conversão
IMUL/IMUL32/IMUL32I	Multiplicação inteira
SHL	Deslocamento para a esquerda
IADD	Adição inteira entre dois registradores
GLD	Carregar dado da memória global
R2A	Mover registrador para o registrador de endereços
A2R	Mover de registrador de endereços para registrador de dados
R2G	Salvar dado na memória compartilhada
BAR	Barreira de sincronização
SHR	Deslocamento para a direita
BRA	Desvio incondicional
ISET	Set inteiro condicional
MOV/MOV32	Mover registrador para registrador
RET	Retorno condicional do kernel
MOV R, S[]	Carregar dado da memória compartilhada
IADD S[], R	Adição inteira entre mem compartilhada e registrador
GST	Salvar dado na memória global
AND C[], R	"E"lógico
IMAD/IMAD32	Multiplicação-adição inteira
SSY	Marcar ponto de sincronização
IADDI	Adição inteira com operando intermediário
NOP	Nenhuma operação
@P	Execução preditiva
MVI	Mover imediato para destino
XOR	XOR lógico
IMADI/IMAD32I	Multiplicação-adição inteira com operando imediato
LLD	Carregar da memória local
LST	Salvar na memória local

ANEXO C – Registradores do Pipeline

E1 - Warp to Fetch

1 program_cntr_wf_reg_d
2 warp_id_wf_reg_d
3 warp_lane_id_wf_reg_d
4 cta_id_wf_reg_d
5 initial_mask_wf_reg_d
6 current_mask_wf_reg_d
7 shmem_base_addr_wf_reg_d
8 gprs_size_wf_reg_d
9 gprs_base_addr_wf_reg_d

E2 - Fetch to Decode

10 instruction_fd_reg_d
11 program_cntr_fd_reg_d
12 next_pc_fd_reg_d
13 warp_id_fd_reg_d
14 warp_lane_id_fd_reg_d
15 cta_id_fd_reg_d
16 initial_mask_fd_reg_d
17 current_mask_fd_reg_d
18 shmem_base_addr_fd_reg_d
19 gprs_size_fd_reg_d
20 gprs_base_addr_fd_reg_d

E3 - Decode to Read

21 program_cntr_dr_reg_d
22 next_pc_dr_reg_d
23 warp_id_dr_reg_d
24 warp_lane_id_dr_reg_d
25 cta_id_dr_reg_d
26 initial_mask_dr_reg_d
27 current_mask_dr_reg_d
28 shmem_base_addr_dr_reg_d
29 gprs_size_dr_reg_d
30 gprs_base_addr_dr_reg_d
31 instr_opcode_type_dr_reg_d
32 instr_subop_type_dr_reg_d
33 alu_opcode_dr_reg_d
34 mov_opcode_dr_reg_d
35 flow_opcode_dr_reg_d
36 instr_marker_type_dr_reg_d
37 instr_src1_shared_dr_reg_d
38 instr_src2_const_dr_reg_d
39 instr_src3_const_dr_reg_d
40 pred_reg_dr_reg_d
41 pred_cond_dr_reg_d
42 set_pred_dr_reg_d
43 set_pred_reg_dr_reg_d
44 write_pred_dr_reg_d
45 is_full_normal_dr_reg_d
46 is_signed_dr_reg_d
47 w32_dr_reg_d
48 saturate_dr_reg_d
49 abs_saturate_dr_reg_d
50 cvt_round_dr_reg_d
51 cvt_type_dr_reg_d
52 cvt_neg_dr_reg_d
53 condition_dr_reg_d
54 addr_hi_dr_reg_d
55 addr_lo_dr_reg_d
56 addr_incr_dr_reg_d
57 mov_size_dr_reg_d
58 const_segment_dr_reg_d
59 mem_type_dr_reg_d
60 sm_type_dr_reg_d
61 imm_hi_dr_reg_d
62 addr_imm_dr_reg_d
63 src1_shared_dr_reg_d
64 src1_mem_type_dr_reg_d
65 src2_mem_type_dr_reg_d
66 src3_mem_type_dr_reg_d
67 dest_mem_type_dr_reg_d
68 src1_mem_opcode_type_dr_reg_d
69 src2_mem_opcode_type_dr_reg_d
70 src3_mem_opcode_type_dr_reg_d
71 dest_mem_opcode_type_dr_reg_d
72 src1_neg_dr_reg_d
73 src2_neg_dr_reg_d
74 src3_neg_dr_reg_d
75 target_addr_dr_reg_d
76 src1_data_type_dr_reg_d
77 src2_data_type_dr_reg_d
78 src3_data_type_dr_reg_d
79 dest_data_type_dr_reg_d
80 src1_dr_reg_d
81 src2_dr_reg_d
82 src3_dr_reg_d
83 dest_dr_reg_d
84 warp_id_re_reg_d
85 warp_lane_id_re_reg_d
86 cta_id_re_reg_d
87 initial_mask_re_reg_d

E4 - Read to Execute

88 current_mask_re_reg_d
89 shmem_base_addr_re_reg_d
90 gprs_size_re_reg_d
91 gprs_base_addr_re_reg_d
92 instruction_mask_re_reg_d
93 next_pc_re_reg_d
94 instr_opcode_type_re_reg_d
95 instr_subop_type_re_reg_d
96 alu_opcode_re_reg_d
97 flow_opcode_type_re_reg_d
98 mov_opcode_type_re_reg_d
99 instr_marker_type_re_reg_d
100 set_pred_re_reg_d
101 set_pred_reg_re_reg_d
102 write_pred_re_reg_d
103 is_full_normal_re_reg_d
104 is_signed_re_reg_d
105 w32_re_reg_d
106 f64_re_reg_d
107 saturate_re_reg_d
108 abs_saturate_re_reg_d
109 cvt_round_re_reg_d
110 cvt_type_re_reg_d
111 cvt_neg_re_reg_d
112 condition_re_reg_d
113 addr_hi_re_reg_d
114 addr_lo_re_reg_d
115 addr_incr_re_reg_d
116 mov_size_re_reg_d
117 mem_type_re_reg_d
118 sm_type_re_reg_d
119 addr_imm_re_reg_d
120 dest_mem_type_re_reg_d
121 dest_mem_opcode_type_re_reg_d
122 src1_neg_re_reg_d
123 src2_neg_re_reg_d
124 src3_neg_re_reg_d
125 target_addr_re_reg_d
126 dest_data_type_re_reg_d
127 src1_re_reg_d
128 dest_re_reg_d
129 pred_flags_re_reg_d

E5 - Execute to Write

130 warp_id_ew_reg_d
131 warp_lane_id_ew_reg_d
132 cta_id_ew_reg_d
133 initial_mask_ew_reg_d
134 current_mask_ew_reg_d
135 shmem_base_addr_ew_reg_d
136 gprs_size_ew_reg_d
137 gprs_base_addr_ew_reg_d
138 instr_mask_ew_reg_d
139 next_pc_ew_reg_d
140 warp_state_ew_reg_d
141 instr_opcode_type_ew_reg_d
142 set_pred_ew_reg_d
143 set_pred_reg_ew_reg_d
144 write_pred_ew_reg_d
145 addr_hi_ew_reg_d
146 addr_lo_ew_reg_d
147 addr_incr_ew_reg_d
148 mov_size_ew_reg_d
149 sm_type_ew_reg_d
150 addr_imm_ew_reg_d
151 src1_ew_reg_d
152 dest_mem_type_ew_reg_d
153 dest_mem_opcode_type_ew_reg_d
154 dest_data_type_ew_reg_d
155 dest_ew_reg_d
156 pred_flags_ew_reg_d

E6 - Write to Warp

157 warp_id_ww_reg_d
158 warp_lane_id_ww_reg_d
159 cta_id_ww_reg_d
160 initial_mask_ww_reg_d
161 current_mask_ww_reg_d
162 shmem_base_addr_ww_reg_d
163 gprs_base_addr_ww_reg_d
164 next_pc_ww_reg_d
165 warp_state_ww_reg_d

Figura 46 – Lista de registradores do *pipeline*