UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GIANI AUGUSTO BRAGA

# Técnicas de Tolerância a Falhas Controladas por Software para a Proteção do Pipeline de Processadores Gráficos

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Orientador: Prof. Dr. José Rodrigo de Azambuja

Porto Alegre
2023

**CIP — CATALOGAÇÃO NA PUBLICAÇÃO**

Braga, Giani Augusto

Técnicas de Tolerância a Falhas Controladas por Software para a Proteção do Pipeline de Processadores Gráficos / Giani Augusto Braga. – Porto Alegre: PPGC da UFRGS, 2023.

76 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2023. Orientador: José Rodrigo de Azambuja.

1. Tolerância a falhas. 2. Unidades de processamento gráficos. 3. Técnicas de mitigação seletiva. I. Azambuja, José Rodrigo de. II. Título.

## AGRADECIMENTOS

Agradecimento especial a minha mãe, Leonice, às minhas tias, Edi e Rejane, às minhas irmãs, Sabrina e Andressa, e meus sobrinhos, Andrei, Cleo e Micheli. O apoio de vocês foi essencial ao longo dessa trajetória. Agradeço por estarem sempre ao meu lado, compreendendo minhas decisões e respeitando minha ausência.

Ao professor Dr. José Rodrigo Azambuja, orientador desse trabalho, por sua competência e auxílio no decorrer desse caminho de estudo e pesquisa. Pela disponibilidade sempre demonstrada, pelo desafio lançado e pela motivação constante para que fosse possível a realização dessa dissertação. Agradeço também por aceitar me orientar no Doutorado.

Ao professor Dr. Ademar Antonio Lauxen agradeço por sua amizade, e pelo constante estímulo ao longo da minha trajetória acadêmica.

À Universidade Federal do Rio Grande do Sul, especialmente ao Programa de Pós-Graduação em Ciência da Computação, à sua coordenação, aos seus professores e funcionários, e ao suporte financeiro do CNPq.

A todos aqueles que, de alguma forma, contribuíram para a concretização deste estudo: amigos, familiares e colegas.

## RESUMO

A utilização de Processadores Gráficos (*Graphics Processing Unit* - GPU) na computação gráfica, em aceleradores de uso geral e Computação de Alto Desempenho (*High Processing Computing* - HPC), recentemente tiveram um crescimento e passaram a ser utilizados em diversas aplicações críticas de segurança, por exemplo em veículos autônomos e aviônica. Embora, as mais recentes tecnologias são utilizadas na fabricação das GPUs para satisfazer os requisitos de consumo de energia e desempenho, ainda são sensíveis e suscetíveis a falhas em algumas áreas, dentre elas a aviônica, por possuir um alto grau de exposição a partículas energizadas, como prótons e nêutrons. Os principais efeitos causados por essas partículas energizadas, em circuitos de alta densidade, são conhecidos como Perturbações de Evento Único (*Single Event Upset* - SEU). Apesar do SEU não resultar na destruição dos circuitos, ele tem o potencial de introduzir erros no armazenamento de dados, afetando principalmente memórias e registradores. Para proteger as GPUs contra esses efeitos, os engenheiros empregam técnicas de tolerância a falhas, que podem ser desenvolvidas por meio de abordagens que combinam o suporte de software e hardware.
O objetivo dessa dissertação foi aprimorar a resiliência dos registradores de pipeline em uma arquitetura de GPU. Para isso, foram desenvolvidas duas técnicas híbridas de tolerância a falhas, baseadas em trabalhos relacionados. A primeira técnica é a híbrida XOR e a segunda técnica é a híbrida paridade, que comparam e detectam, por meio de um bit de confiabilidade, se a instrução duplicada está com erro, caso estiver é realizado a correção. Abordagens anteriores concentraram-se em proteger elementos de memória, como arquivos de registradores e memória compartilhada, priorizando proteção por software. Já a proteção dos registradores de pipeline demanda alterações no hardware, tornando essencial o desenvolvimento de técnicas híbridas.
A análise das técnicas desenvolvidas foram realizadas por meio de simulação da injeção de milhares de falhas no pipeline da GPU, em até seis aplicações de estudo de caso. Os resultados em termos de *overhead* do tempo de execução variam de 1,04x a 1,66x e uma detecção e correção de erros de 100% para as quatro aplicações de estudo de caso da técnica híbrida XOR. Para a técnica híbrida paridade, teve um aumento entre 2% e 15% no *overhead* do tempo de execução, além da redução e correção de em média 47% dos erros.


**Palavras-chave:** Tolerância a falhas. unidades de processamento gráficos. técnicas de mitigação seletiva.

**Software-Controlled Fault Tolerance Techniques for Hardening the Pipelines of Graphics Processor Units**

## ABSTRACT

The use of Graphics Processors (GPU) in graphics computing, in general purpose accelerators and High Performance Computing (HPC), has recently grown and become used in several safety-critical applications, for example in autonomous vehicles and avionics. Although the latest technologies are used in the manufacture of GPUs to satisfy energy consumption and performance requirements, they are still sensitive and susceptible to failures in some areas, including avionics, due to a high degree of exposure to energized particles, such as protons and neutrons. The main effects caused by these energized particles, in high-density circuits, are known as Single Event Upset (SEU). Although SEU does not result in the destruction of circuits, it has the potential to introduce errors into data storage, mainly affecting memories and registers. To protect GPUs against these effects, engineers employ fault tolerance techniques, which can be developed through approaches that combine software and hardware support.

The objective of this dissertation was to improve the resilience of pipeline registers in a GPU architecture. To this end, two hybrid fault tolerance techniques were developed, based on related work. The first technique is hybrid XOR and the second technique is hybrid parity, which compare and detect, through a reliability bit, whether the duplicate instruction has an error, and if so, correction is performed. Previous approaches have focused on protecting memory elements such as register files and shared memory, prioritizing software protection. Protecting pipeline registers requires changes to the hardware, making the development of hybrid techniques essential.

The analysis of the developed techniques was carried out through simulation of the injection of thousands of faults into the GPU pipeline, in up to six case study applications. The results in terms of runtime overhead range from 1.04x to 1.66x and a 100% error detection and correction for the four case study applications of the hybrid XOR technique. For the hybrid parity technique, there was an increase between 2% and 15% in overhead execution time, in addition to the reduction and correction of an average of 47% of errors.

**Keywords:** fault tolerance, graphics processing units, selective mitigation techniques.

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| ALU | Arithmetic Logic Unit |
| BFI | Branch Free Interval |
| BID | Branch Free Interval Identifier |
| CCA | Control Flow Checking using Assertions |
| CFID | Control Flow Identifier |
| CPU | Central Processing Unit |
| DMR | Dual Modular Redundancy |
| DUE | Detected Unrecoverable Error |
| DWC | Duplication With Comparison |
| ECC | Error correction code |
| ECCA | Enhanced Control Flow Checking using Assertions |
| EDDI | Error Detection by Duplicated Instructions |
| FFT | Transformação rápida de Fourier |
| GEMM | General Matrix Multiplication |
| GPRF | General-Purpose Register Files |
| GPU | Graphics Processing Unit |
| HPC | High Processing Computing |
| HPCT | Hardening Post Compile Translator Tool |
| IC | Integrated Circuit |
| ISA | Instruction Set Architecture |
| JRE | Java Runtime Environment |
| MAD | Multiplication-Addition |
| MOS | Metal-Oxide-Semiconductor |
| PCHT | Post-Compiler Hardening Tool |

| | |
|---|---|
| PRF | Predicate Register Files |
| PR | Pipeline Registers |
| PTX | Parallel Thread Execution |
| SASS | Streaming ASSembler |
| SBST | Search Based Software Testing |
| SDC | Silent Data Corruption |
| SECDED | Single Error Correction Double Error Detection |
| SEEs | Single Event Effects |
| SEL | Single Event Latchup |
| SET | Single Event Transient |
| SEU | Single Event Upset |
| SIMT | Single-Instruction Multiple-Thread |
| SInRG | Software-managed Instruction Replication for GPUs |
| SM | Streaming Multiprocessors |
| SoR | Sphere-of-Replication |
| SP | Scalar Processors |
| SWIFT | Software Implemented Fault Tolerance |
| TID | Total Ionizing Dose |
| TLP | Thread-level Parallelism |
| TMR | Triple Modular Redundancy |

## LISTA DE FIGURAS

## LISTA DE TABELAS

**SUMÁRIO**

# 1 INTRODUÇÃO

Nos últimos anos, temos presenciado um notável aumento no número de aplicações, especialmente na área de inteligência artificial, por exemplo redes neurais voltadas para o processamento de imagens e vídeos (SHABBAR, 2022; AYLLON et al., 2019). Tais aplicações demandam um maior poder de processamento devido à sua grande quantidade de operações e parâmetros. Uma solução viável para melhorar o desempenho e reduzir o tempo de processamento é a paralelização das aplicações entre os núcleos dos processadores. As GPUs são particularmente adequadas para esse propósito, pois suportam um grande número de núcleos e podem executar aplicações em paralelo, isso graças ao seu Paralelismo a Nível de Threads (*Thread-level Parallelism* - TLP). Além disso, devido à sua capacidade de processar blocos de dados extensos de forma paralela, estudos indicam que o desempenho máximo das GPUs supera o dos Processadores de Propósito Geral (*General Purpose Processors* - GPPs) (KRÜGER; WESTERMANN, 2005; ASANO; MARUYAMA; YAMAGUCHI, 2009).

As GPUs são adotadas em uma variedade de aplicações, principalmente para as críticas de segurança, dentre elas veículos autônomos, aviônica e exploração espacial, e também, em outras áreas como HPC e aceleradores de uso geral (CONDIA et al., 2020a). As tecnologias mais avançadas são aplicadas por engenheiros na fabricação de GPUs (CONDIA et al., 2020a). No entanto, é importante observar que as GPUs são suscetíveis a falhas em determinadas áreas, dentre elas na aviônica e exploração espacial, devido à exposição a partículas energizadas, como prótons e nêutrons. Quando tais partículas interagem com os componentes eletrônicos no decorrer da execução das aplicações, podem resultar em falhas. Isso tem levado os engenheiros a adotarem técnicas de tolerância a falhas para aprimorar a confiabilidade das GPUs.

Falhas em componentes eletrônicos resultam principalmente da interação com partículas energizadas provenientes da atividade solar e raios cósmicos, com potencial para gerar efeitos tanto permanentes quanto temporários. A probabilidade de uma partícula energizada causar uma alteração em um Circuito Integrado (*Integrated Circuit* - IC) é influenciada por diversos fatores, incluindo a densidade dos transistores (ICs mais densos apresentam maior suscetibilidade a alterações nos transistores por uma única partícula), a frequência operacional (frequências mais elevadas resultam em janelas de vulnerabilidade mais estreitas) e a tensão de alimentação (tensões de limiar mais baixas requerem menos energia para desencadear uma alteração). (DIXIT; WOOD, 2011) (RECH et al., 2013).

Um dos principais efeitos de falhas transitórias observadas em circuitos de alta densidade, causadas por partículas energizadas, é a Perturbação de Evento Único (*Single Event Upset* - SEU), que é classificado como um erro suave (*Soft Error*) e frequentemente chamado de bit-flip. Ao contrário dos efeitos destrutivos, o SEU não danifica fisicamente os circuitos, mas resulta em erros de armazenamento, memória e registradores. Seus impactos podem resultar em erros durante a execução das aplicações nas GPUs. O SEU desencadeia dois principais efeitos: o primeiro é a Corrupção Silenciosa de Dados (*Silent Data Corruption* - SDC), caracterizado pela execução normal do código da aplicação, porém sem obtenção do resultado final esperado. Esse efeito compromete diretamente a precisão dos resultados de aplicações críticas para a segurança. O segundo efeito é o Erro Irrecuperável Detectado (*Detected Unrecoverable Error* - DUE), que ocorre quando a aplicação é interrompida de maneira inesperada durante sua execução ou entra em um *loop* infinito. Esse efeito impacta diretamente as restrições de tempo das aplicações de tempo real. Para mitigar tais efeitos e proteger as GPUs, engenheiros devem empregar técnicas de tolerância a falhas, podendo ser implementadas por meio de suporte de software e hardware.

Com o objetivo de proteger as GPUs contra esses efeitos transitórios induzidos por radiação, os projetistas têm se dedicado ao estudo de técnicas de tolerância a falhas. Como exemplos, temos a técnica de Redundância Modular Dupla (*Dual Modular Redundancy* - DMR) (ALCAIDE et al., 2019), que duplica toda a aplicação, e a técnica de Redundância Modula Tripla (*Triple Modular Redundancy* - TMR) (PORTET et al., 2020), que triplica a aplicação completa, entre outras abordagens. Essas soluções convencionais geram confiabilidade às GPUs. Contudo, a aplicação dessas técnicas em alguns casos podem tornar-se inviáveis, uma vez que demandam a duplicação ou triplicação integral da aplicação, resultando em considerável consumo de recursos físicos e tempo de execução.

O objetivo desta dissertação é aprimorar a resiliência dos registradores de pipeline em uma arquitetura de GPU. Para alcançar esse objetivo, foram desenvolvidas duas técnicas híbridas de tolerância a falhas, baseadas em trabalhos relacionados. Muitas abordagens anteriores se concentraram na proteção de elementos de memória, como arquivos de registradores, memória global e compartilhada, principalmente porque essa proteção pode ser implementada apenas em software. No entanto, a proteção dos registradores de pipeline exige modificações de hardware.

Na abordagem híbrida implementada, são realizadas modificações que vão desde o hardware até o compilador, proporcionando maior eficácia na mitigação de falhas que

afetam a GPU. Inicialmente, são realizadas modificações por meio de software, expandindo o Conjunto de Instruções (*Instruction Set Architecture* - ISA) e introduzindo um bit de confiabilidade que ativa a proteção de hardware por paridade ou do acumulador XOR, diretamente nos estágios do pipeline. As técnicas híbridas XOR e paridade, foram implementadas separadamente. Para a técnica de paridade, é necessário que um bit esteja disponível na descrição ISA para acomodar o bit de confiabilidade. Para a técnica XOR, é implementado um acumulador XOR que é atualizado seletivamente com o resultado das instruções duplicadas. Além disso, foi adicionado a capacidade de correção de erros com custos de implementação e tempo de execução insignificantes.

Por fim, foram testadas as implementações por meio de simulação da injeção de milhares de falhas no pipeline da GPU em até seis aplicativos de estudo de caso. Os resultados revelam que a técnica de paridade proposta pode ser mais eficaz para o *control path*, enquanto a técnica XOR pode ser mais vantajosa para o *data path*. Em ambas as técnicas, a sobrecarga de tempo de execução é compensada pelos benefícios de proteção. É importante destacar que essas técnicas de proteção podem ser aplicadas a diversas arquiteturas de computadores, incluindo a arquitetura de estudo de caso proposta e as arquiteturas modernas de GPU.

A estrutura dessa dissertação segue a seguinte sequência: no Capítulo 2, são abordadas as técnicas de tolerância a falhas empregadas, bem como a descrição da GPU utilizada, a explanação dos efeitos de radiação e a apresentação das ferramentas utilizadas nesse estudo; no Capítulo 3, é analisado o estado da arte das técnicas de tolerância a falhas em hardware e software; no Capítulo 4, é apresentada a proposta principal dessa dissertação, visando a proteção do pipeline da GPU. No Capítulo 5, são apresentadas as implementações propostas para cada técnica de tolerância a falhas; no Capítulo 6 são exibidos os resultados experimentais obtidos; por fim, no Capítulo 7 é concluído com as considerações finais decorrentes da análise realizada e propostas para trabalhos futuros.

# 2 CONCEITOS BÁSICOS

Nesse capítulo são apresentados os conceitos que fundamentam a presente dissertação, primeiro é detalhada a arquitetura da GPU FlexGripPlus, que serviu como a arquitetura base para a execução das aplicações de estudo de caso. Em seguida, são abordados os efeitos de radiação em GPUs, bem como as principais técnicas de tolerância a falhas desenvolvidas para a sua proteção, incluindo as que envolvam proteção de software e hardware, utilizadas juntas ou separadamente. Por fim, é explanado o PCHT, como a ferramenta utilizada para transformar os códigos das aplicações originais em suas respectivas versões protegidas.

## 2.1 Graphics Processing Units - GPUs

Inicialmente, as GPUs tinham o foco direcionado para o processamento gráfico. No entanto, devido ao seu alto nível de paralelismo, sua utilização expandiu-se para diversas áreas, além do âmbito da computação gráfica, elas passaram a ser usadas na computação científica, por exemplo. Nesse cenário, onde as cargas de trabalho frequentemente envolvem grandes volumes de dados, a capacidade de processamento em paralelo oferecida pelas GPUs demonstrou-se altamente eficiente (CONDIA et al., 2020a).

A GPU soft-core FlexGripPlus, utilizada nesta dissertação, se destaca por ser de código fonte aberto. Isso significa que sua arquitetura pode ser livremente obtida, analisada e personalizada conforme as necessidades dos engenheiros. Essa GPU tem como base a microarquitetura da NVIDIA G80 e foi desenvolvida em VHDL, sendo compatível com o ambiente de programação CUDA 1.0. Seu modelo é construído a partir da descrição de um módulo *Streaming Multiprocessor* (SM), o qual tem capacidade para suportar até 28 instruções (CONDIA et al., 2020a).

As GPUs modernas consistem em matrizes de SMs, que desempenham o papel de processadores Thread Múltiplo de Instrução Única (*Single-Instruction Multiple-Thread* - SIMT). Cada SM é composto por um pipeline contendo cinco estágios:

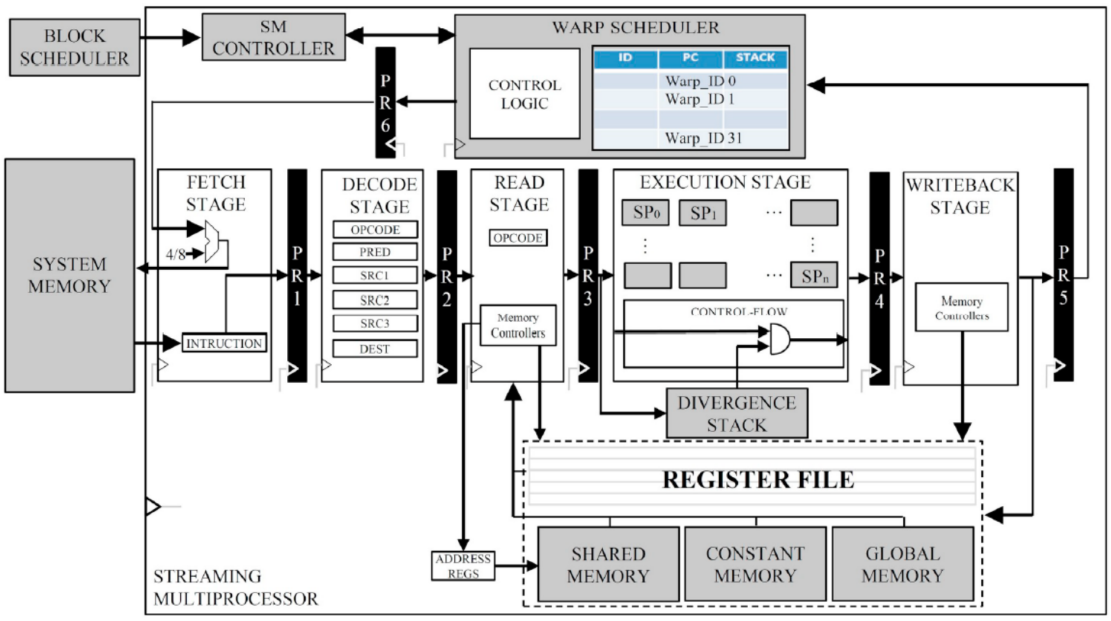
- O estágio *Fetch* é responsável por ler a instrução prevista do *buffer*.
- O estágio *Decode* decodifica o *opcode* e os especificadores de operando.
- No estágio *Read*, os endereços efetivos de cada operando fonte são calculados.
- O estágio *Execute* pode ser configurado para operar com 8, 16 ou 32 Processadores

Escalares (*Scalar Processors* - SPs), onde as operações ocorrem em grupos de 32 *threads*, denominados *warp*, que são gerenciados pelo *warp scheduler*. Os resultados, se existirem, são armazenados no local do operando de destino designado.

- Finalmente, o estágio de *Write* armazena o resultado na memória.

Baseado na microarquitetura da NVIDIA G80, a GPU contém um *warp scheduler* e uma hierarquia de *deep memory*, que inclui *General-Purpose Register Files* (GPRF), *Predicate Register Files* (PRF), *shared memories*, *global memories*, caches e outros elementos de armazenamento (GONCALVES et al., 2020). A representação da arquitetura do SM da FlexGripPlus pode ser visualizada na Figura 2.1.

Figura 2.1: Arquitetura do SM da FlexGripPlus.




Fonte: Adaptado de (CONDIA et al., 2020b).

Os *Pipeline Registers* (PRs) desempenham um papel crucial dentro do pipeline, estão localizados entre os estágios consecutivos, e têm a capacidade de armazenar informações tanto do *data path* quanto do *control path*. A função principal desses registradores é armazenar e separar informações em diferentes ciclos de *clock*. Os PRs desempenham principalmente duas tarefas fundamentais:

- Transferir os dados de processamento entre os estágios do pipeline, como a movimentação de informações dos registradores para a Unidade Lógica Aritmética (*Arithmetic Logic Unit* - ALU) de Processadores Específicos (SPs).
- Conduzir a execução do controle de instrução entre os estágios do pipeline, abran-

gendo elementos como contadores de programa, máscaras de *threads* ativas e deslocamentos de endereço de memória.

Em termos de confiabilidade, falhas que afetam os PRs do primeiro grupo, denominado *data path*, tendem a resultar em corrupção de dados. Por outro lado, falhas que impactam o segundo grupo, denominado *control path*, frequentemente conduzem a erros de tempo de execução do *kernel*.

A quantidade de PRs em uma arquitetura de GPU varia conforme o número de SMs e SPs. Cada SM possui seus registradores de controle, enquanto cada SP possui seus próprios registradores de dados. Em decorrência disso, o número de PRs no *control path* é diretamente proporcional ao número de SMs, enquanto o número de PRs no *data path* é diretamente proporcional ao número de SPs. Na configuração utilizada nessa dissertação para a FlexGripPlus, cada SM é composto por 32 SPs, o que resulta em 1.841 PRs no *control path* e 6.144 bits de PRs no *data path*. Estes valores foram estabelecidos com o objetivo de assegurar o desempenho adequado e eficiente do sistema.

Os SMs são gerenciados pelo *Block Scheduler Controller*, que distribuí os *workloads* em cada SM disponível no sistema. A nível interno, cada SM é subdividido em um pipeline composto por cinco estágios e é dotado de um *Warp Scheduler Controller* que supervisiona e monitora a execução de um conjunto de 32 threads paralelas.

Dentro do SM, um conjunto de PRs estão localizados entre dois estágios consecutivos que armazenam o *datapath* e o *controlpath*. A execução paralela de alto desempenho é alcançada por meio de uma hierarquia de memória na GPU. A memória do sistema é composta por um GPRF, por PRFs, uma memória local (*local memory* - L_mem), uma memória constante (*constant memory* - C_mem), uma memória compartilhada (*shared memory*) e uma memória global (*global memory*) (GONCALVES et al., 2022).

Alguns recursos de memória externa, como *local memory* são usados para armazenar principalmente *arrays*. A *constant memory*, por sua vez, é destinada ao armazenamento de valores constantes que são compartilhados por todas as *threads* durante a execução de um programa. Enquanto isso, a *shared memory* retém operandos de dados que podem ser compartilhados entre *threads* pertencentes ao mesmo bloco. Quanto à *global memory*, ela é utilizada para armazenar tanto os resultados finais quanto as entradas iniciais de um *kernel* de programa, com o computador *host* sendo responsável por recuperar esses valores posteriormente.

Dentro do SM, encontra-se o GPRF, o ARF e o PRF. O GPRF, um recurso de memória de 16KB, destaca-se como a memória líder e mais ágil no SM. Este GPRF é

regularmente usado para que cada *thread* armazene os operandos de dados, os endereços e os resultados durante a execução do programa. O PRF, por sua vez, é responsável pelo armazenamento de predicados resultantes de instruções lógico-aritméticas ou de comparação, permitindo até quatro registradores de predicado por *thread*.

Estes recursos (GPRF, ARF e PRF) são estaticamente organizados em bancos, alinhados conforme o número de núcleos disponíveis no SP, e só podem ser acessados pelo SP correspondente. O número de registradores por *thread* no GPRF é diretamente influenciado pela aplicação e pelo número total de *threads* ativos, alcançando até 64 registradores por *thread* em cada banco (GONCALVES et al., 2022).

A FlexGripPlus oferece a possibilidade de configuração com 8, 16 ou 32 núcleos em cada SM. Isso oferece a flexibilidade de ajustar de maneira adequada ao comprimento do *data path* nos registradores do pipeline, bem como o tamanho do registrador por núcleo em cada banco do GPRF: 2KB, 1KB e 512B, respectivamente, de acordo com a configuração específica de núcleo mencionada previamente. É relevante observar que o tamanho fixo do GPRF determina o tamanho do banco de registradores ao escolher o número de núcleos de execução por SM.

## 2.2 Efeitos da Radiação

Os componentes eletrônicos estão suscetíveis a mau funcionamento, principalmente devido a duas causas distintas: defeitos de fabricação ou exposição a efeitos de radiação. A radiação, por sua vez, consiste na emissão ou propagação de energia através do espaço, na forma de ondas eletromagnéticas ou partículas subatômicas, tais como fótons (partículas de luz), elétrons, prótons, nêutrons, entre outros. Essa energia pode ser transmitida através do vácuo, do ar ou de substâncias sólidas, dependendo do tipo de radiação envolvida. Distinguem-se, portanto, dois tipos principais de radiação: (i) radiação eletromagnética, caracterizada por ondas com variações em frequência e comprimento de onda, abrangendo luz visível, raios X, raios gama, entre outras. A capacidade dessa radiação se propagar no vácuo torna-a essencial para aplicações como comunicações via rádio e estudos astronômicos. (ii) radiação corpuscular, que inclui partículas subatômicas carregadas ou não carregadas, emitidas por fontes radioativas ou resultantes de processos nucleares, como elétrons, prótons, nêutrons, alfa e beta. Essa radiação pode induzir efeitos ionizantes e não ionizantes, dependendo de sua capacidade de remover elétrons dos átomos com os quais interage (DODD et al., 2004).

No momento que os equipamentos sofrem da exposição a radiação, na maioria das vezes resultam em falhas, que podem ser descritas como um desvio de comportamento esperado da lógica. As falhas podem ser transitórias, intermitentes ou permanentes. As falhas transitórias são caracterizadas principalmente por serem momentâneas e não permanentes, essas falhas causam um mau funcionamento temporário dos componentes. Alguns fatores influenciam para que ocorram as falhas transitórias, principalmente induzida por radiação cósmica, ruídos eletromagnéticos ou tempestades solares (DODD; MASSENGILL, 2003).

As falhas intermitentes são imprevisíveis e oscilam entre momentos de funcionamento normal e momentos de mau funcionamento. Essas falhas podem ser causadas por uma variedade de fatores, como conexões soltas ou oxidadas, componentes danificados que podem se comportar de forma inconsistente, variações de temperatura ou tensão que afetam o desempenho do sistema, entre outros. As falhas intermitentes são desafiadoras de identificar e corrigir, pois podem ser difíceis de observa-las em condições de testes controlados, podendo aparecer e desaparecer de maneira aleatória. Sendo capaz de ocasionar grandes problemas em sistemas críticos, onde a confiabilidade e disponibilidade contínua são essenciais (ESPINOSA; ANDRÉS; GIL, 2015).

As falhas permanentes são caracterizadas por resultar em uma condição de mau funcionamento contínuo e irreversível dos componentes ou sistema. Ao contrário das falhas transitórias ou intermitentes, que podem ser momentâneas ou oscilar entre momentos de funcionamento normal e falha, as falhas permanentes resultam em uma condição de não funcionamento ou de funcionamento incorreto que persiste ao longo do tempo. Diversos fatores podem ocasionar essa falha, como danos físicos aos componentes, desgaste ou envelhecimento de materiais, sobrecargas elétricas ou térmicas, entre outros. Uma vez que ocorre uma falha permanente em um componente eletrônico, ele não pode ser recuperado para o seu estado de funcionamento normal, sendo muitas vezes necessária a substituição do componente defeituoso. As falhas permanentes são preocupantes em sistemas críticos, como aplicações aeroespaciais, médicas, militares e industriais, onde a confiabilidade e disponibilidade do sistema são de extrema importância (AVIZIENIS et al., 2004).

A radiação pode afetar componentes eletrônicos de várias maneiras, dependendo do tipo de radiação, do seu nível de energia e da sensibilidade dos componentes envolvidos. A radiação ionizante, como raios gama e certas partículas, possuem energia suficiente para ionizar átomos dentro do material. Quando ocorre a ionização, elétrons são

removidos de suas órbitas, levando à criação de pares elétron-lacuna. Essas partículas carregadas podem perturbar o comportamento dos semicondutores e causar alterações nas propriedades elétricas dos componentes, levando a mau funcionamento ou danos. Alguns dos principais fenômenos comuns que a radiação pode influenciar nos componentes eletrônicos são demonstradas a seguir.

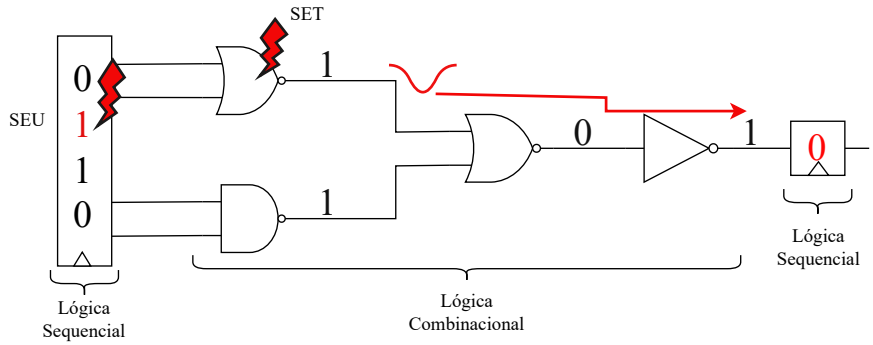
### 2.2.1 Single Event Effects - SEEs

Os Efeitos de Evento Único (*Single Event Effects* - SEEs) são fenômenos que ocorrem quando partículas carregadas de alta energia, como prótons, nêutrons, íons, interagem com componentes eletrônicos sensíveis. Essas partículas energizadas podem ser originadas de diversas fontes, como radiação cósmica, radiação solar ou partículas produzidas por reações nucleares em ambientes com altos níveis de radiação, por exemplo aplicações espaciais ou aviônica. Quando uma partícula energizada colide com um componente eletrônico, ela pode liberar energia suficiente para produzir elétrons adicionais no material semicondutor ou em sua camada de óxido, criando uma nuvem de cargas elétricas temporárias. Essa nuvem pode induzir picos de tensão, corrente ou até mesmo causar a inversão do estado lógico (chamado bit-flip) em dispositivos como memórias de armazenamento, registradores e flip-flops (DODD et al., 2004).

A ionização transitória pode ocorrer quando uma única partícula ionizante de radiação atinge o silício, criando um pulso transiente ou um SEE. Este efeito pode ser destrutivo ou não destrutivo. Um exemplo de efeito destrutivo é o Travamento de Evento Único (*Single Event Latchup* - SEL) que resulta em uma alta corrente operacional, acima das especificações do dispositivo, que deve ser corrigido por uma reinicialização de energia. Efeitos não destrutivos, também conhecidos como soft errors, são efeitos transitórios provocados pela interação de uma única partícula energizada na junção PN de um transistor off-state (DODD et al., 2004). Quando o pulso transiente ocorre em um elemento de memória, como um registrador, ele é classificado como SEU. Quando a partícula atinge um elemento combinacional, induzindo um pulso na lógica combinacional, o distúrbio é classificado como Evento Único Transitório (*Single Event Transient* - SET).

A Figura 2.2 apresenta exemplos de efeitos SEU e SET em um circuito. À esquerda, pode ser observado o efeito SEU, onde uma partícula representada por um raio incide sobre a lógica sequencial (que pode ser visualizada como um registrador), alterando o valor armazenado de "0010"para "0110". Esse efeito impacta diretamente no restante do

Figura 2.2: Efeitos SEU e SET em um circuito.




Fonte: Adaptado de (AZAMBUJA, 2013).

circuito, alterando o valor armazenado na lógica sequencial à direita de "1"para "0". No centro da Figura, podemos observar uma partícula atingindo a porta NOR e causando um pulso na lógica combinacional. Esse pulso é propagado e alcança o circuito sequencial *latch* à direita, que registra o valor incorreto "0"em vez de "1". Esse fenômeno é conhecido como SET e ocorre quando uma partícula carregada afeta temporariamente o estado de um elemento lógico, levando a alterações transitórias nos sinais do circuito (AZAMBUJA, 2013).

### 2.2.2 Dose Total de Ionização - TID

A Dose Total de Ionização (*Total Ionizing Dose* - TID) é um fenômeno ocasionado pela exposição cumulativa de componentes eletrônicos à radiação ionizante ao longo do tempo. Essa radiação abrange raios gama, raios X, prótons, nêutrons e partículas alfa, que possuem energia suficiente para ionizar átomos em materiais semicondutores e isolantes. Quando os componentes eletrônicos são submetidos a ambientes com altos níveis de radiação ionizante, como em aplicações espaciais, satélites, ambientes nucleares ou missões de longa duração no espaço, os átomos nos componentes eletrônicos podem ser ionizados (VELAZCO; FOUILLAT; REIS, 2007).

Os elétrons liberados durante a ionização podem ficar retidos nas camadas de óxido de dispositivos semicondutores, gerando uma carga acumulada que afeta suas propriedades elétricas. A exposição contínua à radiação ionizante pode resultar em uma gradual degradação do desempenho dos componentes eletrônicos. Isso se manifesta no aumento das correntes de vazamento, mudanças nas características de alimentação e na degradação da capacidade de retenção de carga em memórias, como EEPROMs e

SRAMs (VELAZCO; FOUILLAT; REIS, 2007).

Componentes sensíveis à radiação, como memórias de armazenamento e transistores em tecnologia *Metal-Oxide-Semiconductor* (MOS), são particularmente afetados por TID. O acúmulo de cargas nos óxidos e nas interfaces semicondutoras podem afetar o armazenamento de dados, causar erros de leitura e escrita e, em casos extremos, ocasionar a perda total de funcionalidade dos dispositivos (VELAZCO; FOUILLAT; REIS, 2007).

Nesta dissertação, será referido falha transitória como sendo SET e SEU que são os principais efeitos transientes causadores de falhas em GPUs.

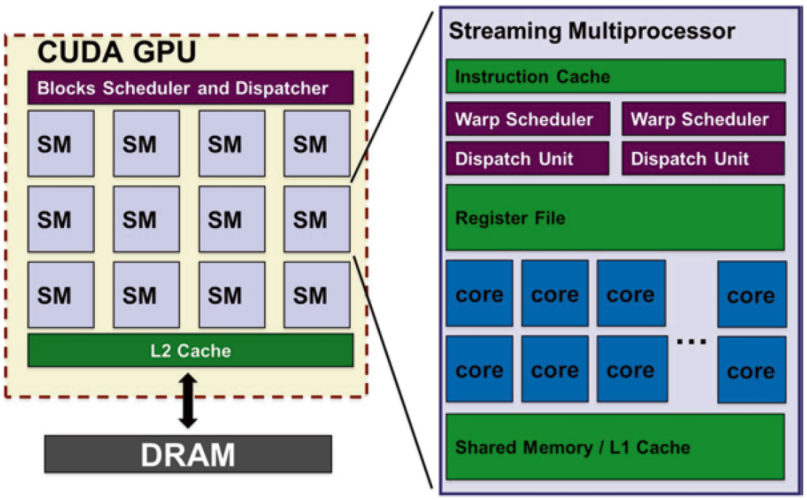
### 2.2.3 Efeitos de Radiação em GPUs

Recentemente, vem ocorrendo uma significativa discussão de pesquisa sobre a sensibilidade das GPUs à radiação (STERPONE et al., 2022; SERRANO-CASES et al., 2023). Essa discussão abrange a análise da probabilidade de ocorrência de falhas em caches e registradores, bem como do rastreamento da propagação de erros até a saída (SHI et al., 2009; HAQUE; PANDE, 2010). Além disso, concentra-se no desenvolvimento de abordagens de software e técnicas arquiteturais destinadas a proteger sistemas de GPUs (GONCALVES et al., 2022). A maioria das pesquisas relacionadas à confiabilidade das GPUs envolvem simulações de injeção de falhas (BRAGA; GONÇALVES; AZAMBUJA, 2023; MAHMOUD et al., 2018), testes de campo e experimentos de exposições à radiação (RECH et al., 2014; GONCALVES et al., 2019).

As GPUs modernas são subdivididas em múltiplas unidades de processamento conhecidas como Multiprocessadores de streaming (*Streaming Multiprocessors* - SM), sendo que cada uma delas possui a capacidade de executar diversas *threads* simultaneamente e de forma paralela, como pode ser visto na Figura 2.3. Cada unidade fundamental de processamento dentro do SM executa uma *thread* utilizando seus próprios registradores dedicados, o que evita o compartilhamento de recursos ou a necessidade de longos pipelines (LINDHOLM et al., 2008). A alocação de blocos a um SM na GPU varia de acordo com critérios como o número de registradores, a quantidade de memória compartilhada disponível no SM e os recursos exigidos por cada bloco a ser processado.

Ao avaliar a confiabilidade das GPUs em relação à radiação, é de suma importância investigar os impactos das diversas distribuições de *threads* no contexto da gestão paralela da GPU, bem como a análise da variação subsequente na seção transversal. A condução dessa avaliação permitirá identificar a configuração que resulta na menor seção

Figura 2.3: Arquitetura do SM de uma GPU.




Fonte: Adaptado de (RECH et al., 2016).

transversal e, consequentemente, na maior probabilidade de execução precisa dos cálculos.

Diversos mecanismos podem levar à incidência de partículas ionizantes nas GPUs, resultando em uma série de efeitos adversos. Essas partículas, como os nêutrons, têm o potencial de causar alterações nos bits da memória, assim como gerar pulsos transitórios nos elementos de computação lógica e circuitos de controle. As GPUs empregam caches amplos e escalonadores de tarefas complexos, utilizados para administrar o paralelismo intrínseco ao sistema. Além disso, a administração de tarefas é realizada por escalonadores dedicados e implementados internamente em hardware, ao contrário da Unidade Central de Processamento (*Central Processing Unit* - CPU), onde o escalonamento é executado por software como parte das funcionalidades do sistema operacional. A importância dos caches e dos escalonadores é particularmente considerável para processadores paralelos, já que falhas nesses elementos podem resultar em diversos erros de saída ou em disrupções funcionais (SHI et al., 2009; HAQUE; PANDE, 2010).

A exposição à radiação pode resultar em uma série de resultados: (1) ausência de impacto na saída do programa (onde o erro é ocultado ou os dados afetados não são utilizados), (2) corrupção silenciosa de dados (resultando em uma saída incorreta da aplicação), (3) interrupção do funcionamento do programa e (4) paralisação completa do sistema (necessitando reinicialização da GPU para restaurar sua operacionalidade). Dentre essas possibilidades, os cenários 2, 3 e 4 são os mais preocupantes, uma vez que permanecendo nesse estado causam defeitos. Já as situações 3 e 4 devem ser estritamente evitadas em aplicações de segurança crítica e em ambientes de alto desempenho (HPC), pois resultam

na perda de funcionalidade, penalizações no rendimento e potencial perda de dados.

Os testes de radiação em núcleos CUDA, por exemplo, são isolados de maneira a garantir que um único evento induzido por radiação afetará apenas as *threads* associadas a ele. As *threads* subsequentes a essa afetada, bem como aquelas atribuídas a núcleos CUDA vizinhos, não serão impactadas. No entanto, erros que ocorram no cache L1 ou na memória compartilhada podem potencialmente afetar várias *threads* no SM, uma vez que todas as *threads* têm acesso a esses dados. Da mesma forma, erros no cache L2, que é compartilhado entre todos os SMs, provavelmente afetará vários blocos de *threads* simultaneamente.

Uma ocorrência de radiação direcionada a um dos escalonadores (*Schedulers*) pode resultar em atribuições incorretas de tarefas, levando as *threads* a processar dados equivocados, causando problemas de sincronização que sucedem em resultados incompletos, ou gerando conflitos e erros no fluxo de controle, que podem desencadear em panes ou travamentos no kernel. A implementação de um número ampliado de *threads* paralelos normalmente reduz o tempo de execução do código, mas eleva a carga sobre o escalonador, que é responsável por gerenciar a execução e o compartilhamento de recursos. No entanto, a aplicação de uma carga adicional sobre o escalonadores (seja no nível do escalonadores de *warp* (*Schedule Warp*) ou do escalonadores de bloco (*Blocks Schedulers*)) traz a desvantagem de incrementar a probabilidade de exposição dos escalonadores à radiação (HAQUE; PANDE, 2010).

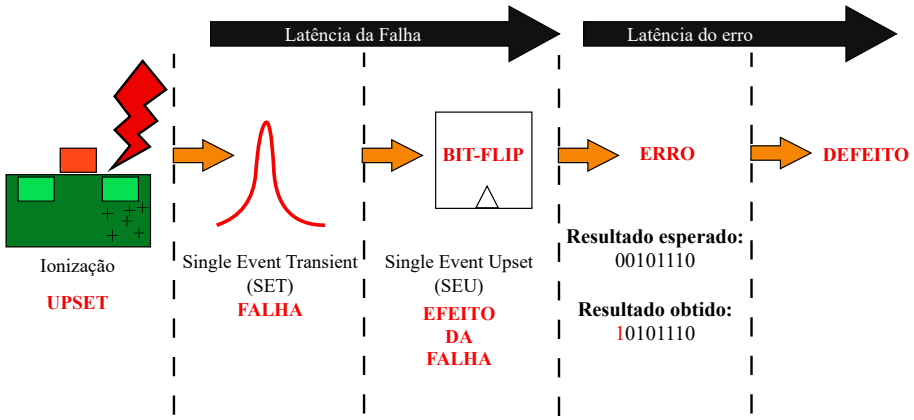
Apenas as estruturas de armazenamento cruciais das GPUs, principalmente voltada para aplicações HPC, são protegidas por meio de técnicas de Correção de Erro Único e Detecção de Erro Duplo (*Single Error Correction Double Error Detection* - SECDED) utilizando Código de Correção de Erro (*Error Correction Code* - ECC). Essas técnicas abrangem a memória do dispositivo, o cache L2, o cache de instruções, os registradores, a memória compartilhada e o cache L1. No entanto, alguns recursos permanecem desprotegidos, como a lógica interna, filas, escalonadores de blocos de *threads*, escalonadores de *warp*, unidades de despacho de instruções (*Dispatch Unit*) e a rede de interconexão. Infelizmente, as particularidades do suporte à confiabilidade para essas estruturas são considerados informações confidenciais pelos fornecedores e, portanto, não estão disponíveis para o público em geral. Vale ressaltar que, em geral, as GPUs destinadas a sistemas embarcados raramente incluem quaisquer sistemas de garantia de confiabilidade (RECH et al., 2014).

### 2.2.4 Falha, erro e defeito

O conceito de tolerância a falhas surgiu em um artigo publicado por Avizienis (1997), nele é mencionado um projeto de 1967 da NASA com o objetivo de construir espaçonaves não tripuladas para exploração espacial. Essas missões durariam cerca de 10 anos ou mais, e exigiriam computação a bordo confiável e com alto desempenho. O projeto possibilitou o estudo de todas as soluções de engenharia acessíveis e relativas à confiabilidade, dessa forma, surgindo o conceito de tolerância a falhas e, em seguida, em 1971 ocorreu o primeiro *Symposium on Fault-Tolerant Computing* organizado pelo IEEE Computer Society.

Os sistemas tolerantes a falhas têm como objetivo evitar que uma falha resulte em um erro e, consequentemente, em um defeito no sistema. Portanto, estabelece-se uma relação de causa e efeito desde a ocorrência da falha, representada por uma partícula, até a produção do resultado incorreto, caracterizando um defeito no sistema, conforme é representado na Figura 2.4. Nesta dissertação, é utilizado a definição de falha, erro e defeito de acordo com o que é apresentado em (AVIZIENIS et al., 2004).

Figura 2.4: Sequência de efeitos causados por SEU.




Fonte: Adaptado de (RECH et al., 2016).

Para a compreensão de falha, erro e defeito, inicialmente deve-se entender como se constitui um sistema. Um sistema, nesse contexto, representa uma entidade que interage com outras entidades, sejam elas sistemas similares, componentes de hardware, software, agentes humanos ou até mesmo com o mundo físico e seus fenômenos naturais. O limite do sistema é a fronteira comum entre o sistema e seu ambiente.

Os sistemas de computação são caracterizados por propriedades fundamentais: funcionalidade, desempenho, confiabilidade, segurança e custo. A funcionalidade repre-

senta a finalidade do sistema, ou seja, o que ele busca realizar, frequentemente detalhado por meio de especificações funcionais em termos de suas capacidades e desempenho esperado. O comportamento do sistema, por outro lado, descreve a maneira como o sistema opera para cumprir sua função. Isso é registrado por meio de uma série de estados e transições entre eles, que moldam o funcionamento do sistema. Por fim, o serviço oferecido por um sistema é a percepção que os usuários têm do seu comportamento. É a forma como o sistema se manifesta na interação com seus usuários (AVIZIENIS et al., 2004).

O serviço é considerado adequado quando implementa corretamente a função do sistema. Nos momentos em que o serviço entregue não está alinhado com a especificação funcional, ou quando essa especificação falha em descrever a função do sistema com precisão, são denominadas defeitos de serviço, também conhecidos como "defeitos". Esses defeitos representam uma transição de um serviço correto para um serviço incorreto, ou seja, a função do sistema não é implementada conforme o esperado. Durante o período em que o serviço está incorreto, ocorre uma interrupção do serviço. No entanto, quando o serviço incorreto é restaurado para o serviço correto, ocorre uma restauração do serviço. Os desvios do serviço correto podem apresentar-se em diferentes formas, conhecidas como "modos de defeito de serviço", e são classificados de acordo com a gravidade do defeito (AVIZIENIS et al., 2004).

Um serviço é composto por uma sequência de estados externos do sistema. Quando ocorre um defeito de serviço, significa que pelo menos um dos estados externos do sistema se desvia do estado correto de serviço. As falhas podem ser classificadas como internas ou externas a um sistema. Para que uma falha externa cause um erro e, potencialmente, defeitos, é necessário a existência prévia de uma vulnerabilidade, ou seja, uma falha interna que permita que uma falha externa afete negativamente o sistema. Em muitos casos, uma falha inicialmente resulta em um erro no estado de serviço de um componente que faz parte do estado interno do sistema, e o estado externo não é afetado imediatamente (AVIZIENIS et al., 2004).

Por essa razão, a definição de erro refere-se à parte do estado total do sistema que pode resultar em um subsequente defeito no serviço. É importante ressaltar que muitos erros não afetam o estado externo do sistema e não resultam em defeitos. Uma falha é considerada ativa quando causa um erro, caso contrário, é considerada dormente. Quando a especificação funcional de um sistema engloba várias funções, o defeito de um ou mais dos serviços que implementam essas funções pode levar o sistema a um modo degradado, ainda oferecendo um subconjunto dos serviços necessários ao usuário. Nesse

contexto, pode-se dizer que o sistema sofreu um defeito parcial em sua funcionalidade ou desempenho (AVIZIENIS et al., 2004).

Portanto, é possível observar que quando ocorre um defeito, ele sempre será percebido pelo usuário. No entanto, nos casos em que ocorre uma falha ou erro, nem sempre será perceptível. Ou seja, pode haver uma falha ou erro que não afeta o sistema a ponto de resultar em um defeito, podendo ser chamados de falhas mascaradas. A literatura oferece várias técnicas para a detecção e correção, tanto de falhas quanto de erros, algumas delas serão apresentadas na próxima seção.

## 2.3 Técnicas de Tolerância a Falhas em GPUs

Conforme apresentado nas seções anteriores, o efeito da radiação em componentes eletrônicos podem resultar em consequências catastróficas, caso não sejam tratados adequadamente. Nesse contexto, um sistema tolerante a falhas é a sua capacidade de manter o seu correto funcionamento, mesmo quando for exposto a falhas (JOHNSON, 1984). Já as técnicas de tolerância a falhas, por sua vez, concentram-se principalmente na detecção e na recuperação das falhas ou erros induzidos principalmente por radiação (LEE; ANDERSON, 1990).

As técnicas de tolerância a falhas clássicas, como a duplicação, fornecem uma das abordagens mais eficazes e completas para a detecção de erros em sistemas computacionais. Porém, elas estão entre as soluções mais caras em termos de redundância necessária, e a sua utilização mais comum é focada na aplicação de duplicação em componentes físicos de hardware. Basicamente, é feito uma cópia exata do sistema e executado simultaneamente com a original. Ao decorrer da execução, os resultados produzidos por ambos os sistemas são comparados por meio de um teste simples de igualdade. Caso for detectado alguma diferença entre o sistema original e a sua réplica, é informado que ocorreu um erro (AVIZIENS, 1976).

Os testes de limite de tempo podem ser usados como complementos para a duplicação, sendo aplicáveis tanto em nível de hardware quanto de software. Quando o limite de tempo é aplicado durante a execução do sistema, operações são utilizadas para estabelecer as restrições de tempo. Se essas restrições não forem cumpridas, uma exceção de *timeout* é gerada, indicando uma falha no componente (DUBROVA, 2013).

A redundância permite que uma falha seja mascarada ou detectada, com posterior localização, contenção e correção. No entanto, a redundância, por si só, não é suficiente

para que uma arquitetura tenha a capacidade de ser tolerante a falhas. Por exemplo, ter dois componentes duplicados conectados em paralelo não tornam um sistema tolerante a falhas, a menos que alguma forma de monitoramento seja fornecida, por exemplo com o uso da técnica de tolerância a falhas DMR, que analisa os resultados e identifica se uma falha foi detectada.

O conceito de *mascaramento de falhas* refere-se ao processo de garantir que apenas os valores corretos sejam passados para a saída do sistema, independentemente da presença de falhas. Esse processo envolve a prevenção dos impactos dos erros, correção dos erros ou aplicação de medidas compensatórias. O objetivo é garantir que o sistema continue funcionando de forma adequada, tornando a existência da falha imperceptível para o usuário. Por exemplo, uma memória protegida por um código corretor de erros é capaz de corrigir os bits defeituosos antes que o sistema utilize os dados. Outro exemplo de mascaramento de falhas é a redundância modular tripla com votação majoritária (RANDELL, 1975).

A *detecção de falhas* é o processo de identificar a ocorrência de uma falha em um sistema. Para esse fim, existem várias técnicas, como testes de aceitação e comparação. Os testes de aceitação são particularmente úteis em sistemas que não possuem componentes duplicados. Nesse método, o resultado de um programa é submetido a um teste. Se o resultado atender aos critérios do teste, o programa continua a sua execução normalmente. No entanto, se o teste de aceitação falhar, isso indica a presença de uma falha no sistema. Qualquer diferença entre esses resultados é interpretada como uma indicação de falha no sistema. Vale observar que os testes de aceitação são frequentemente aplicados em sistemas de software (RANDELL, 1975).

A *localização* de uma falha é o processo de determinar o local específico onde ocorreu uma falha. Vale ressaltar que um teste de aceitação com falha geralmente não oferece informações sobre a localização precisa da falha, ele apenas indica que algo deu errado no sistema. Por outro lado, a *contenção* de falhas é o processo de isolar uma falha e impedir que seu efeito se propague por todo o sistema. Isso é normalmente obtido por meio da detecção frequente de falhas. Uma vez que uma falha é identificada, o sistema pode se recuperar por meio da *correção* de falha, utilizando uma redundância ou um backup previamente estabelecido. Dessa forma, a propagação dos efeitos prejudiciais da falha é evitada, contribuindo para a manutenção da integridade e do desempenho do sistema.

## 2.4 Técnicas de Tolerância a Falhas em Software

A redundância de software consiste na inclusão de software adicional, além do necessário para a execução de uma função específica, com o propósito de detecção e tolerância a falhas em um sistema, conforme mencionado em (JOHNSON, 1984). A seguir, serão apresentadas algumas das técnicas de tolerância a falhas que são implementadas puramente em software. As estratégias de proteção podem ser categorizadas em quatro principais grupos: (i) técnicas aplicadas aos dados da aplicação; (ii) técnicas que protegem o fluxo de controle do programa; (iii) técnicas híbridas que abrangem proteção de dados e controle; (iv) técnicas voltadas para a proteção da memória física dos dados.

## 2.5 Técnicas de Proteção de Dados

Nessa seção são apresentadas algumas das principais técnicas de tolerância a falhas para a proteção de dados presentes na literatura.

### 2.5.1 Error Detection by Duplicated Instructions - EDDI

A técnica Detecção de Erros por Instruções Duplicadas (*Error Detection by Duplicated Instructions* - EDDI) envolve a detecção de falhas tanto temporárias quanto permanentes por meio da execução simultânea de dois programas com a mesma funcionalidade, porém utilizando conjuntos de dados distintos. Nessa abordagem, as saídas dos programas são comparadas. Isso permite identificar falhas temporárias que possam comprometer um dos programas, uma vez que as diferenças nos resultados se tornam evidentes. Dentre as falhas temporárias detectáveis estão as transitórias causando bit-flips na memória (OH; MITRA; MCCLUSKEY, 2002).

O SEU é uma das principais causas de bit-flip em memórias (CUSICK et al., 1985). Um SEU pode resultar na alteração do estado de uma célula de memória, resultando na alteração dos bits, seja de 0 para 1 ou de 1 para 0. Isso não apenas pode afetar a execução de uma instrução do programa, mas também corromper os dados armazenados na memória. A técnica EDDI pode detectar falhas capazes de serem modeladas como inversões de bits na memória. Por exemplo, quando uma instrução é acessada na memória, um bit do barramento de dados pode ser alterado devido a uma falha transitória e, então,

modificar o campo do opcode da instrução. Esse erro pode ser interpretado como uma inversão de bits no segmento de código na memória e pode ser detectado por meio do uso da técnica EDDI.

Erros transitórios que ocorrem em unidades funcionais, na lógica de controle, nos barramentos de endereços e nos barramentos de dados têm o potencial de afetar valores intermediários de cálculos, levando a resultados incorretos. Há ainda a possibilidade de um programa desviar-se de sua sequência correta de instruções, por exemplo, devido a uma falha em um desvio condicional (*branch*), o que pode ocasionar um erro no fluxo de controle. Esses tipos de falhas também podem ser detectados por meio da técnica EDDI (OH; SHIRVANI; MCCLUSKEY, 2002).

## 2.6 Técnicas de Proteção ao Controle

As técnicas de proteção voltadas para controle se concentram exclusivamente na preservação da integridade do fluxo de execução da aplicação. Para isso, são usadas ferramentas como variáveis globais e códigos de identificação de partes essenciais do programa. Isso ocorre porque replicar exatamente como o programa é executado não é viável. A proteção do controle precisa abordar três fatores fundamentais: (i) instruções de desvio (sejam condicionais ou não); (ii) desvios incorretos entre diferentes blocos básicos e; (iii) desvios inadequados internos a um único bloco básico. No entanto, a grande maioria das abordagens concentra-se apenas nos fatores (i) e (ii), o que deixa uma lacuna na cobertura completa de detecção e correção de falhas.

A proteção de dados não faz parte desse conjunto de técnicas, o que significa que falhas em registradores associados a instruções que tomam decisões condicionais e que são capazes de levar a erros de controle, podem não ser detectados. A seguir, será detalhado algumas das principais técnicas de proteção de controle.

### 2.6.1 Control Flow Checking using Assertions (CCA)

De acordo com a técnica de Verificação de Fluxo de Controle Usando Asserções (*Control Flow Checking Using Assertions* - CCA), apresentada por Kanawati et al. (1996), um programa é segmentado em um Conjunto de Intervalos Livre de Desvios (*Branch Free Interval* - BFIs), e para cada um deles, são designados dois identificadores. O Identifi-

cador do Intervalo Livre de Desvio (*Branch Free Interval Identifier* - BID) caracteriza o intervalo sem desvios e é exclusivo para cada BFIs. Enquanto o Identificador de Fluxo de Controle (*Control Flow Identifier* - CFID) representa o fluxo de controle permitido e é comum para todos os intervalos sem desvios que derivam de um intervalo anterior (pai) compartilhado.

A verificação do fluxo de controle é realizada definindo e determinando os BIDs e CFIDs. Na entrada de um intervalo sem desvios, o BID atual é atribuído à variável BID. Ao sair do intervalo livre de desvio, o BID atual é verificado. Se um intervalo livre de desvios for inserido a partir do meio, a incompatibilidade do BID atual será detectada como um erro (KANAWATI et al., 1996).

Os CFIDs são utilizados para assegurar a ordem adequada de execução dos intervalos livres de desvios, e são mantidos em uma fila de dois elementos. A fila é inicializada com o CFID do primeiro intervalo livre de desvios. Quando entra em um intervalo sem desvios, o CFID do próximo intervalo livre de desvios é colocado na fila. Ao sair de um intervalo livre de desvios, um CFID é retirado da fila e verificado se corresponde ao CFID atual. Para garantir a integridade da fila, as operações de enfileiramento e desenfileiramento incluem verificações de estado. Isso significa que o enfileiramento falhará se a fila estiver cheia, e o desenfileiramento verificará se o próximo elemento é o correto antes de ser executado (KANAWATI et al., 1996).

O CCA pode detectar todas as falhas de fluxo de controle simples e a maioria das falhas de fluxo de controle múltiplas. No entanto, essa detecção vem acompanhada de uma sobrecarga significativa. Em situações em que os BFIs são pequenos, adicionar várias linhas de código para enfileirar, desenfileirar, examinar a fila, definir uma variável e testar a variável pode ser ineficiente. Além disso, poderia aumentar a probabilidade de ocorrer um erro de fluxo de controle devido ao aumento do tamanho do programa.

Além disso, o código adicionado ao programa original, no mínimo, consome três locais de memória (um para a variável BID e dois para a fila de dois elementos), o que pode ser traduzidos em três registradores, deixando assim menos registradores para o programa original usar. Aliás, as próprias asserções contêm desvios desprotegidos. Portanto, um erro de fluxo de controle no código de asserção pode relatar uma falha quando nenhuma falha ocorreu no programa original.

### 2.6.2 Enhanced Control Flow Checking using Assertions (ECCA)

A técnica de Verificação Aprimorada de Fluxo de Controle usando Asserções (*Enhanced Control Flow Checking using Assertions* - ECCA), apresentada por Alkhalifa et al. (1999), é uma evolução da técnica CCA, que foi proposta anteriormente para detectar erros de fluxo de controle (KANAWATI et al., 1996), conforme descrito na seção anterior 2.6.1. O ECCA é introduzido para superar as limitações previamente mencionadas do CCA. Ele é implementado nos níveis alto e intermediário do sistema, por meio da adição de um manipulador de exceção, o ECCA introduz somente duas instruções por bloco (um conjunto de BFIs consecutivos), quando implementado na linguagem de alto nível. Nenhuma dessas asserções contém desvios, e apenas uma variável extra é necessária.

No entanto, a representação de baixo nível (código de máquina) incluirá dois desvios; essa abordagem identificará falhas de fluxo de controle induzidas pelo hardware ou pelo compilador. A baixa sobrecarga e a latência reduzida na detecção tornam o ECCA uma opção vantajosa para sistemas de tempo real. Além disso, sua capacidade de ser implementado em sistemas distribuídos heterogêneos acrescenta à sua portabilidade arquitetural.

### 2.6.3 Software Implemented Fault Tolerance - SWIFT

Os autores Reis et al. (2005b) implementaram o SWIFT, uma abordagem baseada em software de *thread* único para implementar redundância e tolerância a falhas. Uma das características notáveis do SWIFT é sua detecção de falhas, que é compatível com a maioria dos mecanismos de notificação e recuperação, tornando-o facilmente expansível para atingir uma tolerância total a falhas. O SWIFT é uma transformação baseada em compilador que duplica as instruções em um programa e insere instruções de comparação em pontos estratégicos durante a geração do código. Ao longo da execução, os valores são computados duas vezes e subsequentemente comparados para verificar sua equivalência, assegurando-se de que quaisquer diferenças provenientes de falhas transitórias não prejudiquem a saída do programa.

Uma abordagem de *thread* único como o SWIFT, baseada em software, oferece diversas características altamente vantajosas. Primeiramente, essa técnica não exige qualquer modificação de hardware. Em segundo lugar, o compilador é livre para utilizar os

intervalos disponíveis no agendamento de um programa, minimizando, assim, qualquer deterioração no desempenho. Terceiro, os engenheiros têm flexibilidade para adaptar a política de tratamento de falhas transitórias dentro de um programa. Por exemplo, podem optar por verificar apenas os trechos críticos do código ou ajustar a abordagem de tratamento de erros detectados para proporcionar a melhor experiência ao usuário. Quarto, um relacionamento coordenado pelo compilador entre as instruções duplicadas facilita a aplicação de métodos simples para lidar com exceções, interrupções e acesso à memória compartilhada.

A técnica SWIFT apresenta duas áreas de vulnerabilidade notáveis. Devido à introdução de redundância por meio de instruções de software, existe a possibilidade de haver um atraso no intervalo de tempo entre a validação e a utilização dos valores de registradores validados. Durante esse intervalo, eventuais perturbações podem comprometer o estado. Ainda que todas as outras instruções possuam algum grau de redundância para proteger-se contra tais perturbações, as inversões de bits no endereço do armazenamento ou nos registradores de dados passam despercebidas. Essa situação pode resultar na execução errônea do programa, devido a gravações incorretas, que ocorrem fora do escopo da detecção de falhas. Isso pode acontecer por conta de valores incorretos a serem armazenados ou de endereços de *store* incorretos.

O segundo ponto de vulnerabilidade surge quando um opcode de instrução é modificado por uma falha transitória, transformando-o em uma instrução de *store*. Nesses casos, tais instruções de *store* ficam desprotegidas, uma vez que essa mudança não foi percebida pelo compilador. Como resultado, a instrução de *store* pode ser executada livremente, permitindo que o valor que ela armazena seja transmitido para fora do escopo da detecção de falhas do sistema.

## 2.7 Técnicas de Tolerância a Falhas em Hardware

Uma das técnicas frequentemente utilizadas em sistemas tolerantes a falhas é a redundância física de hardware. Basicamente, existem três abordagens fundamentais para a redundância de hardware. Primeiramente, os métodos de replicação passiva têm a finalidade de ocultar o impacto das falhas, embora não proporcionem a capacidade de detecção, isolamento ou reparo de um componente defeituoso. Em segundo, os métodos de replicação ativa não têm a intenção de ocultar falhas, mas sim de detectar e localizar defeitos, possibilitando a substituição do componente defeituoso por uma substituto. Por fim, os

métodos híbridos combinam os aspectos vantajosos das abordagens passiva e ativa. Na replicação híbrida, o mascaramento de falhas é utilizado para mitigar o impacto de falhas no sistema, enquanto a detecção de falhas permite a substituição de um módulo defeituoso por um reserva.

### 2.7.1 Triple Modular Redundancy - TMR

Um método passivo amplamente adotado para garantir a redundância é a técnica de Redundância Modular Tripla, conhecida como TMR. Quando aplicada em um contexto passivo, a finalidade do TMR é mitigar falhas individuais através da triplicação de componentes de hardware e da realização de uma votação para determinar os resultados. A forma de votação mais comum empregada no hardware é a votação por maioria, na qual os resultados das três réplicas são comparados em tempo real e aquele que obtiver a maioria (ou que coincidir com as outras duas réplicas) é aceito como o resultado correto. Isso assegura que, se uma das réplicas apresentar um comportamento incorreto devido a uma falha, essa falha será identificada e isolada, enquanto as outras duas réplicas, que se presume estarem operando corretamente, continuarão funcionando ininterruptamente. Contudo, quando a votação é realizada por meio de software, podem ser adotadas abordagens mais sofisticadas para a votação (JOHNSON, 1984).

A *redundância ativa* divide o desafio de tolerar erros em três etapas: detecção de erros, localização de erros e recuperação de erros. A distinção central em relação à redundância passiva reside no fato de que a redundância ativa não tenta mascarar os erros. Isso implica que a saída do sistema pode ser incorreta enquanto o sistema está ocupado detectando, localizando e corrigindo o erro. Um exemplo de redundância ativa é a estratégia denominada *"standby sparing"*: nela o sistema é composto por um módulo em funcionamento e um ou mais módulos sobressalentes. Quando um erro é identificado e localizado no módulo em operação (independentemente do método de detecção e localização de falhas empregado), o módulo em operação é trocado por um dos sobressalentes. A transição entre o módulo com defeito e um dos sobressalentes intactos incorpora a fase de recuperação necessária para restaurar o funcionamento adequado (GOLOUBEVA et al., 2006).

O *"sparing"* pode ser implementado a *cold* ou *hot*. No modo *"cold standby"*, os sobressalentes permanecem ociosos e o sobressalente selecionado é ativado somente quando é preciso substituir o módulo defeituoso. Durante essa transição, o serviço pres-

tado pelo sistema sofre uma breve interrupção. Se a prioridade for minimizar o tempo de recuperação, pode-se optar pelo *"hot standby sparing"*. De acordo com esse método, os sobressalentes permanecem ativos e operam paralelamente ao módulo principal. Assim que o módulo em operação apresenta um erro, um dos sobressalentes pode prontamente assumir o seu lugar (GOLOUBEVA et al., 2006).

A *redundância híbrida* combina redundância passiva e ativa. O mascaramento de erro é usado para inibir o sistema de produzir uma saída errônea, enquanto a detecção, localização e recuperação de erro são usadas para restaurar o módulo defeituoso a um estado livre de falhas (GOLOUBEVA et al., 2006).

## 2.8 Técnicas de Tolerância a Falhas Híbridas

As abordagens híbridas, que consistem na combinação de modificações no código da aplicação (mediante suporte de software) com algum tipo de controle externo (por meio de suporte de hardware), tem ganhado destaque. A combinação entre técnicas de hardware e software é investigada devido à capacidade de gerar sistemas com maior confiabilidade ou menor sobrecarga em comparação com abordagens puramente centradas na proteção de software ou hardware. As alterações necessárias no hardware se limitam a adicionar algum dispositivo especial (muitas vezes chamado de *watchdog*), que interage com o processador e verifica possíveis erros, explorando recursos especiais (GOLOUBEVA et al., 2006).

A operação de um *watchdog* passa por um processo de duas etapas. Na primeira etapa, denominada inicialização, o *watchdog* recebe informações de referência sobre a operação sem falhas do processador que está sendo monitorado. Na segunda fase, conhecida como verificação, as informações de referência são comparadas com os dados de tempo de execução coletados pelo próprio processador *watchdog* de forma simultânea, caso seja identificado diferenças, um erro é detectado. O esquema é o de teste geral: o *watchdog* compara as informações de tempo de execução do processador (dispositivo em teste) com a de referência; o resultado da comparação é um sinal de erro (GOLOUBEVA et al., 2006).

A proteção realizada com recurso a dispositivos *watchdog* é frequentemente classificado como proteção ao nível do sistema, uma vez que funciona ao nível da aplicação, graças à interação com o software de aplicação executado pelo processador principal. Essa forma de proteção pode, naturalmente, ser integrada a outras abordagens (por exem-

plo, eventuais técnicas de proteção aplicadas internamente ao processador).

Reis et al. (2005a) introduziram o CRAFT, que aprimora a mais conhecida técnica somente de software, SWIFT, incorporando elementos de hardware leves inspirados por abordagens puramente de hardware, como TMR. Os resultados sugerem que mesmo a inclusão ou melhoria de uma única estrutura de hardware pode, em muitos cenários, elevar significativamente a confiabilidade de um sistema. Eles também ressaltaram a relevância dos mecanismos híbridos de detecção de falhas hardware/software, que se configuram como alternativas promissoras tanto para sistemas completamente baseados em hardware quanto para os baseados puramente em software.

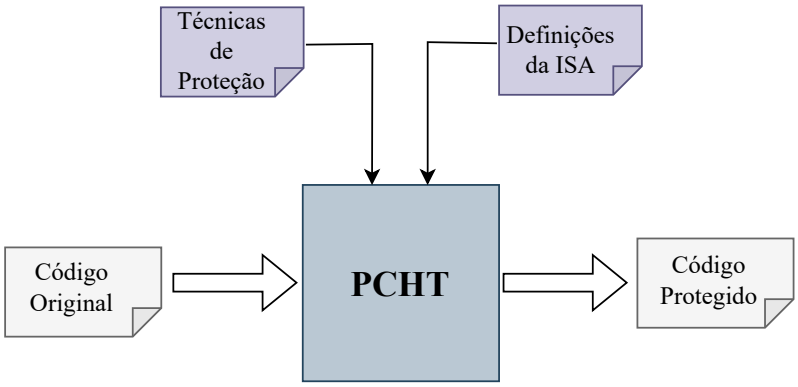
## 2.9 Post-Compiler Hardening Tool - PCHT

O aprimoramento das estratégias de tolerância a falhas frequentemente requer o uso de ferramentas complementares, uma vez que a criação de mecanismos de proteção é uma tarefa consideravelmente complexa. Cada iteração de geração de código protegido implica na análise e processamento do código, a duplicação de instruções e, ocasionalmente, a correção das instruções de desvio (*branch*). Além disso, a tarefa de gerar manualmente essas proteções e implementá-las pode ser exaustiva.

Para automatizar a transformação do código das aplicações de estudo de caso, por meio de software, empregou-se a ferramenta chamada inicialmente de *Hardening Post Compile Translator Tool* (HPCT) (AZAMBUJA et al., 2011), que posteriormente foi renomeada para *Post-Compiler Hardening Tool* (PCHT). Foram realizadas modificações de acordo com as necessidades de implementação das técnicas híbridas propostas. Essa ferramenta foi desenvolvida em Java devido à sua portabilidade em sistemas operacionais, que possuem *Java Runtime Environment* (JRE), e à facilidade na análise e manipulação de *strings*. Ela é capaz de converter automaticamente o código binário original em uma versão protegida, permitindo a inclusão de instruções suplementares e sub-rotinas de tratamento de erros no software.

A Figura 2.5 apresenta o fluxo de execução do PCHT. A ferramenta recebe como entrada o código binário do programa, o que a torna independente de compilador e linguagem. Além disso, são fornecidas as técnicas de proteção a serem aplicadas, as definições de ISA e algumas características da arquitetura do processador. O usuário tem a opção de escolher as técnicas de proteção por meio de uma Interface Gráfica do Usuário (GUI), enquanto as definições do ISA e a arquitetura do processador são descritas em classes.

Figura 2.5: Fluxo de trabalho do PCHT.




Fonte: Adaptado de (AZAMBUJA et al., 2011).

Como resultado, a ferramenta gera um código binário específico para a GPU de destino, que pode ser interpretado diretamente por essa GPU.

A partir do código original do programa, a ferramenta PCHT extrai todas as informações essenciais para realizar a transformação do código. Isso inclui informações como as localizações de memória das instruções de *branch* e os endereços de destino correspondentes, relações entre os nós do gráfico de fluxo, registradores usados e disponíveis, entre outras características. Utilizando esses dados, a ferramenta é capaz de efetuar a inserção, remoção e relocação de instruções e blocos de instruções, incluindo procedimentos.

# 3 ESTADO DA ARTE

Nesse capítulo são apresentadas as pesquisas e avanços recentes das técnicas de tolerância a falhas em GPUs, com um enfoque principal na confiabilidade do pipeline. É abordado uma série de artigos contendo as contribuições para o desenvolvimento dessa área. No primeiro artigo, é abordada a técnica SwapCodes, que são mecanismos cooperativos de software-hardwares projetados para otimizar a duplicação intra-thread. No segundo, os autores desenvolveram uma nova extensão ISA para a NVIDIA SASS 1.0, propondo três novas instruções que envolvem acesso à memória e predicados, combinados com recursos de hardware e software. Por fim, no terceiro artigo, é descrito o SInRG, uma família de técnicas que aprimoram a replicação de instruções baseadas em software.

## 3.1 Swapcodes: Error Codes for Hardware-Software Cooperative GPU Pipeline Error Detection

Sullivan et al. (2020) apresentam a técnica denominada SwapCodes, que é uma família de mecanismos cooperativos de software-hardware projetados para otimizar a duplicação intra-thread em GPUs. Essa abordagem, aproveita o hardware ECC do arquivo de registradores para detectar erros de pipeline, sem a necessidade de sacrificar a capacidade do ECC de detectar e corrigir erros de armazenamento. Ao verificar implicitamente erros de pipeline em cada leitura de registrador, os SwapCodes evitam as sobrecargas da verificação de instrução, sem a necessidade de adicionar novos verificadores de erro de hardware ou buffers.

Além disso, o artigo apresenta uma série de implementações de SwapCodes, cada uma abordando gradualmente as origens da ineficiência na duplicação intra-thread. Cada implementação possui diferentes complexidades e compromissos relacionados à detecção e correção de erros. Mais especificamente, essas implementações incluem:

- *Swap-ECC:* Uma abordagem principalmente de software, que explora a duplicação intra-thread em conjunto com o hardware de detecção de erros nos arquivos de registradores. Isso permite uma detecção eficiente de erros de pipeline em GPUs com modificações de hardware mínimas. O método inclui dois algoritmos, o SEC-DED-DP e o SEC-DP. Ambos mantêm a correção de armazenamento para arquivos de registradores protegidos pelo SEC-DED, evitando completamente erros de correção

de pipeline. O SEC-DED-DP é aplicável a qualquer código SEC-DED, enquanto o SEC-DP impõe algumas restrições de design ao código SEC-DED para reduzir os custos gerais.

- *Swap-Predict:* Esta abordagem é uma extensão do Swap-ECC que incorpora a predição seletiva de bits de verificação ECC para melhorar a eficiência de forma oportunista. Além disso, são apresentadas inovações necessárias para prever códigos residuais para a instrução MAD GPU com largura de operando mista.

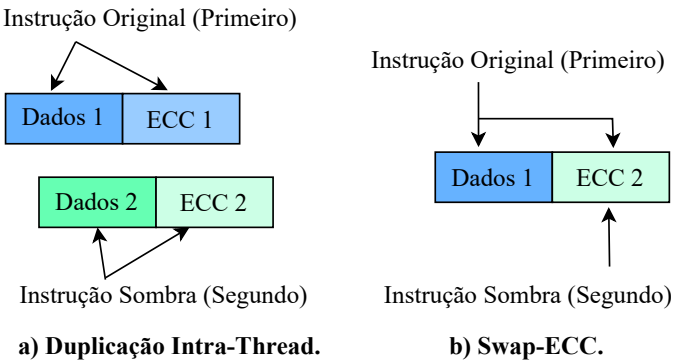
Após a realização dos experimentos utilizando os SwapCodes para proteger um processador baseado em GPU contra erros de pipeline, os resultados apresentados pelos autores Sullivan et al. (2020), demonstram que essa técnica é capaz de detectar mais de 99,3% dos erros de pipeline. Além disso, ela melhora o desempenho e a eficiência do sistema em comparação com a duplicação por software. A configuração mais eficiente dos SwapCodes apresenta apenas uma média de 15% de desaceleração em relação ao programa não duplicado.

O conceito central por trás do SwapCodes é duplicar cada instrução, emparelhando os dados da instrução original com os bits de verificação ECC de sua sombra (*shadow*). Trocar os dados e os bits de verificação da *codeword* original e sombra, garante que um único erro de pipeline não pode afetar os dados e os bits de verificação de qualquer registrador. Essa estratégia permite que o hardware ECC do arquivo de registrador detecte erros de execução em estruturas como registradores de pipeline ou lógica aritmética. Nas subseções a seguir será descrito as técnicas que os autores desenvolveram para proteção do pipeline.

### 3.1.1 Swap-ECC: SwapCodes com Duplicação Intra-Thread

SwapCodes com duplicação intra-thread emprega *codeword* trocadas dentro de cada thread para detectar armazenamento dos arquivos de registradores e erros de pipeline. A Figura 3.1 ilustra essa organização, que é chamada de Swap-ECC, contrastando-a com a duplicação de instruções imposta por software. A duplicação de instrução intra-thread executa cada instrução duas vezes, mantendo um espaço de registrador sombra para o estado duplicado. Swap-ECC também executa cada instrução duas vezes, mas sobrescreve o ECC produzido pela instrução original com o ECC da instrução de sombra.

Figura 3.1: Duplicação Intra-thread e Swap-ECC.




Fonte: Adaptado de (SULLIVAN et al., 2020).

Para que o Swap-ECC funcione, são necessárias algumas modificações tanto em software quanto em hardware. Isso requer que o compilador execute a duplicação de instrução intra-thread. O arquivo de registradores deve suportar a capacidade de escrever apenas os bits de verificação ECC de volta para uma instrução de sombra. O ISA deve ser estendido com um sinalizador de metadados de 1b para identificar instruções de sombra para a operação *write-back* mascarado. Finalmente, a propagação de movimento de ponta a ponta (descrita posteriormente) pode exigir alguns registradores e multiplexadores para propagar os bits de verificação ECC.

Figura 3.2: Exemplo da duplicação intra-thread e Swap-ECC.

(1) Un-Duplicated Code

```
ADD R1, R1, R2
```

(3) Swap-ECC

```
ADD      R3, R1, R2 //orig.
ADD.ECC  R3, R1, R2 //shad.
```

(2) Intra-Thread Duplication

```
    ADD R1, R1, R2 //orig.
    ADD R3, R3, R4 //shad.
    ISETP.EQ P1, R1, R3
@P1 BRA,U `(.L_1) //check
    BPT.TRAP 0x1
.L_1:
```

Fonte: Adaptado de (SULLIVAN et al., 2020).

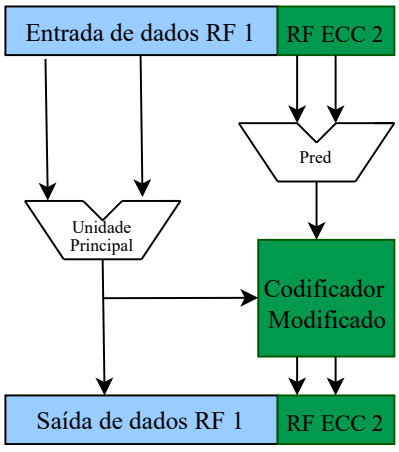
A Figura 3.2 mostra o código gerado para executar e verificar uma única instrução *add* com duplicação *intra-thread* e Swap-ECC. A duplicação *intra-thread* mantém o estado do registrador sombra, inserindo código para verificar a equivalência antes das instruções de fluxo de controle ou memória. Neste exemplo, considera-se que alguma instrução não duplicada, como um armazenamento de memória, usa o registrador de destino no código subsequente. O Swap-ECC duplica a instrução *add* (sem acúmulo de registra-

dor único), escrevendo apenas os bits de verificação ECC com a instrução sombra. Os usos seguintes do registrador de destino Swap-ECC (R3) devem aguardar a conclusão da instrução de sombra devido à dependência de gravação após a escrita. A verificação então ocorre para R3 durante qualquer arquivo de registrador lido sem código explícito.

### 3.1.2 Swap-Predict: Swap-ECC with Check-bit Prediction

Swap-ECC fornece uma organização natural para alavancar unidades de previsão ECC especializadas no *data path*, a fim de evitar a duplicação oportunista de instruções de sombra em operações comuns. A Figura 3.3 mostra essa organização, que os autores chamam de Swap-Predict. O Swap-Predict introduz um pipeline de previsão ECC de largura reduzida ao lado do *data path* convencional. O nome Swap-Predict é inspirado em técnicas de previsão de bit de verificação (como previsão de paridade) que geram os bits de verificação corretos e na sequência uma transformação lógica para verificação simultânea, sem duplicação total. A previsão de bit de verificação não é especulativa, ao contrário das estruturas de previsão de microarquitetura com nomes semelhantes.

Figura 3.3: Swap-Predict.




Fonte: Adaptado de (SULLIVAN et al., 2020).

A principal vantagem do Swap-Predict quando comparado a outras técnicas semelhantes, é que o Swap-Predict não precisa proteger todas as operações. Em vez disso, o Swap-Predict otimiza operações comuns de forma oportunista, enquanto conta com o Swap-ECC para verificar instruções difíceis de prever ou raramente utilizadas. Além disso, ao contrário das organizações anteriores de verificação simultânea, o Swap-Predict não requer implementação de novos verificadores de erros. Mediante o uso de instrumen-

tação binária, observa-se que as instruções aritméticas mais comuns incluem a adição de ponto fixo e a multiplicação-adicional de ponto fixo (*Multiplication-Addition* - MAD). A previsão de adição e multiplicação de ponto fixo pode ser realizada por meio de circuitos ECC específicos.

Após a realização dos experimentos, os autores Sullivan et al. (2020) constataram que Swap-ECC evita a necessidade de armazenamento de sombra ou verificação de instruções com duplicação *intra-thread*, reduzindo as sobrecargas médias de desempenho para 21%. O Swap-Predict adiciona unidades de previsão ECC seletivas ao Swap-ECC para evitar a necessidade de duplicar operações comuns, resultando em uma desaceleração média de 15% para a configuração de melhor desempenho avaliada.

## 3.2 Improving GPU Register File Reliability with a Comprehensive ISA Extension

Goncalves et al. (2020) apresentam uma nova extensão ISA para NVIDIA SASS 1.0, com o objetivo de aprimorar a confiabilidade das GPUs contra falhas transitórias. Foram propostas três novas instruções que envolvem acesso à memória e predicados. A implementação dessa extensão combinou recursos de hardware e software para aprimorar o conjunto de instruções do ISA.

Os resultados da pesquisa evidenciam que as alterações na arquitetura de hardware não tiveram degradação de desempenho, e menos de 1% da sobrecarga de área e consumo de energia. As despesas gerais de tempo de execução mostraram uma diminuição em cerca de 50% em comparação com técnicas de última geração baseadas em software.

Essas novas instruções propostas pelos autores, verificam a consistência dos registradores lidos, notificam o *host* em caso de incompatibilidade, e duplicam a escrita do registrador original em uma única instrução. Esse método substitui várias instruções por uma única, otimizando o tempo de execução. Ou seja, em uma única instrução, é possível duplicar o acesso ao registrador, verificar a consistência dos valores lidos e escritos, e notificar o *host*, realizando uma única busca, decodificação, execução e, o mais importante, um único acesso à memória. Nas técnicas tradicionais de tolerância a falhas com suporte de software, normalmente são necessárias duas instruções separadas no pipeline da GPU, demandando duas buscas, duas decodificações, até quatro acessos aos arquivos de registradores, duas execuções e dois acessos à memória. Além disso, para verificar a consistência, uma instrução extra por registrador de instrução utilizado deve ser totalmente executada pela GPU.

Por fim, os autores esperam que o suporte de software seja capaz de gerar código de programa, considerando essas novas instruções e inseri-las em um contexto onde as técnicas de proteção baseadas em software possam duplicar partes do código e usar efetivamente as novas instruções. O hardware deve ser capaz de executar essas novas instruções geradas pelo suporte de software e notificar o *host* em caso de detecção de falha.

Figura 3.4: Transformação do código da extensão ISA.

| | | Software-based | | | Software-based + ISA Extension | | |
|---|---|---|---|---|---|---|---|
| | Original Code | Memory | Set Predicate | All | Memory | Set Predicate | All |
| 1: | ADD R1,R1,1 | ADD R1,R1,1 | ADD R1,R1,1 | ADD R1,R1,1 | ADD R1,R1,1 | ADD R1,R1,1 | ADD R1,R1,1 |
| 2: | | ADD R1',R1',1 | ADD R1',R1',1 | ADD R1',R1',1 | ADD R1',R1',1 | ADD R1',R1',1 | ADD R1',R1',1 |
| 3: | LOAD R2,[R1] | LOAD R2,[R1] | LOAD R2,[R1] | LOAD R2,[R1] | raLOAD R2,R2',[R1,R1'] | LOAD R2,[R1] | raLOAD R2,R2',[R1,R1'] |
| 4: | | LOAD R2',[R1'] | LOAD R2',[R1'] | LOAD R2',[R1'] | | LOAD R2',[R1'] | |
| 5: | | @!PE SETP PE,R1,R1' | | @!PE SETP PE,R1,R1' | | | |
| 6: | SETP P0,R2,R3 | SETP P0,R2,R3 | SETP P0,R2,R3 | SETP P0,R2,R3 | SETP P0,R2,R3 | raSETP P0,R2,R3,offset | raSETP P0,R2,R3,offset |
| 7: | | | @!PE SETP PE,R2,R2' | @!PE SETP PE,R2,R2' | | | |
| 8: | | | @!PE SETP PE,R3,R3' | @!PE SETP PE,R3,R3' | | | |
| 9: | @P0 BRA 1 | @P0 BRA 1 | @P0 BRA 1 | @P0 BRA 1 | @P0 BRA 1 | @P0 BRA 1 | @P0 BRA 1 |
| 10: | STORE [R4],R1 | STORE [R4],R1 | STORE [R4],R1 | STORE [R4],R1 | raSTORE [R4,R4'],R1,R1' | STORE [R4],R1 | raSTORE [R4,R4'],R1,R1' |
| 11: | | @!PE SETP PE,R1,R1' | | @!PE SETP PE,R1,R1' | | | |
| 12: | | @!PE SETP PE,R4,R4' | | @!PE SETP PE,R4,R4' | | | |
| 13: | | @PE BRA ERROR | @PE BRA ERROR | @PE BRA ERROR | | | |

Fonte: Adaptado de (GONCALVES et al., 2020).

A Figura 3.4 mostra exemplos de transformações de código para cada versão protegida. O código original (em preto) consiste em instruções de *add*, *load*, *set predicate*, *conditional branch*, and *store instructions*. Para a classe da técnica baseada em software, as instruções *add* e *load* são replicadas (em verde) sobre as cópias dos registradores originais R1' e R2' para manter a consistência entre as duplicações de registradores. As instruções de armazenamento (*store*) só devem ser duplicadas em caso de replicação de memória. As instruções de predicado (*predicate*) definido são inseridas (em azul) para verificação de consistência após as instruções de acesso à memória (*memory*), após as instruções de predicado definido (*Set Predicate*) ou depois de ambos (All). Por fim, a notificação do *host* é executada por uma instrução de *branch* condicional (em amarelo). Para a extensão ISA, a replicação da instrução *add* é mantida. Os registradores de *load*, *store* e *set predicate* restantes e suas respectivas verificações de consistência são substituídos (em vermelho) pelas instruções *raLoad* e *raStore* (Memory), *raSetP* (Set Predicate) ou ambas (All). Além disso, a extensão ISA incorpora a funcionalidade de notificação ao *host*.

### 3.3 Software-Managed Instruction Replication for GPUs - SInRG

Mahmoud et al. (2018) introduzem a Replicação de Instruções Gerenciada por Software para GPUs (*Software-managed Instruction Replication for GPUs* - SInRG), que é uma família de técnicas que otimiza a duplicações de instruções baseadas em software para GPUs. O SInRG começa por implementar um algoritmo de duplicação de instruções amplamente estudado em CPUs, presente nos compiladores de produção da NVIDIA, e avalia sua aplicação em GPUs reais. Este algoritmo duplica as cadeias de fluxo de dados, mantendo as instruções não duplicadas e mantendo dois espaços de registradores, de forma que as instruções originais e duplicadas operem nos espaços de registradores original e sombra, respectivamente.

Os resultados mostram uma sobrecarga de 69% no tempo de execução. Dois motivos principais causam esse aumento no tempo de execução, (i) dobrar o número de registradores necessários por *thread* pode afetar adversamente o desempenho de algumas aplicações, uma vez que o arquivo de registradores é um recurso compartilhado, e o seu uso ineficiente pode limitar o número de *threads* simultâneas. (ii) o número total de instruções executadas aumentam com a introdução de instruções adicionais para verificação e notificação. Essas novas instruções também introduzem novas dependências, que podem limitar a capacidade de instruções de pipeline de software, aumentando ainda mais os *overheads* no tempo de execução.

Para resolver o problema acima, o SInRG adota uma abordagem em etapas. Primeiro o SInRG remove dependências diretas entre as instruções de verificação e notificação, usando uma *flag per-thread*. Segundo, a verificação e notificação adiada podem ser implementadas usando apenas uma única instrução de *high-throughput assembly* suportada por GPUs atuais. Terceiro, com o objetivo de eliminar as instruções de verificação e notificação por completo, o SInRG acelera a primeira solução mencionada por meio de suporte de hardware.

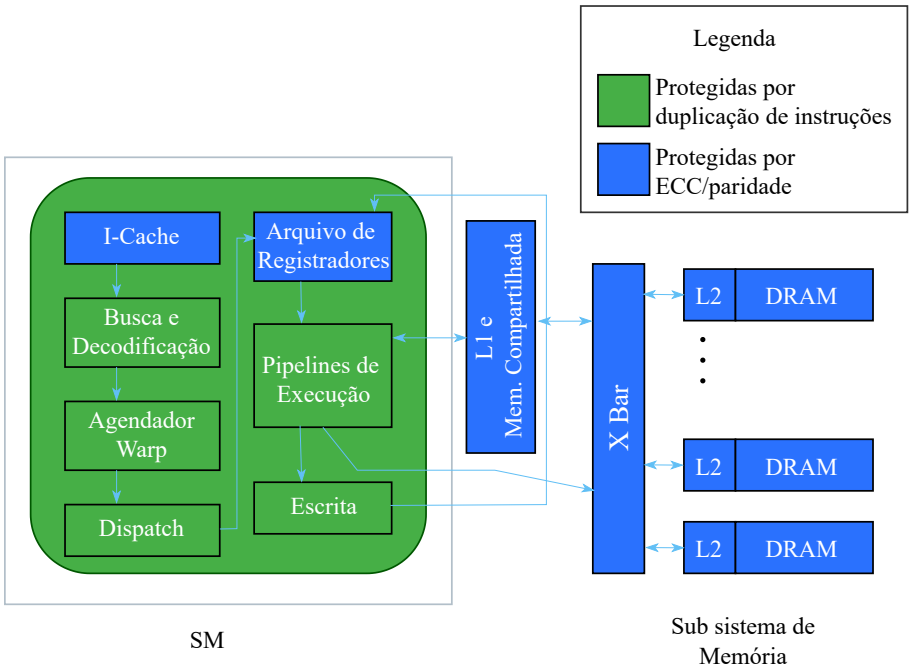
Os resultados indicam que o SInRG consegue reduzir o *overhead* do tempo de execução para 36%, representando uma melhoria de 1,94x em comparação com a implementação convencional, apenas com técnicas baseadas em software. Além disso, com o auxílio de extensões de hardware simples, que eliminam as instruções de verificação e notificação, o *overhead* é ainda mais reduzido, atingindo 30%.

Para proteger as unidades de execução e os estágios do pipeline, os autores empregaram uma abordagem de duplicação de nível de instrução do código *assembly*, sem

duplicar valores na memória. Usam o termo *sphere-of-replication* (SoR) para identificar em alto nível o que está duplicando e quais estruturas de hardware esperam proteger por meio dessa abordagem. Essa técnica permite a detecção de erros que podem afetar tanto as instruções quanto a execução das mesmas. Uma vez que nem todas as instruções são duplicadas (por exemplo, instrução de *branch* e operações atômicas permanecem não duplicadas), a cobertura é alta, porém não é completa.

Na Figura 3.5, é possível observar as estruturas de hardware que o SInRG protege. Praticamente todas as unidades SM envolvidas na execução de uma instrução, desde o estágio de busca (*fetch*) até o estágio de escrita (*write-back*), recebem proteção contra SEU. Além disso, o SInRG oferece uma camada adicional de segurança para algumas estruturas que já contam com proteção de hardware, como ECC/parity.

Figura 3.5: As estruturas de hardware da GPU no SoR.




Fonte: Adaptado de (MAHMOUD et al., 2018).

**Algoritmos de duplicação de instruções:** O SInRG duplica todas as instruções que (i) produzem valores determinísticos, (ii) não modificam diretamente o fluxo de controle e (iii) não gravam na memória.

O SInRG executa a duplicação usando dois algoritmos de base principais. O primeiro algoritmo, duplica todas as instruções em uma cadeia de fluxo de dados levando a uma instrução não duplicada, e verifica os valores apenas no final da cadeia. As instruções duplicadas operam em um espaço virtual de registrador de sombra. Para instruções que não são elegíveis para a duplicação e gravam em um registrador (por exemplo, opera-

ções atômicas e registradores especiais), o resultado da instrução original é copiado para o espaço de registrador sombra para manter a funcionalidade de execução sombra. Esse algoritmo é chamado de DRDV, porque dobra o espaço de registradores virtuais e atrasa a verificação até o final de uma cadeia de fluxo de dados.

O Segundo algoritmo de base, duplica todas as instruções elegíveis para a duplicação e as coloca um pouco antes da instrução original. A instrução duplica e lê os mesmos registradores de origem usados pela instrução original e escreve em um novo registrador virtual. É verificado imediatamente o valor no novo registrador com o valor do registrador de destino da instrução original e notificado a camada de tempo de execução para o tratamento de evento apropriado se a verificação falhar. Este esquema adiciona instruções de verificação e notificação para cada instrução elegível para duplicação, aumentando significativamente o número total de instruções dinâmicas.

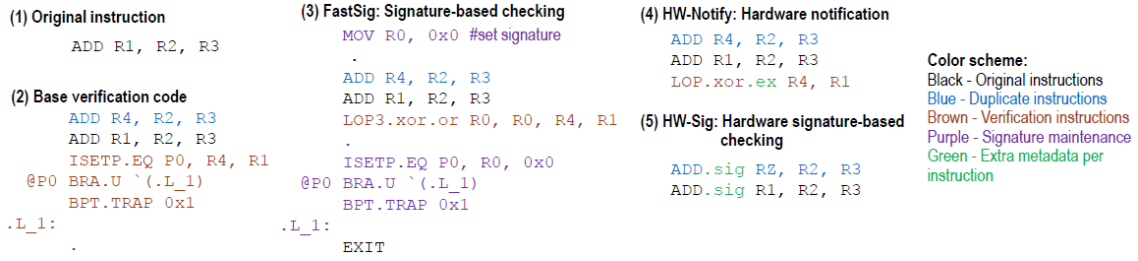
SInRG duplica cada instrução elegível para duplicação uma vez. As instruções que não são elegíveis para duplicação incluem gravações de memória, instruções de fluxo de controle, instruções que produzem valores não determinísticos, instruções de *spill/fill barrier* e instruções que gravam em registradores físicos pré-atribuídos. Instruções não determinísticas - aquelas em que a réplica e a instrução original podem produzir valores diferentes - incluem instruções S2R que leem registradores especiais cujos valores mudam ao longo do tempo (por exemplo, o valor do *clock*), operações atômicas, e leituras de memória volátil e não armazenada em cache. Uma carga pode ser não determinística se houver uma disputa de dados no programa. Os autores identificaram que marcar apenas conjuntos vulneráveis à execução como não determinísticas, identificar esse subconjunto carregado não é viável. Por isso, eles marcaram conservadoramente todas os conjuntos genéricos, globais, compartilhados, de textura e de superfície como não determinísticas. Já os conjuntos locais e contantes foram marcados como determinísticos porque não podem participar da execução dos dados.

Para **DRDV**, a verificação de entradas não duplicadas requer a adição de um conjunto de instruções de verificação e notificação, uma para cada operando de origem. Os autores implementaram uma otimização que insere apenas uma instrução de notificação e erro por instrução não duplicada, encadeando diversas instruções de verificação.

Para **SRIV**, colocaram a duplicação antes da instrução original porque a instrução original pode sobrescrever um operando de origem, e a intenção é que a duplicada gere o mesmo resultado que a instrução original utilizando dos mesmos operandos de origem. Não duplicaram as operações de *move* porque elas duplicam naturalmente o valor do

registrador de origem no registrador de destino. A verificação foi feita comparando os registradores de origem e destino da operação original. A verificação e a notificação consistem em uma operação de comparação, uma instrução condicional de *branch* e uma instrução de *trap* (BPT). A Figura 3.6 mostra um exemplo de instrução ADD usando SRIV.

Figura 3.6: Otimização do código de verificação e notificação.




Fonte: Adaptado de (MAHMOUD et al., 2018).

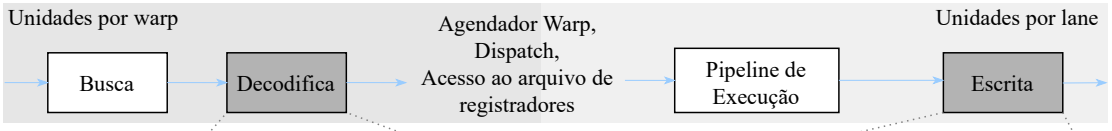
A técnica **FastSig** acumula os resultados das instruções de verificação em um registrador de assinatura e o usa no final da função para notificação de erro adiada. Este registrador de assinatura *(flag)* é inicializado com zero no início de uma função. Em cada registrador de verificação, os valores produzidos pelas instruções originais e duplicadas são adicionados e subtraídos do registrador de assinatura, respetivamente. Se o registrador de assinatura não for igual a zero no final da função, ocorreu um erro.

O uso de operações simples de adição e subtração pode perder alguns erros devido ao *over/under-flow*. Calcularam a diferença *bit-wise* entre os registradores de destino das instruções originais e duplicadas usando **XOR** e, em seguida, **OR** o resultado com o registrador de assinatura para atualizá-lo. Durante uma execução sem falhas, o registrador de assinatura permanecerá zero. A Figura 3.6 (3) mostra um exemplo de como essa otimização reduz o número de instruções de verificação estática de três para um. Uma vez que FastSig relaxa a contenção de erros, um erro pode propagar-se para a memória e, em alguns casos, resultam em *crash/hang* antes da execução da instrução de notificação. Se uma função tiver muitas instruções de retorno condicional, o número de instruções de notificação de erro também será alto, aumentando os *overheads*.

**HW-Notify**: os autores propuseram uma nova instrução *branch-free* que compara dois valores e levanta uma exceção em uma incompatibilidade para fornecer detecção de erro de baixa latência com contenção total de erro. Esta instrução substitui a operação de atualização de assinatura usada pelo *FastSig* e evita a necessidade de manter um registrador de assinatura. As alterações de hardware são resumidas na Figura 3.7, incluindo

suporte ao decodificador de instrução para a nova operação e alguma no estágio de *write-back* do pipeline para gerar uma exceção com base nos resultados de uma verificação *bit-wise*. A Figura 3.7 ilustra como a instrução é usada.

Figura 3.7: Resumo das duas técnicas de hardware.




| | | |
|---|---|---|
| HW - Notify | Decodifica o novo opcode (LOP.xor.ex Ra, Rb) | Detector zero 32b para saída da unidade de execução |
| HW - Sig | Decodifica 1 (ou 2) bits para decidir se deseja atualizar (e adicionar/subtrair) o registro de assinatura | 64b registro de assinatura por pista de hardware, 64b atualização de assinatura GF(2) baseada na aritmética XOR/complemento de um, 64b detector de zero para o registro de assinatura |

Fonte: Adaptado de (MAHMOUD et al., 2018).

**HW-Sig**: Esta técnica elimina todas as instruções de verificação e notificação. Os autores propõem um registrador de assinatura de 64b, o registrador de assinatura é iniciado em zero na inicialização do kernel, além de garantir que seja zero no final do kernel. Conforme cada instrução é executada, ela atualiza o registrador de assinatura adicionando ou subtraindo seus valores do registrador de destino com base no fato da instrução ser original ou duplicada, respectivamente. Operações que são comutativas, fáceis de projetar em hardware e requerem baixa sobrecarga de área são boas candidatas para atualizações de assinatura.

# 4 PROPOSTA

A maioria dos trabalhos presentes na literatura tem como objetivo a proteção em elementos de memória, tais como arquivos de registradores, a memória global e a memória compartilhada. Isso ocorre principalmente devido à eficácia com que o software pode endereçar esses elementos. No entanto, a proteção dos registradores de pipeline de uma GPU deve ser feito por meio de modificações de hardware (GONÇALVES; SAQUETTI; AZAMBUJA, 2018).

O objetivo desta dissertação é apresentar um controle de proteção baseado em software que oferece aos projetistas um gerenciamento abrangente sobre quais instruções devem ser protegidas. Especificamente, esta abordagem envolve uma modificação completa do sistema, que vai desde o hardware até o compilador, visando direcionar de maneira eficaz as vulnerabilidades que afetam os registradores do pipeline.

A primeira etapa consiste na expansão da *Instruction Set Architecture* (ISA) com a inclusão de um bit de confiabilidade, permitindo a ativação direta da proteção por paridade nos estágios do pipeline. A única exigência para a implementação, é que cada instrução com uma versão protegida tenha um bit disponível em sua descrição ISA para acomodar o bit de confiabilidade. No entanto, em situações em que a instrução original não protegida utiliza todos os bits, é possível restringir os campos das instruções (como o endereço do registrador, o comprimento do deslocamento, etc.) para abrir espaço para o bit de confiabilidade. Ao detectar um erro, em vez de simplesmente comunicar ao *host* sobre uma falha, a abordagem proposta realiza uma reexecução da lógica de controle para corrigir quaisquer erros ocorridos durante a execução da instrução, sem a necessidade de reexecutar a instrução por completo.

Na segunda fase, é proposto a implementação de um acumulador XOR no pipeline da GPU, que é atualizado seletivamente com base nos resultados das instruções duplicadas. O método proposto foi implementado por meio de modificações no pipeline da GPU, seguindo a mesma abordagem da primeira etapa, ao introduzir um bit de confiabilidade nas instruções da ISA. Além disso, foram acrescentadas capacidades de correção de erros a custos insignificantes em termos de implementação e tempo de execução.

Essas abordagens aprimoram a resiliência da GPU contra os efeitos da radiação, ao mesmo tempo em que minimizam os recursos consumidos durante a execução protegida. As técnicas de proteção propostas podem ser aplicadas em qualquer arquitetura de computador, desde as aplicações de estudo de caso até as arquiteturas modernas de GPU.

## 4.1 Aplicações de Estudo de Caso

Para essa dissertação, foram selecionados um conjunto de micro-benchmarks composto por até seis aplicações de estudos de caso distintas. Ainda que muitas das aplicações escolhidas possam ser consideradas simples, elas desempenham um papel fundamental na campanha de injeção de falhas e representam os blocos essenciais em grande parte das aplicações altamente paralelas de última geração, como deep learning, veículos autônomos e sistemas aviônicos. Além disso, as aplicações de estudos de caso selecionadas abrangem diferentes áreas do circuito de controle do pipeline da FlexGripPlus, permitindo uma avaliação abrangente dos benefícios em termos de confiabilidade e da degradação de desempenho proporcionados pelas técnicas propostas.

As seis aplicações utilizadas para estudo de caso, livremente adaptadas de (CONDIA et al., 2020a) são:

- *Transformação rápida de Fourier (FFT)*: É baseado no algoritmo Coley-Turkey e implementa o conhecido "elemento borboleta" (*butterfly element*) usando CUDA. Dado que a FlexGripPlus não dispõe de recursos para operações de divisão, essas foram substituídas por uma abordagem de software que se vale de operações logarítmicas e lógicas.
- *Multiplicação de Matrizes*: É baseada na rotina General Matrix Multiplication (GEMM), que é otimizada através da abordagem de "ladrilhos quadrados" (*square tiling*). Inicialmente, as matrizes de entrada são divididas em blocos. Em seguida, a multiplicação dos blocos correspondentes gera resultados parciais. Por fim, esses resultados parciais são agregados para fornecer os resultados finais da multiplicação das matrizes. Vale destacar que essa implementação é restrita a matrizes de entrada de dimensões $32 \times 32$.
- *Soma de Vetores*: É uma das aplicações originais desenvolvidas para a FlexGripPlus. Sendo projetada para calcular a soma de dois vetores.
- *Ordenação Bitônica*: É responsável por ordenar uma sequência de elementos de dados armazenados em uma matriz. Essa aplicação envolve diversas combinações de movimentos de dados entre memória e registradores, além de instruções condicionais de fluxo de controle, resultando na criação de vários fluxos de execução durante o processo.
- *Detecção de Bordas*: É uma aplicação baseada no algoritmo de Sobel para aplicar

um filtro de imagem de tamanho $3 \times 3$ a uma entrada de duas dimensões.

- *M3*: É um algoritmo de autoteste baseado em software (*Search Based Software Testing* - SBST), desenvolvido para realizar testes na memória do sistema de controle. Consiste em um conjunto de instruções de fluxo de controle que fazem uso predominante de módulos no *control path*.

## 4.2 Injeção de Falhas

A fim de avaliar as aplicações de estudo de caso mencionadas anteriormente, é essencial realizar uma injeção exaustiva com milhares de falhas. Para isso, foi utilizado um injetor de falhas desenvolvido principalmente na linguagem de programação python 3.11. Além disso, é usado o simulador ModelSim SE 10.7 para viabilizar a emulação da GPU FlexGripPlus.

A descrição da GPU FlexGripPlus em VHDL permite sua emulação no ModelSim. Essa arquitetura foi configurada com um SM de 32 SPs e sua versão compatível com as técnicas de tolerância a falhas XOR e paridade, conforme detalhado nas seções 5.2.1 e 5.2.2.

A configuração da injeção de falhas foi realizada por meio da injeção de uma única falha em cada execução da aplicação de estudo de caso. A cada execução, o injetor de falhas seleciona aleatoriamente um único bit dos 7.985 flip-flops que compõem o pipeline, e também determina um tempo de injeção variando entre 0 e o tempo total de execução da aplicação. Após essa etapa, a aplicação é executada de maneira convencional até o momento de injeção. Nesse ponto, o valor do bit escolhido é lido e invertido no flip-flop correspondente. Por fim, o restante do tempo de execução da aplicação é concluído (com uma margem extra de 10% de tempo). Em média, aproximadamente 23% das falhas são injetadas nos bits PR do *control path*, enquanto os restantes 77% são injetados aos bits PR do *data path*.

As falhas resultantes da injeção são classificadas em quatro categorias distintas:

- *Silent Data Corruption* (SDC), que acontece quando ocorre corrupção ou perda de dados na saída de uma aplicação.
- *Detected Unrecoverable Error* (DUE), que ocorre quando a aplicação não alcança seu estado final dentro do tempo de execução esperado, com uma margem adicional de 10%, ou quando ela trava completamente.

- *Masked*, quando a falha injetada não tem impacto na execução da aplicação ou nos seus resultados.
- *Detected*, onde as técnicas implementadas notificam o sistema *host* sobre a ocorrência de um erro ou falha.

# 5 IMPLEMENTAÇÃO

Nesse capítulo, são apresentadas as implementações das técnicas de tolerância a falhas. Foram desenvolvidas abordagens híbridas, combinando tanto o suporte de software quanto o suporte de hardware. Essa abordagem permite avaliar a resiliência da GPU em relação aos efeitos induzidos por radiação. Também são apresentadas as modificações realizadas no PCHT.

## 5.1 Técnicas Implementadas em Software

O suporte de software tem como foco o processo de compilação de uma aplicação para GPU. Especificamente, aborda a conversão do código do programa *Streaming ASSembler* (SASS) em um código aprimorado, e a compilação desse código SASS modificado em um microcódigo binário compatível com o hardware. Portanto, o suporte de software engloba tanto a etapa de transformação SASS (que inclui a inserção e a geração de um arquivo SASS) quanto o compilador SASS (que processa um arquivo SASS e gera um arquivo de microcódigo binário). Para isso, é feito uso do PCHT descrito na seção 2.9

Com o objetivo de proteger o código do programa SASS, a técnica proposta introduz um "*.RES*" no opcode da instrução (para diferencia-la das demais instruções), isso para todas as instruções que possuem uma versão protegida correspondente. Por exemplo, para a instrução original não protegida *MOV*, é criada a nova instrução protegida *MOV.RES*, que funciona como a instrução original, mas ativa o circuito de detecção de paridade ou o XOR do pipeline, na arquitetura da GPU. O mesmo princípio é aplicado a todas as instruções, exceto aquelas que não acessam registradores, como *NOP* (*no operation*), *BRA* (*branch*), *BAR* (*barrier synchronization*), *RET* (*kernel return*), *SSY* (*set synchronization point*) e *CAL* (*call to sub-routine*). Ao utilizar a extensão *.RES*, informamos ao compilador SASS que o bit de confiabilidade precisa ser ativado para aquela instrução.

A Tabela 5.1 apresenta um trecho de código SASS de um algoritmo de multiplicação de matrizes (Matrix Mult.), no qual é aplicado proteção a quatro classes de instruções. A primeira coluna contém o código original do aplicativo (em preto). Nas colunas subsequentes, além do código original, estão incluídas as classes de instruções duplicadas (em verde), que são: instruções de operação, instruções de acesso à memória, instruções de predicado e todas as instruções. As linhas 2, 6 e 7 não foram duplicadas, uma vez que não são implementadas versões protegidas para instruções sem acesso explícito aos regis-

tradores. É relevante observar que, embora este exemplo divida as instruções em grupos, os projetistas de software podem direcionar as instruções individualmente, balanceando assim o aumento da confiabilidade e a possível degradação de desempenho conforme suas necessidades de proteção. Além disso, essa abordagem não introduz acréscimo no tamanho do programa, é apenas substituído instruções não protegidas por suas versões protegidas.

Tabela 5.1: Exemplo da transformação das classes de instruções protegidas por software

| | Código Original | Operação | Acesso a Memória | Predicado | Todos |
|---|---|---|---|---|---|
| **1:** | ISET.C0 R2, R1, LE; | ISET.C0 R2, R1, LE; | ISET.C0 R2, R1, LE; | ISET.RES.C0 R2, R1, LE; | ISET.RES.C0 R2, R1, LE; |
| **2:** | RET C0.NE; | RET C0.NE; | RET C0.NE; | RET C0.NE; | RET C0.NE; |
| **3:** | IADD32 R3, g[6], R3; | IADD32.RES R3, g[6], R3; | IADD32 R3, g[6], R3; | IADD32 R3, g[6], R3; | IADD32.RES R3, g[6], R3; |
| **4:** | MOV R1, g[7]; | MOV R1, g[7]; | MOV R1, g[7]; | MOV R1, g[7]; | MOV R1, g[7]; |
| **5:** | GST g[R3], R8; | GST g[R3], R8; | GST.RES g[R3], R8; | GST g[R3], R8; | GST.RES g[R3], R8; |
| **6:** | BRA C0.NE, 0x90; | BRA C0.NE, 0x90; | BRA C0.NE, 0x90; | BRA C0.NE, 0x90; | BRA C0.NE, 0x118; |
| **7:** | NOP; | NOP; | NOP; | NOP; | NOP; |

No contexto do compilador SASS, a técnica proposta exige a proteção de um único bit na descrição binária de cada instrução. Idealmente, designar uma única posição de bit para todas as instruções protegidas (como o bit de confiabilidade), seria o cenário preferível. No entanto, as modificações de hardware podem gerir diferentes posições de bit para diversas classes de instrução (inclusive instruções individuais). Em situações em que a descrição binária da instrução já está completamente ocupada, sem bits sobressalentes disponíveis para servir como bit de confiabilidade, ainda é viável reduzir outros campos de descrição. Por exemplo, uma instrução com 32 bits para valores de *offset* poderia ser reduzida para 31 bits, liberando um bit para a integração do bit de confiabilidade. Mesmo que isso reduza o escopo da instrução original, o pior cenário implicaria que o compilador teria que adicionar uma instrução extra para equilibrar a perda de intervalo.

## 5.2 Técnicas Implementadas em Hardware

Nesta seção, são apresentadas as técnicas de tolerância a falhas implementadas em hardware. Foram implementadas duas abordagens: a primeira é baseada no acumulador XOR e a segunda na técnica de paridade.

### 5.2.1 Técnica XOR

O objetivo do suporte de hardware é incorporar o acumulador XOR no circuito do pipeline da arquitetura da GPU, ao mesmo tempo em que viabiliza a capacidade de notificação ao host e correção de erros.

Na linguagem VHDL, são encontradas diversas portas lógicas, incluindo a XOR, que gera uma saída com valor "0" (falso) quando suas duas entradas são iguais, ou seja, quando ambas são "1" ou ambas são "0". Em contraste, portas lógicas como AND e OR produzem uma saída "1" quando ambas as entradas são "1". No caso da porta XOR, se uma entrada for diferente da outra, a saída será "1" (verdadeiro) (HARRIS; HARRIS, 2015). A Figura 5.1 ilustra o símbolo, a função lógica implementada e a tabela verdade da porta lógica XOR.

Figura 5.1: Porta lógica XOR.


$$Y = A \oplus B$$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

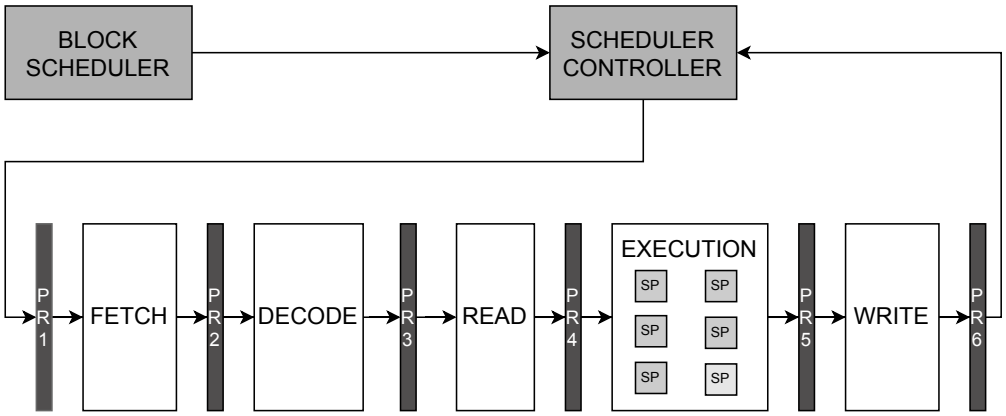
Fonte: Elaborada pelo próprio autor.

Nesse contexto, quando a primeira instrução sinalizada com a flag (bit de confiabilidade) chega ao primeiro estágio do pipeline, em vez do seu resultado ser gravado no registrador de destino, ele é armazenado em um registrador temporário. Como resultado, nenhum dado é gravado nos registradores de saída da GPU, e o fluxo de instruções é interrompido nesse ponto. À medida que a segunda instrução atinge o terceiro estágio de execução do pipeline, é realizada uma operação XOR entre seu resultado e o resultado da instrução anterior, que foi previamente armazenado no registrador temporário. Se o resultado da operação XOR não for igual a zero, um erro é notificado ao host.

A Figura 5.2 ilustra os estágios do pipeline da FlexGripPlus e o fluxo de execução das instruções. No estágio de decodificação (*Decode*), são introduzidas modificações para determinar se a instrução recém-chegada deve ser protegida, examinando se contém

o *bit/flag* ativado pelo software. Caso a instrução não necessite de proteção, ela segue o fluxo normal do processo de execução no pipeline.

Figura 5.2: Esquema geral do Pipeline na FlexGripPlus.




Fonte: Elaborada pelo próprio autor.

Quando a proteção é necessária, o estágio de *Decode* interrompe o fluxo de execução das instruções para o estágio anterior do pipeline, estágio de busca (*Fetch*), e sinaliza aos estágios subsequentes para realizar a operação XOR nessa instrução específica. Após esse processo, o estágio de *Decode* encaminha a mesma instrução (segunda instrução) para o próximo estágio de leitura (*Read*), enquanto libera o estágio anterior *Fetch* para que esteja pronto para receber uma nova instrução.

Ao chegar a instrução no estágio de escrita (*Write*), no momento em que a primeira instrução sinalizada é processada, o resultado dessa instrução que originalmente seria gravado no registrador de destino é preservada em um registrador temporário. Consequentemente, não há gravação de dados nos registradores de saída da GPU, resultando no término do encadeamento de instruções nesse ponto.

Quando a segunda instrução chega no estágio *Read*, nesse momento é realizada a operação XOR entre o seu resultado e o resultado proveniente da instrução anterior, a qual estava guardada no registrador temporário. Um erro é notificado ao *host* caso o resultado da operação XOR não resulte em zero. Cabe observar que o fluxo de controle e as instruções de sincronização não passam pelo processo de duplicação. Tais instruções são identificadas no estágio de *Decode* e, consequentemente, não sofrem a replicação.

Uma das principais vantagens consiste na simplificação do desenvolvimento de software, uma vez que somente um único bit das instruções protegidas é ativado, preservando integralmente a estrutura original do código SASS. Além disso, é esperado um processo de reexecução mais ágil, uma vez que as duas instruções não precisam percorrer todos os estágios do pipeline. Adicionalmente, a primeira instrução não requer a escrita

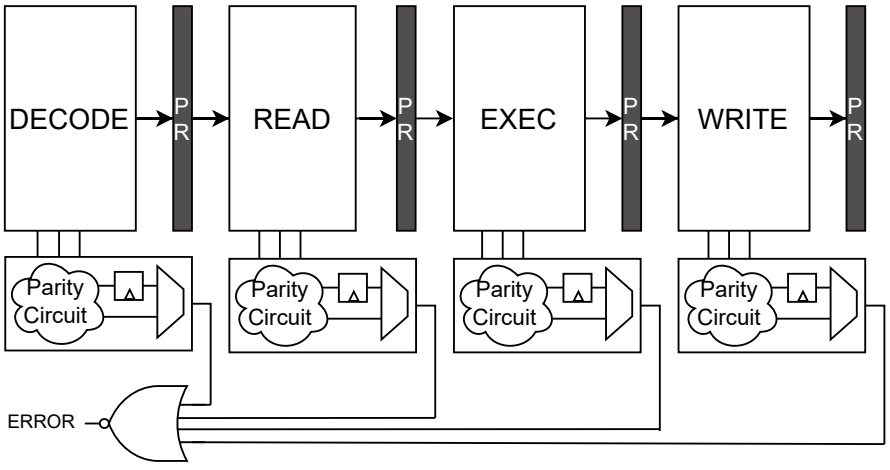
nos registradores de saída da GPU. Ou seja, todo o fluxo, englobando replicação, verificação e notificação, ocorre entre os estágios de *Decode* e *Write*, algo que não é viável por meio do suporte de software.

### 5.2.2 Técnica Paridade

O suporte de hardware tem como objetivo a implementação do circuito de verificação de paridade no pipeline da arquitetura da GPU, possibilitar a notificação ao *host*, além de corrigir os erros detectados.

Essa técnica de verificação de paridade tem sido amplamente empregada ao longo de décadas como um método eficiente para detecção de falhas em sistemas. Sua principal vantagem é na simplicidade e eficiência do circuito que permite identificar falhas únicas em matrizes com n entradas. Um exemplo de um circuito de verificação de paridade em um nível mais abstrato pode ser visualizado na Figura 5.3. Para proteger um determinado módulo utilizando uma matriz de n saídas, o circuito de paridade requer apenas uma porta XOR com n entradas. Esse circuito combinacional, juntamente com a reexecução do módulo de hardware, calcula a paridade da saída do módulo duas vezes. O resultado da primeira execução é armazenado em um único bit de memória (um *flip-flop*) e é comparado com o resultado da segunda execução para verificar sua consistência.

Figura 5.3: Esquema de controle de paridade em hardware.




Fonte: Elaborada pelo próprio autor.

As modificações na arquitetura da GPU estão divididas em três etapas: (i) melhorar o estágio de decodificação para identificar o bit de confiabilidade nas novas instruções geradas pelo lado do software e diferenciá-las das instruções originais sem proteção, (ii)

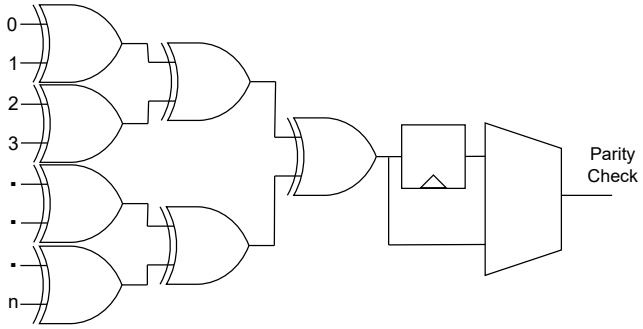
implementar o circuito de verificação de paridade em cada estágio do pipeline do FlexGripPlus e (iii) direcionar as verificações de paridade de estágios de pipeline individuais para uma única notificação de exceção.

O estágio de *Decode* no pipeline passou por modificações, baseado na posição do bit de descrição da instrução, designado para armazenar o bit de confiabilidade no compilador SASS. Para abranger todas as instruções CUDA 1.0 essenciais à execução das aplicações de estudo de caso, são divididas as instruções em dois grupos, cada um com uma posição de bit distinta.

Assim, ao avaliar o *opcode* da instrução no estágio de *Decode*, é possível acessar o bit de confiabilidade e distinguir entre as versões protegidas e não protegidas das instruções. Quando uma versão protegida é identificada, o *Decode* inicia uma sequência de controle para reexecutar a instrução original e alertar o circuito de paridade para calcular e armazenar sua paridade ou verificar sua consistência com a execução anterior. Vale destacar que uma desvantagem dessa implementação reside no fato de que a abordagem proposta não pode detectar erros de fluxo de controle nos registradores do pipeline que ocorrem antes do estágio de *Decode*.

O circuito de verificação de paridade foi implementado para todos os estágios de pipeline, a partir do estágio de *Decode* (isto é, *Decode*, *Read*, *Execute*, e *Write*). Portanto, o estágio de busca (*Fetch*), bem como os escalonadores (*Schedulers*), permaneceram inalterados. A representação da implementação pode ser observada na Figura 5.4. A notificação originada do estágio de *Decode* determina se o cálculo resultante da paridade deve ser armazenado ou verificado em relação ao valor anterior. A implementação da notificação do *host* foi realizada por meio de um OR de quatro entradas, encaminhando todas as detecções de inconsistências na verificação de paridade para um único sinal de exceção no hardware.

Figura 5.4: Circuito de verificação de paridade.




Fonte: Elaborada pelo próprio autor.

No que diz respeito à *overhead* de área, observa-se que o estágio de *decode* no pipeline necessita de algumas portas lógicas para identificar instruções protegidas e seis *flip-flops* para propagar a notificação por todo o pipeline. O circuito de paridade requer aproximadamente 1.840 portas XOR de 2 entradas para calcular a paridade de um total de 1.841 bits de dados nos registradores do pipeline, além de quatro *flip-flops* para armazenar os resultados em cada estágio de destino do pipeline. Por fim, o circuito de notificação ao *host* exige uma única porta NOR de 4 entradas.

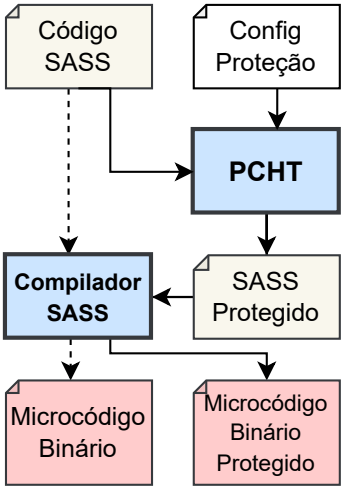
Em um panorama geral, o impacto na área é mínimo quando considera-se a totalidade da arquitetura FlexGripPlus. Esse baixo impacto é devido ao fato de que, para cada registrador de pipeline dedicado ao fluxo de controle, existem outros três destinados ao fluxo de dados, e pelo menos seis são alocados nos bancos de registradores. Além disso, a sobrecarga de área aumenta proporcionalmente ao número de SMs, mas permanece constante ao aumentar o número de SPs por SM. Dessa forma, a abordagem é escalável para GPUs comerciais.

Uma desvantagem significativa é a necessidade de reexecução do hardware, o que pode levar à degradação do desempenho. No contexto de arquiteturas de GPU, onde o controle é relativamente menor em comparação com o processamento de dados, a adoção de verificadores de paridade parece ser uma escolha de design vantajosa para maximizar a eficácia na detecção de erros de pipeline. Isso se torna particularmente relevante quando se considera que a reexecução do hardware e a potencial diminuição do desempenho podem ser restritas ao processamento do fluxo de controle.

## 5.3 Implementações no Fluxo de Execução

A Figura 5.5 ilustra o fluxo de compilação original destinado a FlexGripPlus (representado por linhas tracejadas) e as modificações que foram desenvolvidas (representadas por linhas contínuas) para implementar o suporte de software mencionado na seção 5.1. É importante observar que o fluxo de compilação inicial inclui a inserção de um código SASS no compilador, o que resulta na geração de um microcódigo binário. Em vez disso, essa abordagem envolve a inserção do código SASS na ferramenta PCHT, a qual também é fornecida com configurações de proteção, detalhando quais instruções devem ser protegidas. A partir dessas configurações, a ferramenta PCHT produz um código SASS protegido, o qual é posteriormente incorporado ao compilador SASS, responsável por gerar um microcódigo binário também protegido.

Figura 5.5: Fluxo de trabalho do PCHT.




Fonte: Adaptado de (AZAMBUJA et al., 2011).

Foi optado por utilizar a ferramenta PCHT (AZAMBUJA et al., 2011) para automatizar a transformação do código SASS, adequando-a para funcionar com os códigos SASS da FlexGripPlus. Para isso, foram realizados ajustes que permitiram a substituição das classes de instruções por suas respectivas versões protegidas, identificadas por meio da extensão "*.RES*" nos *opcodes* das instruções.

De forma mais específica, foram definidas quatro categorias de instruções para proteção: (i) Instruções de operação, como MOV (movimentação entre registradores), MVI (movimentação de imediato para destino), IMUL (multiplicação inteira), IADD (adição inteira) e LOP (operações lógicas bit a bit); (ii) instruções de acesso à memória, como GLD (carregamento da memória global) e GST (armazenamento na memória global); (iii) Instruções de predicado, que realizam comparações e estabelecem um registrador de predicado com base no resultado da comparação, como o ISET (comparação inteira); e (iv) Todas as instruções, abrangendo a replicação das classes i a iii. É importante notar que instruções que não fazem menção explícita a registradores não foram submetidas à proteção. Exemplificações destas categorias estão disponíveis na Tabela 5.1.

Modificar o compilador SASS apresenta um desafio complexo, uma vez que não existe um compilador SASS de código aberto disponível na literatura. Além disso, os fabricantes de GPUs, como NVIDIA, AMD e Intel, normalmente não compartilham seus compiladores SASS (ou similares), pois suas descrições ISA geralmente são limitadas a descrições intermediárias de código *assembly*, como o código de *parallel thread execution* (PTX). Para superar esse problema, foi desenvolvido o próprio compilador SASS que traduz o código SASS em microcódigo binário específico para o FlexGripPlus.

Dessa forma, é possível criar as ferramentas essenciais a fim de gerar a maioria das instruções CUDA 1.0 necessárias para executar as aplicações de estudo de caso mencionados na seção 4.1. Utilizando o código-fonte do compilador SASS, é realizada uma análise minuciosa de cada instrução que precisava ser protegida e identificado ou alocado um bit de confiabilidade apropriado. Com essa alocação de posição na descrição da instrução para todas as instruções a serem protegidas, é dado sequência à implementação das novas instruções, replicando as originais e ativando o bit de confiabilidade conforme necessário.

Por meio dessas modificações, é possível automatizar completamente o processo de fluxo de compilação e proteger as seis aplicações de estudo de caso, de acordo com as quatro classes de proteção de instruções: instruções de operação (Op), instruções de memória (Mem), instruções de predicado (Pred) e todas as instruções (All).

A tabela 5.2 apresenta os *overheads* de tempo de execução resultantes quando comparados com os aplicativos originais. É perceptível que a degradação média do desempenho varia de 1,02 a 1,15 vezes a aplicação original (ou seja, uma sobrecarga de tempo de execução de 1% a 14%), com um pico de 1,22× para proteger o FFT com a classe All e um mínimo de 1,00× para a proteção do M3 com a classe de Pred protegido.

Tabela 5.2: Sobrecarga de tempo de execução para técnica Paridade.

| Aplicação | Original ($\mu s$) | Proteção | | | |
|---|---|---|---|---|---|
| | | Op | Mem | Pred | All |
| FFT | 406.3 | 1.18x | 1.03x | 1.02x | 1.23x |
| Matrix Mult. | 177.3 | 1.09x | 1.03x | 1.01x | 1.14x |
| Vector Sum | 84.7 | 1.07x | 1.03x | – | 1.10x |
| Bitonic Sort | 501.5 | 1.05x | 1.03x | 1.03x | 1.11x |
| Edge Detection | 3276.1 | 1.11x | 1.02x | 1.02x | 1.15x |
| M3 | 316.7 | 1.16x | 1.15x | 1.00x | 1.19x |
| Média | 793.8 | 1.11x | 1.05x | 1.02x | 1.15x |

Esses resultados indicam que, mesmo ao proteger todas as instruções contra erros de fluxo de controle nos registradores do pipeline, o custo da degradação do desempenho permanece razoável. Além disso, adaptar a proteção para um grupo específico de instruções, ou até mesmo para instruções individuais, pode resultar em uma relação custo-benefício mais favorável. Em última análise, pode-se argumentar que esse é um investimento de baixo custo para implementar a técnica de tolerância a falhas em aplicações de software específicas.

### 5.3.1 Ambiente de injeção de falhas

Foi preciso realizar uma exaustiva campanha de injeção de falhas. Chegando a mais de 960.000 falhas. O computador usado para a simulação é um Ryzen 9 7900X com 32 GB de RAM, injetando as falhas durante alguns dias. O próximo capítulo descreve os resultados experimentais dessas injeções de falhas.

# 6 RESULTADOS EXPERIMENTAIS

Neste capítulo, serão analisados os resultados experimentais das técnicas de tolerância a falhas XOR e paridade implementadas e apresentadas nas seções anteriores 5.2.1, 5.2.2.

## 6.1 Resultados da Técnica XOR

A avaliação da implementação da técnica de tolerância a falhas híbrida XOR é conduzida por meio de uma extensiva campanha de injeção de falhas. Para essa finalidade, utiliza-se o injetor de falhas previamente descrito na seção 5.3.1, executado no simulador ModelSim. O simulador em questão está configurado com a versão original da arquitetura FlexGripPlus, equipada com um SM contendo 32 SPs, e também com a versão adaptada para a abordagem XOR, conforme descrito na Seção 5.2.1.

Como referência, as aplicações de estudo de caso detalhados na Seção 4.1 são testadas em uma variedade de configurações: versões não protegidas (Original), proteção de Predicado (Pred), proteção de Operação (OP), proteção de Memória (Mem) e proteção Total (All).

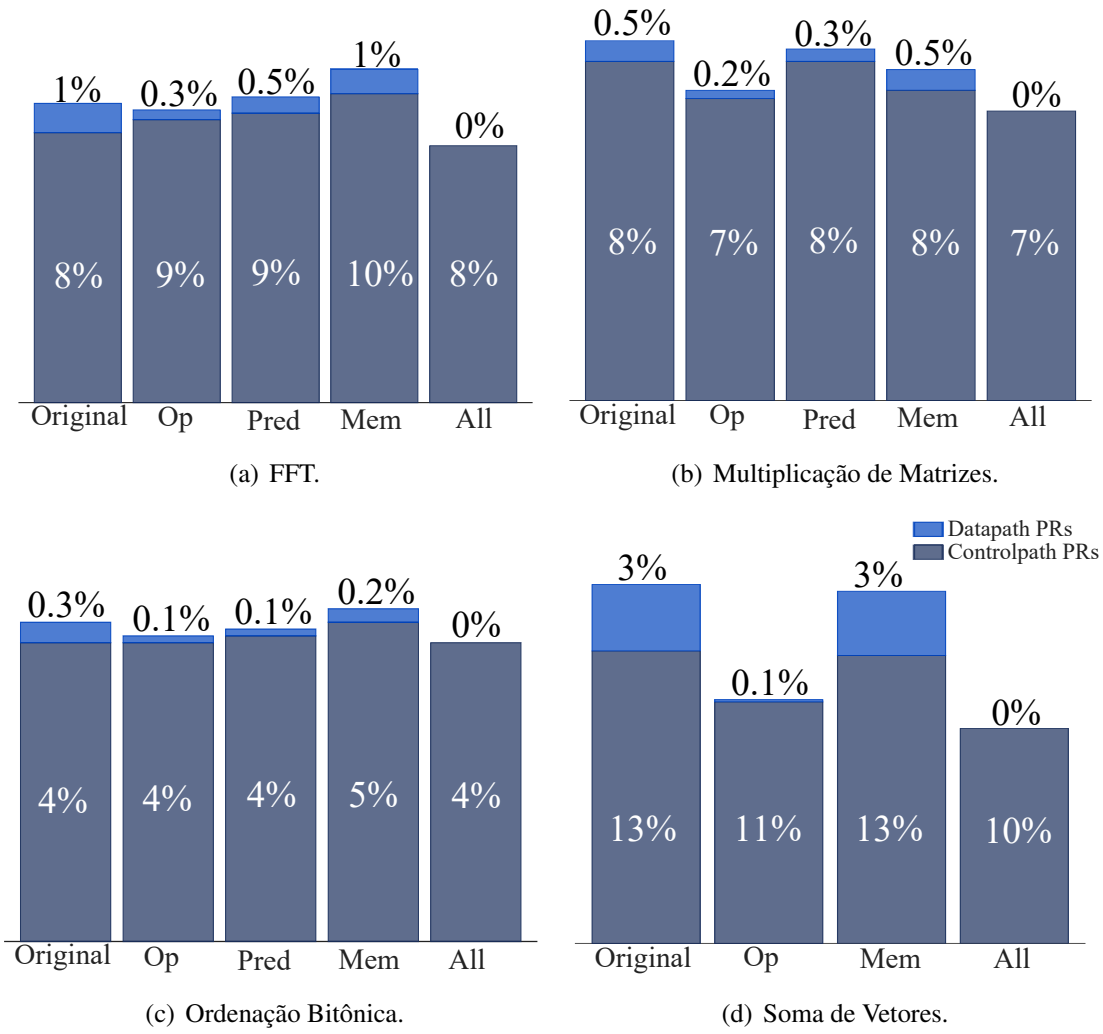
A Figura 6.1 apresenta a porcentagem de erros para todas as versões protegidas dos aplicativos de estudo de caso. Para maior clareza, os SDCs e DUEs foram agrupados como erros, e os efeitos masked e detected foram excluídos da discussão. Os resultados exibidos são coloridos de acordo com o (*Control Path* ou o *Data Path* do PR.

Dentro do pipeline, cada PR desempenha uma função à medida que a instrução passa por ele. Os PRs do *control path* têm a responsabilidade de direcionar a execução da instrução, enquanto os PRs do *data path* são encarregados de efetuar o movimento dos dados ao longo do pipeline, geralmente entre os componentes de memória. Apesar das variações nos padrões de fluxo de execução entre as aplicações, é possível identificar algumas tendências nos resultados obtidos pelo *benchmark*.

Inicialmente, é notável que a técnica híbrida XOR possui um impacto relativamente limitado na melhoria da tolerância a falhas dos registradores de pipeline em geral. O desempenho mais significativo é evidenciado no caso da aplicação Soma de vetores, no qual o grupo "All"demonstrou a capacidade de reduzir os erros de 16% para 10%. Ao ponderar seu aumento de 1,66x no tempo de execução (conforme Tabela 6.1), torna-se claro que a replicação de hardware surge como uma opção mais eficaz. Mesmo após

a avaliação de outras classes protegidas das aplicações, observa-se um resultado menos promissor para as demais aplicações, com menores reduções de erros.

Figura 6.1: Resultados da injeção de falhas.




(a) FFT.

(b) Multiplicação de Matrizes.

(c) Ordenação Bitônica.

(d) Soma de Vetores.

Fonte: Elaborada pelo próprio autor.

Esses resultados decorrem principalmente da natureza da técnica híbrida XOR, que se destaca na detecção de falhas que impactam os PRs do *data path*. Isso é evidente na versão All, que consegue reduzir os erros a 0% em todas as aplicações. No entanto, essa abordagem pode ter um efeito adverso em relação aos erros que afetam os PRs do *control path*.

A Tabela 6.2 exibe as taxas de redução de erro proporcionadas pela abordagem híbrida XOR para todas as versões protegidas das aplicações, categorizadas entre os PRs do *data path* e do *control path*. No contexto dos erros nos PRs do *data path*, a eficácia da técnica é notável. Ela assegura uma redução de erros de 100% para todas as aplicações.

Considerando a sobrecarga média de tempo de execução de 1,66x (conforme registrado na Tabela 6.1), essa abordagem se mostra como uma alternativa de proteção interessante. Esse fato é especialmente relevante quando se compreende que abordagens simples baseadas em hardware, como a Duplicação com Comparação (DWC), aumentam o uso de recursos em mais de duas vezes e podem afetar o caminho crítico da GPU, resultando em frequências operacionais reduzidas.

Tabela 6.1: Sobrecarga do Tempo de Execução da técnica XOR.

| Aplicação | Original ($\mu s$) | Op | Pred | Mem | All |
|---|---|---|---|---|---|
| FFT | 406.3 | 1.42x | 1.03x | 1.28x | 1.74x |
| Multiplicação de Matrizes | 177.3 | 1.22x | 1.02x | 1.50x | 1.74x |
| Ordenação Bitônica | 501.5 | 1.10x | 1.07x | 1.35x | 1.53x |
| Soma de Vetores | 84.7 | 1.18x | – | 1.49x | 1.66x |
| Average | 292.5 | 1.23x | 1.04x | 1.41x | 1.66x |

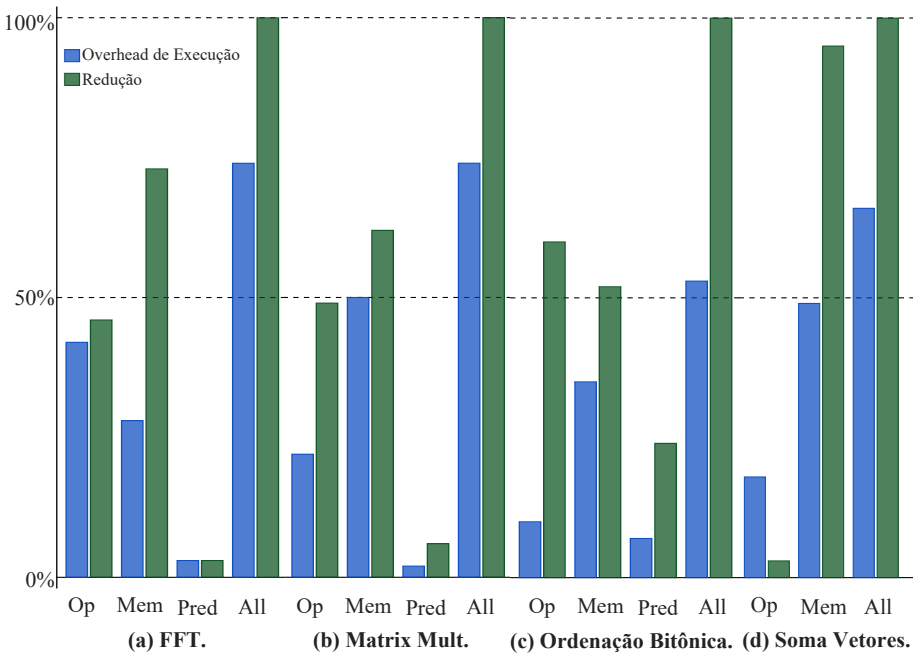
Tabela 6.2: Redução de erros dos PRs de Data Path e Control Path (%)

| Aplicação | Data Path PRs | | | | Control Path PRs | | | |
|---|---|---|---|---|---|---|---|---|
| | Op | Pred | Mem | All | Op | Pred | Mem | All |
| FFT | 46 | 3 | 73 | 100 | -8 | -14 | -4 | 5 |
| Multiplicação de Matrizes | 49 | 6 | 62 | 100 | 0 | 8 | 10 | 14 |
| Ordenaçao Bitônica | 60 | 24 | 52 | 100 | -1 | -7 | 0 | -1 |
| Soma de Vetores | 3 | – | 95 | 100 | 1 | – | 17 | 26 |
| Média | 40 | 11 | 71 | 100 | 0 | -4 | 9 | 11 |

A Figura 6.2 esquematiza a redução de erros nos PRs do *data path*. Essa estratégia resulta nos benefícios mais notáveis em termos de redução de erros. Os gráficos revelam que a categoria *All* atinge uma redução de erros de 100% em todas as aplicações. Além disso, o aumento no tempo de execução é praticamente insignificante, tendo em conta o número de instruções presentes em cada aplicação, e, portanto, permanece praticamente inalterado.

Além disso, é relevante notar que apenas algumas técnicas de tolerância a falhas são capazes de direcionar efetivamente os registradores de pipeline de uma GPU. No entanto, é possível atingir sobrecargas de tempo de execução inferiores a 1,66x ao ajustar a redução de erros por meio de diferentes versões de proteção. Por exemplo, a versão Mem apresenta uma sobrecarga de tempo de execução de 1,41x com uma redução de erros de 71%, a versão Op oferece uma sobrecarga de tempo de execução de 1,23x com uma redução de erros de 40%, e a versão Pred apresenta uma sobrecarga de tempo de execução de 1,04x com uma redução de erros de 11%.

Figura 6.2: Resultados para sobrecarga de tempo de execução e redução de erros nos PRs do *data path*.




Fonte: Elaborada pelo próprio autor.

No entanto, para os erros que afetam os PRs do *control path*, a abordagem híbrida XOR mostrou resultados distintos. A versão All, que é capaz de detectar 100% dos erros que afetam os PRs do *data path*, conseguiu apenas uma redução de 11% nos erros dos PRs do *control path*. As outras versões protegidas apresentaram resultados ainda mais desfavoráveis, com a versão Pred mostrando, em média, um aumento de 4% no número de erros. Observe que para todas as versões foi possível corrigir com sucesso todas as falhas detectadas.

Esses resultados podem ser atribuídos à complexidade do pipeline da GPU. A inclusão de circuitos adicionais para decodificação, execução e escrita das instruções originais, juntamente com o circuito de implementação da técnica XOR em hardware (englobando registradores XOR, comparação com zero e notificação ao *host*, entre outros), aumentou a suscetibilidade do *control path* da FlexGripPlus a falhas. Esse cenário levou à situação em que a tolerância a falhas introduzida pela proteção da técnica XOR acabou sendo menor do que a sensibilidade que ela própria trouxe ao pipeline por meio de sua implementação.

Quando se avalia a técnica híbrida XOR em relação aos erros nos PRs de *control path* e *data path*, observa-se resultados gerais que ainda favorecem a redução de erros. Entretanto, os benefícios de eliminar erros nos PRs do *data path* foram consideravel-

mente reduzidos devido à introdução de novos erros nos PRs do *control path*, resultantes da crescente complexidade do pipeline. Mesmo assim, é viável abordar esse desafio por meio da incorporação de técnicas adicionais de tolerância a falhas, especificamente focadas na mitigação dos erros nos PRs do *control path*, a fim de compensar as perdas de confiabilidade que decorrem desse aumento de complexidade.

## 6.2 Resultados da Técnica de Paridade

De maneira similar à abordagem XOR, foi realizada uma campanha de injeção de falhas abrangendo todas as aplicações de estudo de caso, bem como suas versões protegidas correspondentes, englobando as quatro classes de instruções protegidas: instruções de Operação (Op), instruções de Memória (Mem), instruções de Predicado (Pred) e todas as instruções (All).

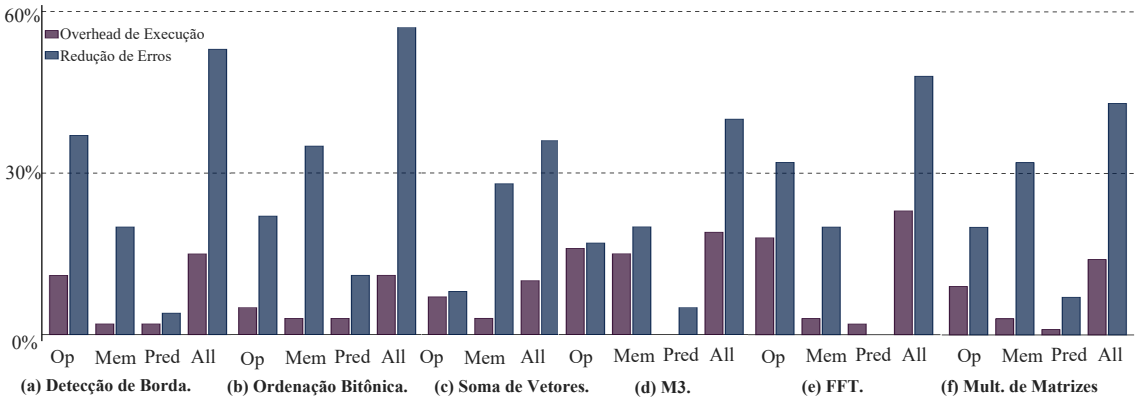
A tabela 6.3 apresenta os resultados obtidos a partir da campanha de injeção de falhas. Os dados contemplam tanto as versões originais não protegidas das aplicações quanto a porcentagem de falhas que geraram erro (SDC ou efeito Hang). Além disso, são destacadas as reduções de erros alcançadas pelas versões protegidas das aplicações, acompanhadas da porcentagem de falhas que resultaram em erro nas versões originais. Observa-se que, em média, a abordagem proposta é capaz de diminuir os erros do *control path* dos registradores do pipeline em até 47%. Devido à ausência de proteção nos estágios do pipeline anterior ao estágio de *Decode*, a obtenção de uma detecção de 100% das falhas não é possível, especialmente ao se considerar a injeção de falhas nesses registradores. No entanto, a redução apresentada é muito interessante, especialmente considerando os *overheads* de tempo de execução.

Tabela 6.3: Resultados da injeção de falhas e redução de erros

| Aplicação | Desprotegido (%) | Redução de erro protegido (%) | | | |
|---|---|---|---|---|---|
| | | Op | Mem | Pred | All |
| FFT | 8.7 | 32 | 20 | 0 | 48 |
| Multiplicação de Matrizes | 8.1 | 21 | 33 | 9 | 44 |
| Soma de Vetores | 13.3 | 8 | 27 | – | 36 |
| Ordenação Bitônica | 4.6 | 22 | 17 | 13 | 57 |
| Detecção de Borda | 2.7 | 30 | 19 | 4 | 56 |
| M3 | 6.5 | 20 | 20 | 2 | 40 |
| Média | 7.3 | 22 | 23 | 6 | 47 |

A Figura 6.3 é organizada juntamente com os dados da tabela 5.2 afim de avaliar a eficiência de cada classe de proteção. A técnica de tolerância a falhas proposta não conseguiu reduzir drasticamente os erros do pipeline, pois visa apenas uma parte dos registradores do pipeline. Porém, com exceção da aplicação FFT protegida com *Pred*, todas as versões apresentaram redução e correção de erros para todas elas, além de não causar impacto significativo no tempo de execução, mostrando que é possível, efetivamente, aumentar a confiabilidade da GPU com a abordagem proposta. É importante ressaltar que, para todas as classes de instrução, foi possível corrigir todas as falhas detectadas.

Figura 6.3: Resultados para sobrecarga de tempo de execução e redução de erros.




A versão protegida "All"demonstrou a maior redução de erros, diminuindo as falhas de pipeline em até 57%, com um aumento indireto no tempo de execução de 10% em comparação com a aplicação Ordenação Bitônica. Por outro lado, a aplicação Soma de Vetores obteve uma redução menor, atingindo 36%. Quando se leva em conta a sobrecarga de tempo de execução, observa-se que a FFT foi a aplicação mais afetada em termos de custo, com uma elevação superior ao dobro quando comparada ao Ordenação Bitônica e a Soma de vetores. Provando que o benefício dessas técnicas podem variar consideravelmente entre diferentes aplicações.

A versão protegida de Memória se destacou como uma das alternativas mais eficazes em termos de proteção, com um acréscimo médio de apenas 3% no tempo de execução para a maioria das aplicações. Exceto no caso da M3, onde proporcionou uma média de redução de erros de 23%, chegando a 32% de redução na aplicação FFT. Isso indica que essa abordagem é valiosa em termos de relação entre redução de erro e aumento no tempo de execução.

Em relação à versão protegida de Operação, os resultados variaram entre as aplicações. No contexto da FFT, ela apresentou melhorias superiores à versão de Memória. Porém, para as demais aplicações, seu desempenho foi inferior. Devido ao aumento de

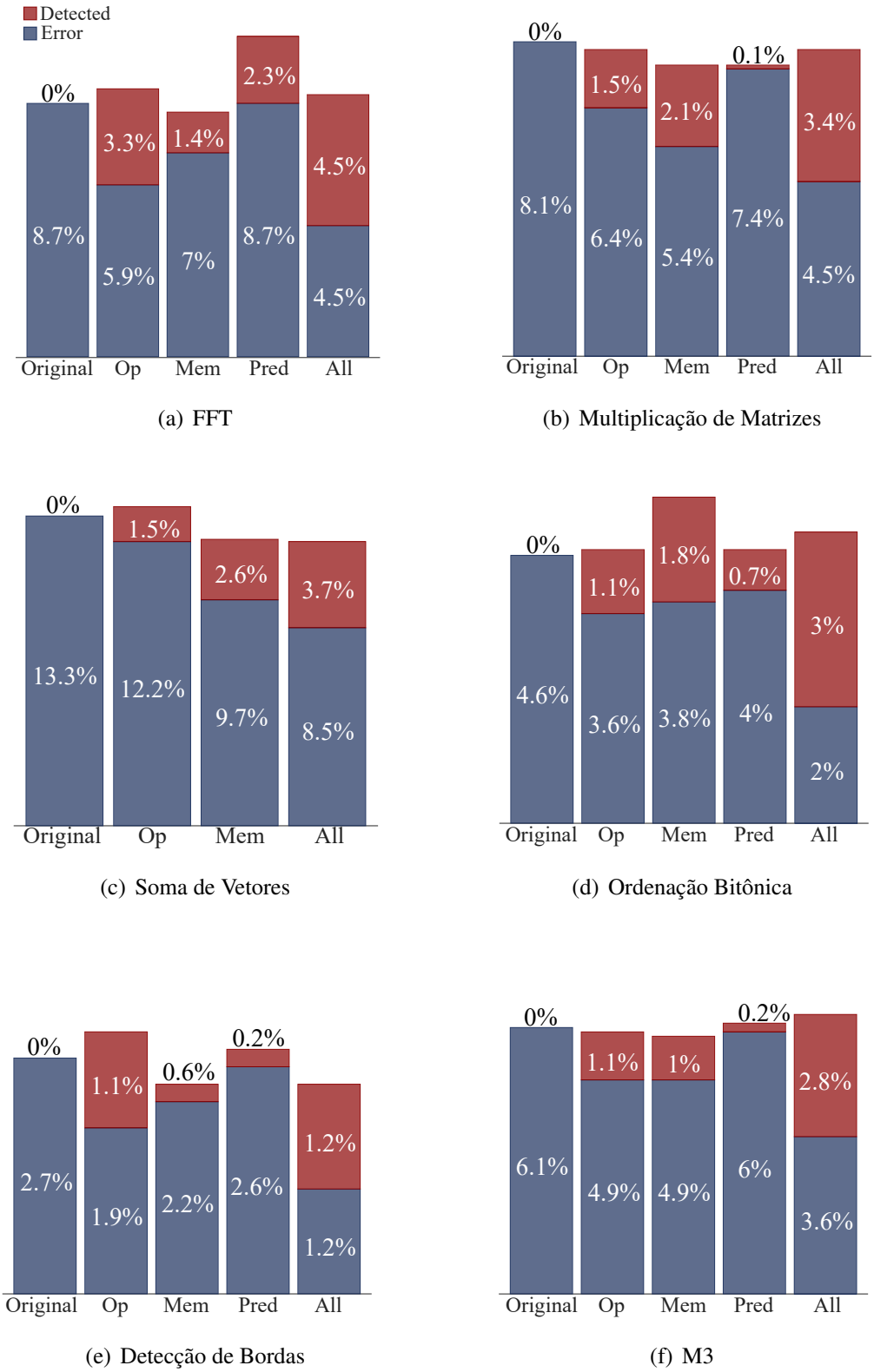
tempo de execução mais significativo em comparação com a proteção de Operação, essa abordagem demonstrou ser menos eficiente em todas as aplicações.

Por fim, a classe protegida de Predicado demonstrou as menores taxas de sobrecarga no tempo de execução, pois visa um conjunto menor de instruções protegidas. Como era de se esperar, as reduções de erro resultantes foram as mais modestas em todos os experimentos. Ainda assim, ao considerar a relação entre a redução de erros e a taxa de sobrecarga no tempo de execução, essa abordagem apresentou resultados notáveis. Ela se mostrou mais eficaz do que outras classes na aplicação de Multiplicação de Matrizes. No entanto, observa-se que foi a única classe protegida que não conseguiu melhorar a redução de erros em uma das aplicações (FFT).

A Figura 6.4 extrai informações da Tabela 6.3 e inclui a porcentagem de detecção de falhas. É importante observar que a abordagem é direcionada a falhas, não a erros, o que significa que pode identificar falhas que tem potencial de não resultar em um cálculo incorreto. Entretanto, na maioria dos casos, esse aumento é de magnitude pequena ou insignificante. Além disso, esse gráfico fornece uma visão sobre como a proteção de diferentes classes de instruções afeta a mesma aplicação. Tomando a aplicação FFT como exemplo, é possível notar que a versão protegida com a classe de Predicado não apenas manteve o número de erros, mas também resulta em um aumento na detecção de falhas.

Por outro lado, a versão protegida com a classe All consegue reduzir os erros em cerca de metade. Esses dados sugerem que, para a aplicação FFT, a opção de proteção mais eficaz possivelmente seria proteger todas as instruções de Operação e Memória, deixando as instruções de predicado sem proteção. Essa combinação de proteção (Operação + Memória) provavelmente proporcionaria melhores resultados do que proteger todas as instruções (todas as versões de proteção). Essa mesma tendência pode ser observada em menor escala na aplicação Detecção de Borda.

Figura 6.4: Resultados de injeção de falhas e capacidades de detecção.




(a) FFT

(b) Multiplicação de Matrizes

(c) Soma de Vetores

(d) Ordenação Bitônica

(e) Detecção de Bordas

(f) M3

Fonte: Elaborada pelo próprio autor.

# 7 CONCLUSÕES E CONSIDERAÇÕES FINAIS

Nessa dissertação, foram implementadas e avaliadas duas técnicas híbridas de tolerância a falhas. A primeira delas foi originalmente proposta pela NVIDIA como uma solução para mitigar os efeitos da radiação em pipelines de GPUs. A implementação forneceu uma abordagem *full-stack* completa para a técnica de tolerância a falhas híbrida XOR. Foi modificado o pipeline da GPU FlexGripPlus, incorporando novas instruções em seu conjunto de instruções CUDA 1.0. Além disso, foram proporcionados suporte ao compilador e geradas versões protegidas para quatro aplicações de estudo de caso. As versões protegidas apresentaram custo de overhead no tempo de execução que variaram de 1,04x a 1,66x. Quando comparados às técnicas de proteção DWC, esses resultados demonstram ser aceitáveis, especialmente considerando a vantagem de poder adicionar redundância de arquivo de registradores sem incorrer em degradação significativa do desempenho.

A segunda implementação propõe uma técnica de proteção de paridade de pipeline controlada por software, com o objetivo de proteger o pipeline de uma arquitetura de GPU contra os efeitos da radiação durante a execução do fluxo de controle. Foram propostos suporte de software e hardware e implementado uma técnica híbrida, incluindo fluxo de compilação e modificações arquitetônicas para permitir que os engenheiros de software direcionassem os grupos de instruções a serem protegidas. As modificações no *control path* não afetaram o fluxo original de compilação e conduziram a um acréscimo de desempenho médio que variou entre 2% e 15% nos tempos de execução, a fim de fortalecer determinados conjuntos de instruções. Além disso, as adaptações realizadas no hardware geraram custos insignificantes, especialmente quando consideramos a utilização global da área disponível na GPU FlexGripPlus no contexto do nosso estudo de caso. Além disso, foram corrigidos todos os erros nas duas técnicas propostas.

Para avaliar a eficácia da técnica de paridade, foi realizada uma extensa campanha de injeção de falhas, inserindo um total de 580.000 falhas no pipeline da GPU. Os resultados dessa análise demonstraram uma redução média nos erros, variando de 6% a 47%. Uma análise mais detalhada demonstrou que proteger todas as instruções disponíveis nem sempre se configura como a melhor escolha, mesmo em termos de diminuição de erros. Isso ressalta a importância da flexibilidade oferecida aos engenheiros de software para personalizar e direcionar a proteção a instruções específicas em um determinado aplicativo. As descobertas validam a eficácia desse enfoque customizável.

Para avaliar a eficácia e o desempenho da técnica de tolerância a falhas híbrida XOR, foi conduzida uma extensa campanha de injeção de falhas, introduzindo 380.000 falhas nos registradores de pipeline da FlexGripPlus. Os resultados revelaram uma redução de erros abaixo das expectativas em todos os aplicativos estudados. Ao aprofundar a análise das reduções de erros, ao dividir as falhas injetadas entre o *data path* e os PRs do *control path*, foi observado que a técnica de tolerância a falhas híbrida XOR demonstrou uma eficácia notável na detecção de falhas no *data path*. Contudo, essa eficácia não se repetiu da mesma maneira na detecção de falhas no *control path*, chegando ao ponto de tornar a GPU mais sensível a algumas versões protegidas de determinadas aplicações.

Em trabalhos futuros, a intenção é expandir o escopo do estudo, incluindo um maior número de aplicações de estudo de caso e ampliando significativamente a campanha de injeção de falhas. Por último, planeja-se projetar e integrar abordagens de tolerância a falhas especificamente desenvolvidas para abranger os erros PR do *control path*. Essa iniciativa visa criar uma técnica híbrida de tolerância a falhas que possa identificar e até mesmo corrigir todas as falhas presentes no pipeline.

# PUBLICAÇÕES

## Artigos em Periódicos

1. **BRAGA, G.**; GONÇALVES, M. M.; AZAMBUJA, J. R. F. Software-controlled pipeline parity in GPU architectures for error detection. **MICROELECTRONICS RELIABILITY**, v. 148, p. 115155, 2023.
2. **BRAGA, G.**; BENEVENUTI, F.; GONCALVES, M. M.; MUNOZ, H.; HUBNER, M.; BRANDALERO, M.; KASTENSMIDT, F. L.; AZAMBUJA, J. R. Evaluating Softcore GPU in SRAM-Based FPGA under Radiation-Induced Effects. **MICROELECTRONICS RELIABILITY**, v. 1, p. 113768, 2021.

## Artigos em Anais de Conferências

1. **BRAGA, G. A.**; GONÇALVES, M. M.; AZAMBUJA, J. R. Evaluating an XOR-based Hybrid Fault Tolerance Technique to Detect Faults in GPU Pipelines. In: *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2023, Foz do Iguacu.
2. **BRAGA, G. A.**; GOBATTO, L.; GONÇALVES, M. M.; AZAMBUJA, J. R. Improving GPU Reliability with Software-Managed Pipeline Parity for Error Detection and Correction. In: *IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, 2024, Punta del Este. (**submetido aguardando revisões**)
3. **BRAGA, G. A.**; GOBATTO, L.; GONÇALVES, M. M.; AZAMBUJA, J. R. An Investigation into Fault Detection and Correction in GPU Pipelines with a Hybrid XOR Approach. In: *IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, 2024, Punta del Este. (**submetido aguardando revisões**)

# REFERÊNCIAS

ALCAIDE, S. et al. Software-only diverse redundancy on gpus for autonomous driving platforms. In: IEEE. **2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)**. [S.l.], 2019. p. 90–96.

ALKHALIFA, Z. et al. Design and evaluation of system-level checks for on-line control flow error detection. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 10, n. 6, p. 627–641, 1999.

ASANO, S.; MARUYAMA, T.; YAMAGUCHI, Y. Performance comparison of fpga, gpu and cpu in image processing. In: IEEE. **2009 international conference on field programmable logic and applications**. [S.l.], 2009. p. 126–131.

AVIZIENIS, A. Toward systematic design of fault-tolerant systems. **Computer**, IEEE, v. 30, n. 4, p. 51–58, 1997.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, v. 1, n. 1, p. 11–33, 2004.

AVIZIENS, A. Fault-tolerant systems. **IEEE transactions on computers**, IEEE, v. 100, n. 12, p. 1304–1312, 1976.

AYLLON, M. A. et al. Detection of overall fruit maturity of local fruits using convolutional neural networks through image processing. In: **Proceedings of the 2nd International Conference on Computing and Big Data**. New York, NY, USA: Association for Computing Machinery, 2019. (ICCBD 2019), p. 145–148. ISBN 9781450372909.

AZAMBUJA, J. R. et al. Detecting sees in microprocessors through a non-intrusive hybrid technique. **IEEE Transactions on Nuclear Science**, IEEE, v. 58, n. 3, p. 993–1000, 2011.

AZAMBUJA, J. R. F. d. Designing and evaluating hybrid techniques to detect transient faults in processors embedded in fpgas. 2013.

BRAGA, G.; GONÇALVES, M. M.; AZAMBUJA, J. R. Software-controlled pipeline parity in gpu architectures for error detection. **Microelectronics Reliability**, Elsevier, v. 148, p. 115155, 2023.

CONDIA, J. E. R. et al. Flexgripplus: An improved gpgpu model to support reliability analysis. **Microelectronics Reliability**, Elsevier, v. 109, p. 113660, 2020.

CONDIA, J. E. R. et al. Flexgripplus: An improved gpgpu model to support reliability analysis. **Microelectronics Reliability**, Elsevier, v. 109, p. 113660, 2020.

CUSICK, J. et al. Seu vulnerability of the zilog z-80 and nsc-800 microprocessors. **IEEE Transactions on Nuclear Science**, IEEE, v. 32, n. 6, p. 4206–4211, 1985.

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

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

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.

DUBROVA, E. **Fault-tolerant design**. [S.l.]: Springer, 2013.

ESPINOSA, J.; ANDRÉS, D. de; GIL, P. Increasing the dependability of vlsi systems through early detection of fugacious faults. In: IEEE. **2015 11th European Dependable Computing Conference (EDCC)**. [S.l.], 2015. p. 190–197.

GOLOUBEVA, O. et al. **Software-implemented hardware fault tolerance**. [S.l.]: Springer Science & Business Media, 2006.

GONCALVES, M. et al. Improving gpu register file reliability with a comprehensive isa extension. **Microelectronics Reliability**, Elsevier, v. 114, p. 113768, 2020.

GONCALVES, M. et al. Selective fault tolerance for register files of graphics processing units. **IEEE Transactions on Nuclear Science**, IEEE, v. 66, n. 7, p. 1449–1456, 2019.

GONÇALVES, M.; SAQUETTI, M.; AZAMBUJA, J. R. Evaluating the reliability of a gpu pipeline to seu and the impacts of software-based and hardware-based fault tolerance techniques. **Microelectronics Reliability**, Elsevier, v. 88, p. 931–935, 2018.

GONCALVES, M. M. et al. Evaluating low-level software-based hardening techniques for configurable gpu architectures. **The Journal of Supercomputing**, Springer, v. 78, n. 6, p. 8081–8105, 2022.

HAQUE, I. S.; PANDE, V. S. Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu. In: IEEE. **2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing**. [S.l.], 2010. p. 691–696.

HARRIS, S. L.; HARRIS, D. **Digital design and computer architecture**. [S.l.]: Morgan Kaufmann, 2015.

JOHNSON, B. Fault-tolerant microprocessor-based systems. **IEEE Micro**, IEEE Computer Society, v. 4, n. 06, p. 6–21, 1984.

KANAWATI, G. A. et al. Evaluation of integrated system-level checks for on-line error detection. In: IEEE. **Proceedings of IEEE International Computer Performance and Dependability Symposium**. [S.l.], 1996. p. 292–301.

KRÜGER, J.; WESTERMANN, R. Linear algebra operators for gpu implementation of numerical algorithms. In: **ACM SIGGRAPH 2005 Courses**. [S.l.: s.n.], 2005. p. 234–es.

LEE, P. A.; ANDERSON, T. Fault tolerance. In: **Fault Tolerance**. [S.l.]: Springer, 1990. p. 51–77.

LINDHOLM, E. et al. Nvidia tesla: A unified graphics and computing architecture. **IEEE micro**, IEEE, v. 28, n. 2, p. 39–55, 2008.

MAHMOUD, A. et al. Optimizing software-directed instruction replication for gpu error detection. In: IEEE. **SC18: International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.], 2018. p. 842–854.

OH, N.; MITRA, S.; MCCLUSKEY, E. Ed/sup 4/i: error detection by diverse data and duplicated instructions. **IEEE Transactions on Computers**, v. 51, n. 2, p. 180–199, 2002.

OH, N.; SHIRVANI, P. P.; MCCLUSKEY, E. J. Error detection by duplicated instructions in super-scalar processors. **IEEE Transactions on Reliability**, IEEE, v. 51, n. 1, p. 63–75, 2002.

PORTET, S. A. et al. Software-only triple diverse redundancy on gpus for autonomous driving platforms. In: IEEE. **2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)**. [S.l.], 2020. p. 82–88.

RANDELL, B. System structure for software fault tolerance. In: **Proceedings of the international conference on Reliable software**. [S.l.: s.n.], 1975. p. 437–449.

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.

RECH, P. et al. Soft-error effects on graphics processing units. **FPGAs and Parallel Architectures for Aerospace Applications: Soft Errors and Fault-Tolerant Design**, Springer, p. 309–325, 2016.

RECH, P. et al. Impact of gpus parallelism management on safety-critical and hpc applications reliability. In: IEEE. **2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks**. [S.l.], 2014. p. 455–466.

REIS, G. A. et al. Design and evaluation of hybrid fault-detection systems. In: IEEE. **32nd International Symposium on Computer Architecture (ISCA'05)**. [S.l.], 2005. p. 148–159.

REIS, G. A. et al. Swift: Software implemented fault tolerance. In: IEEE. **International symposium on Code generation and optimization**. [S.l.], 2005. p. 243–254.

SERRANO-CASES, A. et al. Analysis of kernel redundancy for soft error mitigation on embedded gpus. **IEEE Transactions on Nuclear Science**, v. 70, n. 8, p. 1700–1707, 2023.

SHABBAR, A. Convolution neural networks: Intersection of deep learning and image processing in computational art. In: **10th International Conference on Digital and Interactive Arts**. New York, NY, USA: Association for Computing Machinery, 2022. (ARTECH 2021). ISBN 9781450384209.

SHI, G. et al. On testing gpu memory for hard and soft errors. In: **Proc. Symposium on Application Accelerators in High-Performance Computing**. [S.l.: s.n.], 2009. v. 107.

STERPONE, L. et al. Analysis and mitigation of soft-errors on high performance embedded gpus. In: **2022 21st International Symposium on Parallel and Distributed Computing (ISPDC)**. [S.l.: s.n.], 2022. p. 91–98.

SULLIVAN, M. et al. **System and methods for hardware-software cooperative pipeline error detection**. [S.l.]: Google Patents, 2020. US Patent 10,621,022.

VELAZCO, R.; FOUILLAT, P.; REIS, R. **Radiation effects on embedded systems**. [S.l.]: Springer Science & Business Media, 2007.