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

Márcio Gonçalves, Mateus Saquetti, José Rodrigo Azambuja

Resumo— Este trabalho apresenta uma abordagem de tolerância a falhas baseada em *software* de baixo nível para detectar SEUs nos bancos de registradores de dados, de endereço e de predicado de GPUs. As técnicas foram aplicadas sobre quatro algoritmos de estudo de caso, os quais foram executados em uma *soft-core* GPGPU, baseada na GPU NVIDIA G80. Campanhas de injeção de falhas foram simuladas em nível RTL sobre os três bancos de registradores e também sobre os registradores de pipeline da GPGPU, durante a execução das quatro aplicações em suas versões originais e protegidas. Os resultados mostram redução em erros de até 100% e custos de tempo de execução e de ocupação de memória de até 1,78 e 2,15 vezes os valores obtidos das aplicações originais, respectivamente.

***Palavras-chave*— GPUs, SEUs, tolerância a falhas**

I. INTRODUÇÃO

UNIDADES de processamento gráfico (GPUs) são sistemas dedicados ao processamento paralelo de alto desempenho, que possuem múltiplos núcleos em sua arquitetura e tiram proveito do paralelismo em nível de *thread* (TLP) para lidar com cálculos em computação gráfica [1][2]. A capacidade de manipular rapidamente grandes blocos de memória e executar várias tarefas elementares em paralelo a altas velocidades tornam GPUs mais eficazes do que os microprocessadores de uso geral (GPPs) para processamento de algoritmos em que dados possam ser executados em paralelo. Exemplos de aplicações onde tais algoritmos são utilizados são: exploração de petróleo; análise do fluxo de tráfego aéreo; processamento de imagens médicas; álgebra linear; estatística; reconstrução 3D; determinação de preço de venda de ações no mercado financeiro [3].

O advento significativo do suporte à programação de GPUs provocou a rápida disseminação desses dispositivos, tendo atraído programadores à utilizar GPUs até mesmo em aplicações que requerem alto grau de confiabilidade, tais como aplicações automotivas, médicas e espaciais [4][5][6]. Nessas aplicações, o uso de técnicas de tolerância a falhas é obrigatório para detectar ou corrigir falhas, uma vez que esses sistemas devem continuar funcionando corretamente mesmo sob a existência de falhas. No entanto, a confiabilidade de GPUs ainda é uma questão em aberto.

A demanda por desempenho computacional fez com que os últimos modelos de GPUs passassem a ser desenvolvidos em processos de fabricação menores que 28 nanômetros, operando a frequências de *clock* superiores a 1 GHz. O aumento na frequência de operação de sistemas digitais, combinado com os respectivos aumentos de densidade de transistores e redução de tensão de alimentação, tornam os circuitos integrados cada vez mais suscetíveis a falhas induzidas por radiação [7]. Tais falhas, causadas principalmente por partículas energizadas de radiação, fazem com que as mais recentes GPUs estejam sujeitas a experimentar erros induzidos por essas partículas [8][9], até mesmo em aplicações terrestres, em execução ao nível do solo, os nêutrons são as principais fontes de origem de erros [10].

Um dos principais tipos de falhas observado em circuitos de alta densidade de transistores expostos a radiação é o *Single Event Upset* (SEU), que é um evento transiente que pode ocorrer em qualquer elemento de memória [11]. GPUs possuem uma grande quantidade de elementos de memória e o efeito de SEUs deve ser investigado em todas essas estruturas. Uma maneira de qualificar o comportamento de uma GPU sob a ocorrência desses eventos é expondo o circuito integrado a um acelerador de partículas capaz de gerar um feixe de nêutrons ou íons pesados sobre o dispositivo [12]. Outra opção é a simulação de campanhas de injeção de falhas.

Para lidar com erros induzidos por radiação, técnicas de *software* de tolerância a falhas têm sido propostas para GPPs nos últimos anos, demonstrando taxas elevadas de detecção e baixa degradação de desempenho [13]. Tais técnicas podem ser aplicadas automaticamente sobre o código *assembly* de um programa, simplificando, assim, a tarefa para desenvolvedores de *software*. Deste modo, os custos de desenvolvimento podem ser reduzidos significativamente [14].

Este trabalho apresenta a utilização de três técnicas de tolerância a falhas baseadas em *software*, para proteger os bancos de registradores de GPUs contra SEUs. Cada técnica tem o objetivo de proteger um banco de registradores diferente. Para realização deste trabalho, foi utilizada uma *soft-core* GPGPU baseada na arquitetura G80 da NVIDIA, que executa quatro aplicativos de estudo de caso. Os códigos do

Márcio Gonçalves é mestrando do curso de Mestrado em Engenharia de Computação do Programa de Pós Graduação em Computação da Universidade Federal do Rio Grande. (marciogoncalves@furg.br)

Mateus Saquetti é graduando do curso de Engenharia de Computação da Universidade Federal do Rio Grande. (mateussaquetti@furg.br)

José Rodrigo Azambuja é Doutor em Computação pela Universidade Federal do Rio Grande do Sul e professor adjunto da Universidade Federal do Rio Grande do Sul. (jrfazambuja@furg.br)

programa são transformados em nível *assembly* e os custos de desempenho e de ocupação de memória são avaliados. Por fim, campanhas de injeção de falhas são realizadas em *Register Transfer Level* (RTL) sobre os bancos de registradores (dados, predicado e endereço) e sobre os registradores do pipeline da GPGPU a fim de avaliar a eficiência e a eficácia das técnicas de tolerância a falhas implementadas.

II. TRABALHOS RELACIONADOS

Técnicas de tolerância a falhas baseadas em *software* têm sido utilizadas para proteger muitas arquiteturas de processamento, incluindo GPUs [15]. Elas são classificadas, principalmente, em três classes: (1) *naïve duplication*, onde o programa é duplicado, (2) duplicação seletiva, onde somente partes do código são duplicadas, aproveitando-se de recursos de *hardware* não utilizados (por exemplo, registradores, memória *threads* e módulos de *hardware*), e (3) *Algorithm-Based Fault Tolerance* (ABFT), em que alguns determinados tipos de algoritmos podem ser protegidos.

Ao considerar a *naïve duplication*, os autores em [16] propõem diferentes abordagens para proteger uma GPU, mas o *overhead* em tempo de execução pode variar de 1.9 a 2.51 vezes o original, enquanto o uso de recursos é aumentado em 2 vezes. A duplicação seletiva é capaz de diminuir os custos em tempo de execução e a sobrecarga de recursos, reduzindo a taxa de detecção de falhas, tal como em [17], em que uma ferramenta denominada *Hauberk* é capaz de reduzir os erros SDC em 85% a um *overhead* de tempo de execução de 15%, porém, a abordagem não considera os erros do tipo HANG, que são essenciais em aplicações críticas, além disso, as simulações de falhas foram realizadas somente sobre as variáveis observadas no programa em alto nível de abstração. Técnicas ABFT são capazes de alcançar altas taxas de detecção com pouca sobrecarga de tempo de execução [18], mas estão limitadas a um grupo específico de aplicações.

Até onde sabemos, este é o primeiro trabalho na literatura a (1) aplicar técnicas de tolerância a falhas baseadas em *software* de baixo nível em GPUs, (2) proteger todos os bancos de registradores de uma GPU por *software*, e (3) executar campanhas de injeção de falhas, em nível RTL, direcionadas aos bancos de registradores e aos registradores de pipeline de uma GPU.

III. TÉCNICAS DE TOLERÂNCIA A FALHAS

As técnicas de tolerância a falhas apresentadas neste trabalho estendem [19] e são destinadas a proteger cada um dos bancos de registradores de uma GPU. Para isso, primeiro devemos considerar a arquitetura de bancos de registradores de GPUs. Normalmente estes dispositivos têm três tipos de bancos de registradores, que são utilizados para armazenar diferentes tipos de dados. São eles: (1) Banco de Registradores de Dados (VRF), utilizado para guardar dados intermediários para a execução das instruções; (2) Banco de Registradores de Predicados (PRF), utilizado para armazenar sinalizações de predicado, ou *flags*, que são utilizadas em instruções que geram execuções condicionais e (3) Banco de Registradores de Endereços (ARF), que guarda *offsets* de memória para operações de leitura e escrita.

Cada banco de registradores conta com diferentes instruções de suporte da *Instruction Set Architecture* (ISA) disponível para o dispositivo e, portanto, cada banco deve ser tratado com diferentes técnicas de proteção. Além disso, quando se utiliza técnicas de tolerância a falhas baseadas em *software*, a disponibilidade de recursos está diretamente traduzida em número de registradores disponíveis e até mesmo no comprimento dos registradores, que também deve ser levado em conta quando se lida com diferentes tipos de bancos de registradores. É importante ressaltar que, caso não haja registradores suficientes disponíveis, um subconjunto de registradores deve ser escolhido para ser protegido, o que reduzirá a capacidade de detecção de falhas, ou então, deve-se armazenar as réplicas na memória global ou compartilhada, o que causará custos em desempenho.

A fim de proteger efetivamente os bancos de registradores mais comumente encontrados em arquiteturas de GPUs, as seguintes subseções apresentarão três diferentes técnicas de tolerância a falhas baseadas em software.

A. Proteção dos Registradores de Dados (VRF Hard)

O principal objetivo da proteção para o VRF é proteger o sistema contra falhas que corrompem dados que podem causar erros diretos na aplicação ou erros indiretos, como o desvio de fluxo de controle errôneo. Para fazer isso, o código do programa original é modificado para que seja capaz de detectar falhas no armazenamento de dados. O conceito desta técnica tem sido utilizado em trabalhos anteriores [20], mas nunca foi aplicado e adaptado para GPUs.

A técnica de proteção para o VRF executa duas etapas, que são: (1) replicar as variáveis utilizadas pela aplicação, e (2) efetuar verificações de consistência entre os valores de sua réplica e o original. Para executar a primeira etapa, todos os registradores utilizados no código do programa devem ser replicados em registradores não utilizados. Para executar a segunda etapa, as verificações de consistência devem ser realizadas quando a memória é acessada (instruções de *load* e *store*), para que a memória não seja comprometida diretamente por falhas, e antes de instruções de *branch*, de modo que uma falha de dados não cause uma falha de controle, afetando o fluxo de execução do programa.

As Figs. 1, 2, 3, e 4 ilustram trechos de códigos em suas versões original e protegida. As figuras cobrem instruções de operação, de *load*, *store* e *branch* em código *assembly*. O código original pode ser visto no lado esquerdo da figura, enquanto o código protegido, com instruções adicionais em negrito, é mostrado à direita.

Para proteger instruções de operação, replica-se a instrução sobre as variáveis réplicas. Neste caso, não é necessário realizar a verificação de consistência, pois estas instruções não acessam a memória. A Fig. 1 mostra que, para replicar a instrução original 1, que realiza a operação sobre o registrador *R1*, a instrução 2 é adicionada, porém esta é modificada para operar sobre o registrador *R1'*, que é a réplica de *R1*.

As instruções de *load* e *store* fazem acessos à memória, e, portanto, é preciso que sejam realizados testes de consistência entre os registradores originais e suas réplicas. Para realizar estas transformações, duas instruções são inseridas: a primeira para comparar as variáveis e setar uma *flag* (ISET) em caso de discrepância; e a segunda para saltar para uma rotina de erro

Original Code	Hardened Code
1:IADD32I R1, R1, 1;	1:IADD32I R1, R1, 1; 2:IADD32I R1', R1', 1;

Figura 1. *VRF* – transformação para instrução de *operação*

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

Figura 2. *VRF* – transformações para instrução *store*

Original Code	Hardened Code
1:GLD R3,[R8];	1:GLD R3,[R8]; 2:MOV R3', R3 3:ISET.C1 R8, R8', NE; 4:BRA C1.NE, ERROR;

Figura 3. *VRF* – transformações para instrução *load*

(BRA), caso a *flag* indique que ocorreu discrepância. Estas transformações podem ser vistas na Fig. 2, em que a instrução original 1, é uma instrução de *store* cujas variáveis *R0* e *R6* são verificadas, pelas instruções 2 e 3, e 4 e 5, respectivamente, antes de efetivar a movimentação dos dados.

O mesmo padrão pode ser visto na Fig. 3, em que a instrução original 3 tem sua variável, armazenada em *R8*, verificada através das instruções 3 e 4. A fim de manter a consistência entre o registrador *R3* e sua réplica, a própria instrução de *load* também deveria ser replicada em *R3'*. No entanto, a fim de diminuir a degradação de desempenho que uma instrução extra de *load* causaria, esta instrução foi substituída por uma instrução *mov*, que pode ser observada na instrução 4. Dessa maneira, é introduzido um ponto de falha no sistema, porque se ocorrer uma falha durante a instrução de *load* sobre o registrador utilizado por esta instrução, tal falha não poderá ser detectada pois o valor incorreto que será carregado através da instrução *load* será copiado para o registrador réplica com a próxima instrução *mov*.

Apesar da instrução de *branch* não acessar diretamente a memória, os desvios condicionais baseiam-se em comparações realizadas sobre variáveis. Portanto, é necessário proteger estas instruções pois um desvio incorreto poderia levar o programa a executar *loads* ou *stores* imprevistos, que poderiam corromper a memória, e também fazer com que uma *thread* perca o sincronismo com as demais causando um erro

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

Figura 4. *VRF* – transformações para instrução de *branch*

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

B. Proteção dos Registradores de Predicado (PRF Hard)

O principal objetivo da proteção para o PRF é detectar falhas que causam erros no fluxo de controle da GPU, como *branchs* incorretos, finalização precoce do processo, dessincronização de *threads*, entre outros. Para fazê-lo, devemos proteger o PRF, bem como, todas as instruções que o acessam, o que ocorre em dois casos: (1) instruções de desvio condicional, as quais estão associadas a outra instrução de definição de *flag* de predicado; e (2) execução de instrução condicional, onde normalmente um bloco de instruções está associado a uma única instrução de definição de *flag* de predicado. Ambos os casos devem ser levados em conta.

A ideia por trás da técnica de proteção *PRF Hard* é semelhante à da técnica *VRF Hard*, uma vez que também replica-se os registradores do PRF. A principal diferença, porém, é que a maioria das ISAs de GPUs não fornecem instruções para comparação de valores entre registradores do PRF. A verificação de consistência, portanto, não pode ser feita de forma direta. Para fazê-la, em vez de comparar os valores originais e replicados, replicamos as instruções de *branch* de forma a identificar uma incompatibilidade entre elas. O principal problema é que as instruções de *branch* sempre tem dois possíveis caminhos de controle: quando o desvio é tomado e quando ele não é.

Quando o desvio não é tomado, uma instrução de *branch* pode ser simplesmente replicada e inserida logo após a instrução original, mas com um endereço de destino apontando para uma sub-rotina de erro. Se o desvio não é tomado na instrução original, ele também não deve ser tomado na instrução replicada, que será executada logo após a original.

Na possibilidade de que o desvio é tomado, uma instrução de *branch* replicada deve ser inserida no endereço de destino do desvio, que é a próxima instrução a ser executada pela GPU. A diferença é que se o desvio original foi tomado, o mesmo desvio deve ser tomado novamente. Para manter o fluxo original do programa, a instrução de *branch* replicada é invertida e seu endereço de destino passa a apontar para uma sub-rotina de erro. As figuras 5 e 6 mostram dois exemplos onde os casos 1 e 2 são cobertos, respectivamente.

A Fig. 5 mostra um exemplo em que uma instrução de *branch* é protegida. O desvio condicional original, instrução 3, é precedido pela instrução 1, que define o registrador *C0* do PRF, e seguido pelas instruções 5 e 8, que representam ambos os caminhos possíveis, se o desvio não é tomado e, se é

Original Code	Hardened Code
1: ISET.C0, R1, R10, EQ	1: ISET.C0, R1, R10, EQ **2: ISET.C0', R1, R10, EQ**
3: BRA C0.NE, 8	3: BRA C0.NE, **7**
5: IADD R6, 2	**4: BRA C0'.NE, ERROR** 5: IADD R6, 2
8: IADD R6, 1	**6: BRA 8** **7: BRA C0'.EQ, ERROR** 8: IADD R6, 1

Figura 5. PRF – transformações para instrução de *branch*

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

Figura 6. PRF – transformações para instruções condicionais

tomado, respectivamente. Para replicar a instrução 1, duas instruções adicionais são acrescentadas, utilizando o registrador *C0'* como réplica para registrador *C0*. A instrução 3 do desvio condicional original permanece a mesma no código transformado, mas com um endereço de destino atualizado para as instruções adicionais. A instrução 4 representa o desvio "não tomado" e, portanto, é a mesma que a original, mas com o destino apontando para uma sub-rotina de erro. A instrução 7 representa o caminho "tomado" e, portanto, é invertida (EQ em vez de NE) e aponta para o erro, ambas operando sobre a réplica *C0'* do PRF. A instrução extra 6 é necessária para garantir que o único ponto de entrada para a instrução 7 seja o desvio condicional original.

Ao considerar o segundo caso, onde um bloco de instruções condicionais é executado, o princípio permanece o mesmo. Como pode-se ver na Fig. 6, a instrução original 1 também é replicada sobre o registrador réplica *C0'*. A diferença é que, depois de todo o bloco ser executado (instruções 3 e 4), o registrador *C0* do PRF e sua réplica *C0'* devem ser verificados quanto a consistência. Assim, as instruções adicionais 5, 6, 7, e 8 são utilizadas para encadear *C0* e sua réplica *C0'*, em dois *branches* invertidos consecutivos.

C. *Proteção dos Registradores de Endereço (ARF Hard)*

O principal objetivo em proteger ARF é prevenir a GPU de acessar endereços de memória incorretamente. Tais acessos podem fazer com que o sistema corrompa a memória e cause erros que nem mesmo técnicas como *Error Correction Codes* (ECC) são capazes de detectar ou corrigir. A fim de fazer isso, a técnica aplicada utiliza o mesmo conceito da técnica de proteção aplicada para o VRF.

Assim como em *VRF Hard*, o método aplicado para o ARF replica os dados dos registradores de endereço do ARF e realiza verificações de consistência entre os registradores originais e as réplicas. As principais diferenças estão em dois aspectos que surgem ao lidar com ARF: (1) a ISA da GPU de estudo de caso não apresenta instruções que dão suporte para comparar diretamente registradores de ARF, e (2) o número de registradores de ARF normalmente é muito limitado.

Para resolver estas duas questões, ao invés de usar o banco de registradores ARF para armazenar os dados replicados, a técnica de proteção utiliza registradores do VRF. Ao fazer isso, é possível ter mais registradores disponíveis à um pequeno custo de ocupação, uma vez que o VRF possui mais registradores do que ARF bem como instruções que permitem comparar registradores diretamente. A principal desvantagem, por outro lado, é a degradação de desempenho e a ocupação de memória de programa causadas pelas instruções extras de movimentação de dados entre o ARF e o VRF.

A transformação aplicada ao código original considera duas operações: (1) quando uma instrução transfere dados de um registrador do VRF para um registrador do ARF ou, em outras palavras, quando um registrador ARF é escrito; e (2) quando uma instrução de acesso à memória lê um valor de um registrador do ARF para endereçar um elemento de memória através de uma instrução de acesso à memória.

No primeiro caso, a replicação do registrador do ARF é realizada de uma forma muito simples, uma vez que o valor original é armazenado no VRF. A Fig. 7 exemplifica este caso. Como pode ser observado, a instrução 1 move o dado do registrador *R3* do VRF, com deslocamento à esquerda (*shift left*) em 0x2, para o registrador *A1* do ARF, enquanto a instrução 2 cria uma réplica de *A1* em *R12*, ao mover *R3* deslocado à esquerda em 0x2 para o *R12*.

O segundo caso, em que registradores do ARF são lidos, é um pouco mais complicado, uma vez que é preciso verificar a consistência do registrador do ARF com sua réplica armazenada no VRF. Para fazer isso, primeiro é necessário mover o dado contido no registrador do ARF para um registrador do VRF. Deste modo é possível comparar os dados e notificar o usuário em caso de discrepância. A Fig. 8 mostra um exemplo do código de programa protegido que considera este caso. A instrução original 1 carrega um dado da memória compartilhada em *R0* através do endereçamento realizado pelo valor contido em *A1* adicionado de 0x8. A instrução adicional 2 move o dado do ARF *A1* para um registrador temporário do VRF, *R1*4, enquanto as instruções 3 e 4 realizam a verificação de consistência entre o registrador temporário *R14* e a réplica de *A1*, armazenada em *R12*.

IV. METODOLOGIA APLICADA

Original Code	Hardened Code
1: R2A A1, R3, 0x2;	1: R2A A1, R3, 0x2; **2: SHL R12, R3, 0x2;**

Figura 7. ARF – transformação para instrução de escrita

Original Code	Hardened Code
1: MOV R0, g [A1+0x8];	1: MOV R0, g [A1+0x8]; **2: A2R R14, A1;** **3: ISET.C1 R14, R12, NE;** **4: BRA C1.NE, ERRO;**

Figura 8. ARF - transformações para instrução de leitura

A metodologia utilizada para a elaboração do trabalho consistiu em seis passos, os quais são descritos a seguir.

A. Definição da GPU de Estudo de Caso

A GPU escolhida foi a *soft-core* GPGPU *FLEXible GRaphIcs Processor* (FlexGrip), baseada na arquitetura NVIDIA G80. Apesar de ser considerada uma GPU antiga, a G80 foi a primeira arquitetura a contar com o modelo de programação CUDA, que introduziu o conceito de *Computing Capability* (CC). Esta versão é também uma das poucas GPUs disponíveis na literatura com descrição em linguagem de *hardware*, sintetizável em circuito programável FPGA. Além disso, essa GPGPU foi testada em uma placa de desenvolvimento ML605 Virtex-6 [21] e apresenta seu código aberto, sem restrições para seu uso ou alterações, o que possibilita a alteração de seu *hardware* bem como a execução de campanhas de injeção de falhas através de simulação.

A arquitetura G80 possui 16 *Streaming Multiprocessors* (SMs) com 8 *Scalar Processors* (SPs) em cada SM. A FlexGrip, no entanto, possibilita que o sistema seja configurado para utilizar 8, 16 ou 32 *SPs*, o que permite explorar o *tradeoff* entre recursos de *hardware* e desempenho. O sistema conta com um escalonador de blocos que é responsável por escalonar blocos de *threads* através do método *round-robin* em um *Multiprocessor* (SM). O número máximo de blocos que podem ser escalonados para um SM é restringido pela quantidade de registradores e de memória compartilhada disponíveis.

O SM possui um escalonador de *warps*, os quais são criados quando o número de *threads* supera o número de *SPs*, ou seja, *warps* são grupos de *threads* que são escalonadas através do método *round-robin*. O número máximo de 24 *warps* podem ser instanciados em um SM. O SM é composto de um pipeline de 5 estágios (busca, decodificação, leitura, execução, escrita), e três bancos de registradores. Cada *warp* tem seu próprio *Program Counter* (PC) e pode ter até 32 *threads*. O controle individual para cada *thread* é realizado através de máscaras. O estágio de execução do pipeline pode contar com até 32 SPs, onde as *threads* são escalonadas e executadas. Os estágios do pipeline estão conectados através de blocos de registradores compostos por um total de 165 registradores que compreendem 1636 *bits*. Cada bloco de registradores, entre dois estágios do pipeline, possui um sinal de *enable* para o controle da transferência dos sinais entre os estágios.

Os bancos de registradores foram desenvolvidos para serem implementados fisicamente em FPGA em uma memória BRAM e estão divididos em VRF, PRF e ARF. Cada SP possui os seus próprios bancos de registradores (VRF, PRF e ARF), o que permite o acesso paralelo aos dados. Cada banco VRF contém 288 registradores de 32 bits, divididos entre as *threads* que são executadas pelo mesmo SP, para armazenamento de dados durante a execução da instrução. Cada banco PRF conta com 96 registradores de 4 bits, utilizados pelas *threads* para armazenar *flags* de predicado para o controle de desvios condicionais e para a execução de instruções condicionais. Os bancos ARF são compostos por 96 registradores de 32 bits, e são utilizados para armazenar valores de *offset* para endereçamento de memória. Tanto para PRF quanto para ARF, um número fixo de apenas 4 registradores é disponibilizado para cada *thread* instanciada pela aplicação.

B. Algoritmos de Estudo de Caso

A fim de avaliar a eficácia do modelo de injeção de falhas proposto escolhemos quatro algoritmos de estudo de caso: multiplicação de matrizes de dimensões 8x8, *bitonic sort*, autocorrelação e redução. Todos os estudos de casos são aplicações simples, mas divergem quanto à utilização do caminho de controle e de dados da GPGPU. Em termos de orientação ao fluxo de dados e de controle, a multiplicação matricial e a autocorrelação são em sua maior parte orientadas ao fluxo de dados, com poucos desvios condicionais, enquanto que a ordenação e a redução são principalmente orientadas ao fluxo de controle, com muitos desvios condicionais. As aplicações de estudo de caso multiplicação de matrizes, *bitonic sort*, autocorrelação e redução utilizam, em termos dos bancos de registradores VRF-PRF-ARF, 11-1-0, 6-1-2, 8-1-0, e 6-1-2 registradores por *thread*, respectivamente, e foram implementadas utilizando 64-8-8-8 *threads*, respectivamente.

C. Implementação das Técnicas de Tolerância a Falhas

Os códigos de programa foram protegidos manualmente, apesar de atualmente estarmos trabalhando na integração da transformação do código fonte para a ferramenta *Hardening Post Compiling Translator* (TCPH), que pode executar automaticamente todos os passos necessários para proteger o código [22]. As versões protegidas compreendem as técnicas de tolerância a falhas apresentadas na Seção III para proteger bancos de registradores de dados, predicados e endereços. As tabelas I e II apresentam os custos de tempos de execução e ocupação de memória, respectivamente, para as aplicações de estudo de caso.

Como pode ser observado na Tabela I, os tempos de execução para *VRF Hard* variam de 1,32 a 1,78 em relação aos tempos de execução dos algoritmos originais. A multiplicação de matrizes apresenta o menor *overhead* devido ao percentual de acessos à memória global realizado (suas réplicas são substituídas por instruções *mov*, como mostrado na Fig. 3, que são mais rápidas do que as instruções *load*), e o fato das instruções de *load* levarem os tempos de execução mais longos para serem concluídas.

Tabela I: Tempo de execução em milissegundos

Algoritmo de Estudo de Caso	Código Original	VRF Hard	PRF Hard	ARF Hard
Multiplicação de Matrizes	**0,33**	**0,43** (1,32x)	**0,34** (1,04x)	-
Bitonic Sort	**0,33**	**0,55** (1,68x)	**0,45** (1,36x)	**0,52** (1,59x)
Autocorrelação	**0,27**	**0,47** (1,71x)	**0,31** (1,13x)	**-**
Redução	**0,13**	**0,23** (1,78x)	**0,16** (1,26x)	**0,16** (1,27x)

Para a técnica *PRF Hard*, o tempo de execução é proporcional ao número de instruções de desvios condicionais e de instruções que utilizam *flags* de predicado para serem

executadas. Por esta razão, os dois algoritmos orientados ao controle *bitonic sort* e redução apresentaram os tempos de execução mais elevados, variando de 1.26 a 1.36 vezes o tempo dos algoritmos originais.

A técnica *ARF Hard* apresentou um custo de desempenho proporcional à utilização dinâmica de registradores de ARF, que é maior no *bitonic sort* do que na redução, demonstrando um tempo de execução de 1.27 a 1.59 vezes o tempo dos algoritmos originais. É interessante observar que o *overhead* entre diferentes aplicações pode ser maior que o dobro.

Tabela II: Ocupação de memória em bits

Algoritmo de Estudo de Caso	Código Original	VRF Hard	PRF Hard	ARF Hard
Multiplicação de Matrizes	**3200**	**6656** (2,08x)	**3840** (1,20x)	-
Bitonic Sort	**2944**	**6144** (2,09x)	**4352** (1,48x)	**4544** (1,54x)
Autocorrelação	**2112**	**4352** (2,06x)	**2880** (1,36x)	-
Redução	**2112**	**4544** (2,15x)	**3072** (1,45x)	**2944** (1,39x)

A Tabela II mostra que a quantidade de memória ocupada pelos algoritmos protegidos com a técnica *VRF Hard* supera o dobro da quantidade utilizada pelos algoritmos originais, variando de 2,06 a 2,15 vezes os valores originais. Em *PRF Hard*, a ocupação de memória variou de 1,20 a 1,48 enquanto *ARF Hard* utilizou de 1,39 a 1,54 vezes a quantidade de memória utilizada pelos algoritmos originais. Isso ocorre devido às instruções adicionais que são inseridas, porém, não representa um impacto real no sistema, uma vez que a memória de programa, representa um pequeno percentual da memória total do sistema, normalmente tem baixo custo e pode ser protegida através da utilização de tecnologias de memórias tolerantes a falhas como FLASH ou EEPROM.

D. Injetor de Falhas (GPUFI)

Uma das principais vantagens oferecida pela GPGPU FlexGrip é a possibilidade de simular efeito causado por falhas induzidas por partículas de radiação diretamente em elementos de *hardware* do sistema. Assim, foi desenvolvido um *script* em linguagem TCL (*Tool Command Language*), denominado GPUFI, que modela um injetor de falhas que reproduz SEUs nos bancos de registradores e nos registradores de pipeline da GPGPU FlexGrip, durante a execução de uma aplicação no simulador ModelSim.

Para simular SEUs nos bancos de registradores, o valor de um *bit* arbitrário é comutado em um momento arbitrário através do comando *change.* Para simular SEUs nos em registradores do pipeline não é possível é utilizar o comando *change*, pois estes registradores não são implementados em memória. Sendo assim, para simular SEUs nos registradores do pipeline, o comando *force* é utilizado. O injetor define um momento e um *bit* arbitrários para execução da falha. Quando ocorre o momento da injeção, a execução da aplicação é continuada até que o sinal de *enable* do registrador selecionado esteja em nível alto, para então forçar um *bit-flip*. Dessa maneira, SEUs são escritos nos registradores, e, assim, mantém-se até que os registradores sejam escritos novamente ou até o fim da execução do programa.

Para avaliar a confiabilidade do sistema, as falhas são classificadas de acordo com os efeitos que elas causam no sistema. Quando o programa conclui sua execução com os resultados esperados, a falha é caracterizada como *unnecessary for Architecturally Correct Execution* (unACE). Quando o programa termina sua execução normalmente, mas apresenta um resultado errôneo, a falha é classificada como *Silent Data Corruption* (SDC). Finalmente, quando uma falha faz com que o programa seja encerrado anormalmente ou faz com que o programa entre em *loop* infinito, ela é classificada como HANG.

E. Campanhas de Injeção de Falhas

A fim de validar a suscetibilidade de GPUs a SEUs e a eficácia das técnicas de detecção de falhas implementadas, foram executadas duas distintas campanhas de injeção de falhas sobre a GPGPU FlexGrip através de simulação em nível RTL no simulador ModelSim SE 10.1c. Primeiro, as falhas foram injetadas sobre os três bancos de registradores da FlexGrip. Após, as falhas foram injetadas sobre os registradores do pipeline da GPGPU. As campanhas foram realizadas automaticamente através do injetor GPUFI, e cada uma delas foi realizada em dois passos. No primeiro passo, as falhas foram injetadas durante a execução dos algoritmos de estudo de caso originais, com o objetivo de verificar a suscetibilidade da GPGPU a SEUs. No segundo passo, as falhas foram injetadas durante a execução das aplicações protegidas, para verificar a eficácia das técnicas de proteção. Para cada campanha de injeção de falhas, 10.000 simulações foram realizadas e apenas uma falha foi injetada em cada simulação.

Conforme mencionado anteriormente, a FlexGrip possui três bancos de registradores (VRF, PRF e ARF), e este trabalho propõe três diferentes técnicas de tolerância a falhas, cada uma com o objetivo de detectar falhas em um banco de registradores específico. Além disso, este trabalho também propõe submeter o pipeline da GPGPU à SEUs para analisar o comportamento do dispositivo. Assim, as campanhas de injeção de falhas foram divididas em quatro subcategorias, dependendo da fonte de falhas, sendo que as falhas foram injetadas em (a) VRF, (b) PRF, (c) ARF e (d) registradores do pipeline. Para os bancos de registradores (VRF, PRF e ARF), as falhas foram injetadas somente sobre os registradores endereçados pelo compilador, o que significa que os registradores que não são escritos ou lidos pela aplicação foram removidos das campanhas de injeção.

Para fazer uma análise completa sobre as técnicas propostas, as falhas foram injetadas em cada um dos três bancos de registradores, bem como nos registradores do pipeline, durante a execução das aplicações com VRF, PRF e ARF protegidos. Deste modo, é possível analisar se uma técnica de tolerância a falhas, desenvolvida para proteger um banco de registradores específico, é capaz de detectar falhas originadas nos outros bancos. Ademais, é possível verificar se as técnicas desenvolvidas para proteger os bancos de registradores de GPUs possuem algum potencial de redução de erros causados por falhas no pipeline.

É importante ressaltar que a quantidade de *bits* de cada banco de registradores e dos registradores do pipeline é diferente, o que significa que, embora a quantidade absoluta

de falhas seja exatamente a mesma para esses registradores, os efeitos da radiação em aplicações reais tenderão a aparecer em maior quantidade no VRF, seguido do ARF, PRF e pipeline em uma relação percentual de aproximadamente 72/24/3/1, respectivamente.

V. RESULTADOS DAS INJEÇÕES DE FALHAS

A. *Campanha de Injeção de Falhas no VRF*

Nesta campanha de injeção de falhas, as falhas foram injetadas apenas no VRF. A Tabela III mostra os resultados das injeções para as aplicações de estudo de caso originais, enquanto as Tabelas IV, V e VI apresentam os resultados das injeções para os algoritmos protegidos em VRF, PRF e ARF, respectivamente.

Tabela III: Resultados para as versões originais

Algoritmo de Estudo de Caso	unACE	ACE	
		SDC	HANG
Multiplicação	70,8%	19,9%	11,3%
Bitonic Sort	72,6%	6,2%	21,2%
Autocorrelação	72,5%	21,4%	6,1%
Redução	76,6%	7,7%	15,7%
Total	73,1%	13,3%	13,6%

Conforme pode ser observado na Tabela III, de todas as falhas injetadas, 73,1% resultaram em unACE, o que significa que elas não causaram erro no resultado final da aplicação. Por outro lado, os resultados demonstram que 26,9% das falhas injetadas causaram erros no sistema, sendo que 13,3% das falhas resultaram em SDC e 13,6% resultaram em HANG.

As aplicações orientadas a dados mostraram um número mais elevado de SDC (19,9% e 21,4%), enquanto as aplicações orientadas ao controle mostraram um número mais elevado de HANGs (15,7% e 21,2%). Tais HANGs ocorrem devido ao maior número de instruções condicionais que, quando corrompidas por alguma falha, podem fazer com que as *threads* percam o sincronismo, o que pode levar a GPU para um estado de HANG. As aplicações orientadas a dados são menos suscetíveis a desvios de controle, mas ainda são sensíveis a falhas nos registradores de dados, resultando em um maior número de SDCs. Isso ocorre pois falhas em dados, embora não interfiram no funcionamento das instruções, alteram os valores esperados das variáveis que são executadas por estas instruções, o que pode acarretar na propagação de dados incorretos até a saída do sistema.

Tabela IV: Resultados para *VRF Hard*

Algoritmo de Estudo de Caso	Detectado	unACE	ACE	
			SDC	HANG
Multiplicação	62,8%	36,6%	0,6%	0,0%
Bitonic Sort	62,7%	36,9%	0,3%	0,1%
Autocorrelação	44,3%	55,3%	0,4%	0,0%
Redução	44,9%	54,5%	0,3%	0,3%
Total	54,0%	45,5%	0,4%	0,1%

Quando a técnica de proteção do VRF é aplicada às aplicações de estudo de caso, as falhas são reduzidas de 26,9% para 0,5%, o que representa uma redução de 98,1%. Ao considerar falhas do tipo HANG, a redução foi de 99,9%, de 13,6% na versão original para 0,1% na versão protegida. Os resultados não alcançam 100% de detecção de falhas por causa do ponto de falha em que a replicação da instrução de *load* é feita utilizando uma simples instrução de *mov* ao invés de uma segunda instrução de *load*.

É importante mencionar que o VRF é o maior banco de registradores em termos de quantidade de utilização e comprimento de registradores, e, consequentemente, este banco possui um maior número de bits suscetíveis a falhas. O VRF é, portanto, mais propenso a ser perturbado por eventos de radiação. Considerando esta informação, a redução alcançada de 98,1% em SDCs e 99,9% em HANGs pode ser ainda maior em aplicações reais.

Tabela V: Resultados para *PRF Hard*

Algoritmo de Estudo de Caso	Detectado	unACE	ACE	
			SDC	HANG
Multiplicação	0,1%	68,1%	19,0%	12,8%
Bitonic Sort	0,7%	70,3%	5,4%	23,6%
Autocorrelação	0,2%	72,1%	21,9%	5,8%
Redução	8,2%	76,6%	5,1%	10,1%
Total	2,3%	71,7%	12,9%	13,1%

Tabela VI: Resultados para *ARF Hard*

Algoritmo de Estudo de Caso	Detectado	unACE	ACE	
			SDC	HANG
Bitonic Sort	10,5%	61,4%	6,0%	22,1%
Redução	3,1%	76,0%	6,2%	14,7%
Total	6,8%	68,7%	6,1%	18,4%

Quando as técnicas *PRF Hard* e *ARF Hard* são utilizadas para proteger VRF, os resultados são apenas um pouco melhores. Para as primeiras três aplicações, PRF protegido não foi capaz de detectar mais de 1% das falhas, mas alcançou 8,2% de detecção para o algoritmo de redução. *ARF Hard*, por sua vez, embora tenha sido capaz de detectar 10,5% das falhas no *bitonic sort*, o número de falhas manteve-se próximo dos 26,9% obtido na aplicação original. É importante mencionar que a proteção do ARF foi apenas aplicada aos algoritmos orientados ao controle, e, assim, mostrou resultados inferiores em relação ao total obtido do conjunto completo das aplicações originais.

B. *Campanha de Injeção de Falhas no PRF*

Nesta campanha, as falhas foram injetadas apenas no PRF. A Tabela VII mostra os resultados das injeções para as aplicações de estudo de caso originais, enquanto as Tabelas VIII, IX e X apresentam os resultados para as proteções de PRF, VRF e ARF, respectivamente.

A Tabela VII indica a suscetibilidade do PRF a SEUs. Como pode ser observado, apenas 2,9% de todas as injeções resultaram em erros (0,7% HANGs e 2,1% SDCs). Estes valores podem ser considerados baixos quando se leva em conta que o PRF é composto por registradores de 4 bits e cada *thread* utiliza apenas 4 registradores. Ainda assim, tais falhas devem ser levadas em consideração no projeto de técnicas de tolerância para proteger GPUs.

Tabela VII: Resultados para as versões originais

Algoritmo de Estudo de Caso	unACE	ACE	
		SDC	HANG
Multiplicação	99,3%	0,5%	0,2%
Bitonic Sort	97,4%	1,5%	1,1%
Autocorrelação	97,4%	2,2%	0,4%
Redução	94,3%	4,6%	1,1%
Total	97,1%	2,2%	0,7%

Tabela VIII: Resultados para *PRF Hard*

Algoritmo de Estudo de Caso	Detectado	unACE	ACE	
			SDC	HANG
Multiplicação	1,0%	98,8%	0,0%	0,2%
Bitonic Sort	4,6%	94,8%	0,0%	0,6%
Autocorrelação	3,1%	96,9%	0,0%	0,0%
Redução	21,3%	77,9%	0,6%	0,2%
Total	7,5%	92,1%	0,1%	0,3%

Quando a técnica de proteção de *PRF Hard* é aplicada, a pequena porcentagem de SDCs foi reduzida para 0, exceto no algoritmo de redução que mostrou 0,6% de erros (87% de redução), enquanto os HANGs foram reduzidos em 64,3%. Embora a proteção tenha sido capaz de eliminar os erros do tipo SDC, a técnica não foi capaz de alcançar as mesmas taxas de detecção para os erros do tipo HANG. Isso ocorre porque GPUs executam múltiplas *threads* que trabalham de maneira sincronizada, e, deste modo, por exemplo, se uma *thread*, que não conta com instruções precedentes de sincronismo, não realiza um salto esperado por causa de uma falha em PRF, ela pode perder o sincronismo com as *threads* remanescentes. Neste caso, o *hardware* de controle de fluxo da GPU pode optar por priorizar a execução das *threads* remanescentes, assim, o programa não encontra instruções de retorno para retomada da *thread* que teve seu fluxo de execução alterado. Assim, a falha não é detectada pela técnica pois a instrução de proteção que leva a *thread* errônea à rotina de erro está situada após a instrução de desvio que causou a dessincronização.

Tabela IX: Resultados para a técnica de proteção do VRF

Algoritmo de Estudo de Caso	Detectado	unACE	ACE	
			SDC	HANG
Multiplicação	0,0%	98,5%	1,3%	0,3%
Bitonic Sort	0,4%	95,7%	1,5%	2,4%
Autocorrelação	0,0%	97,3%	2,1%	0,6%
Redução	2,8%	90,2%	5,5%	1,5%
Total	0,8%	95,4%	2,6%	1,2%

Tabela X: Resultados para a técnica de proteção do ARF

Algoritmo de Estudo de Caso	Detectado	unACE	ACE	
			SDC	HANG
Bitonic Sort	0,0%	98,0%	1,0%	0,9%
Redução	1,7%	93,8%	3,9%	0,7%
Total	0,9%	95,8%	2,5%	0,8%

As técnicas de proteção de VRF e ARF, que não foram projetadas para detectar falhas em PRF, aumentaram um pouco o número de erros, de 2,9% para 3,8% e 3,3%, respectivamente. Uma vez que as duas técnicas de proteção não afetam o PRF, e, teoricamente, não podem fazer o PRF mais suscetível a falhas, logo o acréscimo em menos de 1% deveria ser descartado.

C. Campanha de Injeção de falhas no ARF

Nesta campanha de injeção de falhas, as falhas foram injetadas somente no ARF. A Tabela XI mostra os resultados das injeções para as aplicações de estudo de caso originais, enquanto as tabelas XII, XIII e XIV apresentam os resultados para as proteções de ARF, VRF e PRF, respectivamente.

Tabela XI: Resultados para as versões originais

Algoritmo de Estudo de Caso	unACE	ACE	
		SDC	HANG
Bitonic Sort	96,1%	3,9%	0,0%
Redução	98,2%	1,8%	0,0%
Total	97,2%	2,8%	0,0%

Como pode ser observado na Tabela XI, ARF não é tão sensível à falhas como VRF. De todas as falhas injetadas, apenas 2,8% causaram SDCs e nenhuma falha causou HANG. Esta diferença entre SDCs e HANGs acontece, principalmente, devido à leitura de dados incorretos, sendo que, nesta situação, SDCs são mais propensos a ocorrer. Armazenar dados incorretos poderia causar HANGs se o código de programa fosse sobrescrito, o que não é possível em arquiteturas de memória *Harvard*.

Tabela XII: Resultados para a técnica de proteção do ARF

Algoritmo de Estudo de Caso	Detectado	unACE	ACE	
			SDC	HANG
Bitonic Sort	41.6%	58.4%	0.0%	0.0%
Redução	73.5%	26.5%	0.0%	0.0%
Total	57.6%	42.4%	0.0%	0.0%

Apesar do pequeno percentual de erros, a técnica *ARF Hard* foi capaz de detectar 100% das falhas injetadas que causam erros no sistema, reduzindo o número de erros a zero. Em termos de detecção de falhas, estes resultados são excelentes.

Quando as injeções foram direcionadas para ARF, as técnicas de proteção de VRF e PRF não foram capazes de reduzir os erros. Embora tenham sido capazes de detectar 0,2% e 0,1%, respectivamente, o número de erros aumentou em 0,1%, ou por um fator de 3,5%, o que pode ser considerado o mesmo resultado obtido para as aplicações originais. Estes resultados mostram que, para proteger o ARF, a técnica destinada para o ARF deve ser utilizada.

Tabela XIII: Resultados para a técnica de proteção do VRF

Algoritmo de Estudo de Caso	Detectado	unACE	ACE	
			SDC	HANG
Bitonic Sort	0,0%	95,8%	4,2%	0,0%
Redução	0,5%	97,9%	1,6%	0,0%
Total	0,2%	96,9%	2,9%	0,0%

Tabela XIV: Resultados para a técnica de proteção do PRF

Algoritmo de Estudo de Caso	Detectado	unACE	ACE	
			SDC	HANG
Bitonic Sort	0,3%	95,8%	3,9%	0,0%
Redução	0,0%	98,2%	1,8%	0,0%
Total	0.1%	97.0%	2.9%	0.0%

D. Campanha de Injeção de Falhas no Pipeline

Nesta campanha de injeção, as falhas foram injetadas sobre os registradores do pipeline da GPGPU. A Tabela XV mostra os resultados de injeção de falhas para as aplicações de estudo de caso originais, enquanto que as Tabelas XVI, XVII e XVIII apresentam os resultados para as técnicas de proteção *VRF Hard*, *PRF Hard* e *ARF Hard*, respectivamente.

Tabela XV: Resultados para as versões originais

Algoritmo de Estudo de Caso	unACE	ACE	
		SDC	HANG
Multiplicação	75,8%	14,2%	10,0%
Bitonic Sort	91,5%	4,1%	4,4%
Autocorrelação	92,7%	2,8%	4,5%
Redução	88,0%	5,3%	6,8%
Total	87,0%	6,6%	6,4%

A Tabela XV mostra que, de todas as falhas injetadas, 13.0% resultaram em erros no sistema sendo que 6,6% das falhas causaram SDCs e 6,4% causaram HANGs. Estes resultados demonstram que o pipeline é menos suscetível a erros do que o banco de registradores VRF, porém é mais sensível do que os bancos PRF e ARF.

A aplicação de estudo de caso mais afetada pelas falhas foi a multiplicação de matrizes. Isso ocorre porque as *threads* são executadas em *warps* que comportam 32 *threads.* A multiplicação de matrizes utiliza 64 *threads,* as quais são escalonadas em 2 *warps* totalmente utilizados. As outras aplicações, por outro lado, foram configuradas para utilizar somente 8 *threads*, o que resulta em apenas um *warp* em que 3/4 de sua capacidade não é utilizada pelas aplicações. Assim, quando uma falha afetar uma *thread* que não está sendo utilizada pela aplicação, esta falha tende a não se propagar no sistema com a mesma facilidade que as falhas que afetam as *threads* em uso.

A Tabela XV demonstra que, na maior parte das aplicações, as falhas tendem a gerar mais HANGs do que SDCs. Porém, isso não ocorre na multiplicação de matrizes porque esta aplicação utiliza o caminho de dados mais intensivamente, executando muitas instruções de acesso à memória, e, falhas durante a execução dessas instruções podem se propagar para os dados do sistema mais facilmente gerando um SDC. SDCs ocorrem quando as falhas se propagam no sistema alterando valores de registradores ou provocando um mau funcionamento de instruções que não interferem no fluxo de controle da aplicação. HANGs, por sua vez, tendem a acontecer quando falhas se propagam para instruções ou dados utilizados no controle da aplicação. Além disso, HANGs podem ocorrer quando as falhas afetam mecanismos de controle do pipeline como, por exemplo, os sinais responsáveis pelo mascaramento de execução de *threads* bem como os sinais de controle de execução de *warp* e de bloco. Sendo assim, a suscetibilidade a HANGs ou SDCs devido a falhas no pipeline depende das instruções que compõem o algoritmo em execução e da configuração da GPU. Por estas razões, os algoritmos não demonstraram a mesma tendência a SDCs e HANGs que pode ser observada quando as falhas foram injetadas sobre os bancos de registradores.

Tabela XVI: Resultados para a técnica de proteção do VRF

Algoritmo de Estudo de Caso	Detectado	unACE	ACE	
			SDC	HANG
Multiplicação	5,3%	74,4%	10,9%	9,4%
Bitonic Sort	2,6%	90,8%	2,4%	4,2%
Auto correlação	4,4%	90,4%	1,2%	4,0%
Redução	5,4%	86,9%	2,1%	5,6%
Total	4,4%	85,6%	4,1%	5,8%

A tabela XVI demonstra que quando a técnica *VRF Hard* foi aplicada, o total de SDCs caiu de 6,6% para 4,1% enquanto o total de HANGs caiu de 6,4% para 5,8%. Isso representa uma taxa de redução média em erros do tipo SDC e HANG de 37,1% e 9,7%, respectivamente. Tal redução de falhas ocorre, principalmente, devido às falhas que se propagam até os registradores do programa, os quais podem ser detectados pela técnica *VRF Hard*. Tais resultados demonstram que a técnica *VRF Hard,* além de apresentar elevada taxa de redução de erros que ocorrem devido às falhas em VRF, também apresenta benefícios para redução de erros, principalmente SDCs, causados por falhas no pipeline.

Tabela XVII: Resultados para a técnica de proteção do PRF

Algoritmo de Estudo de Caso	Detectado	unACE	ACE	
			SDC	HANG
Multiplicação	0,3%	75,8%	13,7%	10,2%
Bitonic Sort	2,0%	90,7%	3,0%	4,2%
Auto correlação	1,1%	92,7%	2,3%	4,0%
Redução	1,1%	88,8%	4,0%	6,1%
Total	1,1%	87,0%	5,7%	6,1%

Tabela XIII: Resultados para a técnica de proteção do BRE

Algoritmo de Estudo de Caso	Detectado	unACE	ACE	
			SDC	HANG
Bitonic Sort	1,9%	91,0%	3,1%	4,1%
Redução	1,5%	88,0%	3,7%	6,8%
Total	1,7%	89,5%	3,4%	5,4%

Quando a técnica *PRF Hard* foi aplicada, houve uma taxa de redução média em HANGs e SDCs de 12,8% e 4,6%, respectivamente. Quando a técnica *ARF Hard* foi aplicada, a taxa de redução média em HANGs e SDCs foi de 27,4% e 3,1%, respectivamente. Neste caso, houve uma taxa de redução média em SDCs superior ao obtido pela técnica *PRF Hard*, embora a taxa de redução média de HANGs tenha sido a menor entre as três técnicas. Essas taxas de redução em falhas não representam valores tão expressivos quanto os obtidos por *VRF Hard* porém ainda representam uma pequena melhora. *VRF Hard* foi capaz alcançar maiores taxas de redução de erros porque VRF é o banco de registradores mais utilizado pois a maior parte das instruções utilizam estes registradores. Assim, uma falha no pipeline pode se propagar para esses registradores com mais facilidade do que para ARF e PRF e, então, ser detectada através da técnica *VRF Hard.*

VI. CONCLUSÕES E TRABALHOS FUTUROS

Este artigo apresentou um conjunto de técnicas de tolerância a falhas baseadas em software projetadas para proteger os bancos de registradores de uma GPGPU contra SEUs. Três técnicas de proteção, *VRF Hard*, *PRF Hard* e *ARF Hard*

foram utilizadas para proteger quatro aplicações de estudo de caso.

Os resultados experimentais demonstraram que os custos, em termos de tempo de execução, variaram de 1,32 a 1,78 vezes em relação aos tempos originais para *VRF Hard*; de 1,04 a 1,60 para *PRF Hard*; e de 1,27 a 1,59 para *ARF Hard*. Ao considerar a utilização de memória de programa, os resultados mostraram-se até 2,24 vezes os valores originais para o *VRF Hard;* enquanto 1,48 vezes para *PRF Hard;* e 1,54 vezes para a técnica de proteção *ARF Hard*. Tais resultados de implementação mostram que técnicas de tolerância a falhas baseadas em software de baixo nível podem ser utilizadas em GPUs a custos de tempo de execução e consumo de memória de programa aceitáveis.

As campanhas de injeção de falhas foram concluídas com 360.000 falhas injetadas nos bancos de registradores e 140.000 falhas injetadas nos registradores do pipeline. As falhas foram direcionadas para os três bancos de registradores da GPGPU e para os registradores do pipeline da GPGPU. Quando as falhas foram injetadas nos bancos de registradores, as técnicas *VRF Hard* e *PRF Hard* demonstraram uma redução média de erros de 98,1% e 87%, respectivamente, enquanto *ARF Hard* foi capaz de reduzir os erros em 100%, o que significa que nenhuma falha foi capaz de causar erro no sistema. Por fim, quando as falhas foram injetadas sobre os registradores de pipeline, os resultados para as aplicações protegidas demonstraram alguns benefícios, embora as técnicas não tenham sido desenvolvidas com a finalidade de proteger o pipeline. Os resultados para *VRF Hard* apontaram uma redução média de erros de 26%, enquanto *PRF Hard e ARF Hard* apresentaram uma redução média de erros de 11,7% e 14,4%, respectivamente.

Até onde sabemos, este é o primeiro trabalho da literatura a aplicar técnicas de tolerância a falhas baseadas em software de baixo nível para proteger GPUs, bem como proteger todos os bancos de registradores e realizar campanhas de injeção de falhas em nível RTL orientadas aos bancos de registradores. Além disso, os resultados são promissores, a fim de auxiliar desenvolvedores a implementar projetos de tolerância a falhas para arquiteturas de alto desempenho de processamento paralelo, tais como GPGPUs.

Em continuidade a este trabalho, pretendemos realizar campanhas de injeções de falhas sobre outros componentes da GPGPU, como a memória *cache,* e explorar os pontos fracos das técnicas apresentadas para alcançar taxas de detecção mais elevadas. Pretendemos, também, explorar a duplicação seletiva, que possibilita o *tradeoff* entre *overhead* e taxa de detecção. Por fim, objetivamos aplicar as técnicas à GPUs comerciais e expor os dispositivos à feixes de nêutrons.

REFERÊNCIAS

[1] NVIDIA Next Generation CUDA Compute Architecture: Fermi [online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_architecture_Whitepaper.pdf

[2] AMD Graphics *SPs* Next Architecture [online]. Available: http://www.amd.com/la/Documents/GCN_Architecture_whitepaper.pdf

[3] J. Kruger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," ACM Trans. Graph, vol. 3, no. 22, pp. 908–916, 39–55, 2003.

[4] Q. Hu, B. Xiao, and M. Friswell, "Robust fault-tolerant control for spacecraft attitude stabilisation subject to input saturation," Control Theory Applications, vol. 5, no. 2, pp. 271–282, 2011.

[5] N. Zhang, "Investigation of Fault-Tolerant Adaptive Filtering for Noisy ECG Signals," in IEEE Symposium on Computational Intelligence in Image and Signal Processing, pp. 177–182, 2007.

[6] A. Strano, D. Bertozzi, A. Grasset, S. Yehia, "Exploiting structural redundancy of SIMD accelerators for their built-in self-testing/diagnosis and reconfiguration," in IEEE Int. Conf. on App.-Specific Systems, Architectures and Processors, 2011.

[7] R. C. Baumann, "Soft errors in advanced semiconductor devices-part I: the three radiation sources," in IEEE Transactions on Device and Materials Reliability. March 2001.

[8] C. Slayman and O. A. La Carte, "Soft errors-past history and recent discoveries," in Proc. IEEE Int. Integr. Reliab. Workhop, 2010, pp. 25–30, Invited Paper.

[9] A. Dixit and A. Wood, "The impact of new technology on soft error rates," in IEEE Int. Reliab. Phys. Symp. 2011.

[10] P. Rech, C. Aguiar, C. Frost, L. Carro, "An efficient and experimentally Tuned Software-Based Hardening Strategy for matrix multiplication on GPUs," IEEE Transactions on Nuclear Science, Vol. 60, pp. 2797-2804, 2013.

[11] P. Dodd, L. W. Massengill, "Basic mechanism and modeling of single-event upset in digital microelectronics," IEEE Transactions on Nuclear Science, Vol. 50, pp. 583-602, 2003.

[12] P. Rech, C. Aguiar, R. Ferreira, C. Frost, L. Carro, "Neutron radiation test of graphic processing units," IEEE On-Line Testing Symposium (IOLTS), pp. 55-60, 2012.

[13] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, "Software-Implemented Hardware Fault Tolerance," 1st ed. New York: Springer, 2006. Print.

[14] E. Rhod, C. Lisboa, L. Carro, M. S. Reorda, M. Violante, "Hardware and software transparency in the protection of programs against SEUs and SETs," in Journal of Electronic Testing, (24):45-56, 2008.

[15] M. Dimitrov, M. Mantor, H. Zhou, "Understanding software approaches for GPGPU reliability," Workshop on General Purpose Processing on Graphics Processing Units, pp. 94-104, 2009.

[16] P. Rech, L. Pilla, P. Navaux, L. Carro, "Impact of GPUs parallelism management on safety-critical and HPC applications reliability," IEEE Int. Conf. on Dependable Systems and Networks, pp. 455-466, 2014

[17] K. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, R. Iyer, "Hauberk: lightweight silent data corruption error detector for GPGPU," IEEE Parallel & Distributed Processing Symposium (IPDPS), pp. 287-300 , 2011.

[18] P. Rech, C. Aguiar, C. Frost, L. Carro, "An efficient and experimentally tuned software-based hardening strategy for matrix multiplication on GPUs," IEEE Trans. on Nucl. Sci., vol. 60, no. 4, pp. 2797–2804, 2013.

[19] M. Saquetti, M. Goncalves, J. R. Azambuja, "Evaluating the Efficiency of Software-based Fault Tolerant Techniques to Detect SEUs in GPUs," European Conference on Radiation and Its Effects on Components and Systems (RADECS), 2016.

[20] J. R. Azambuja, A. Lapolli, M. Altieri and F. Kastensmidt, "Evaluating the Efficiency of Data-flow Software-based Techniques to Detect SEEs in Microprocessors," Latin American Test Workshop (LATW), pp. 1-6, 2011.

[21] K. Andryc, M. Merchant, "FlexGrip: A soft GPGPU for FPGAs," IEEE Field-Programmable Technology (FPT), pp. 230-237, 2013.

[22] J. R. Azambuja, S. Pagliarini, M. Altieri, F. L. Kastensmidt, M. Hubner, J. Becker, G. Foucard, R. Velazco, "A fault tolerant approach to detect transient faults in microprocessors based on a non-intrusive reconfigurable hardware," IEEE Trans. Nucl. Sci., vol. 59, no. 4, pp. 1117-1124, 2012.