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

WERNER MAURICIO NEDEL

Analise dos Efeitos de Falhas Transientes no Conjunto de Banco de Registradores em Unidades Gráficas de Processamento

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

Prof. Dr. Fernanda Lima Kastensmidt
Orientador

Prof. Dr. José Rodrigo Azambuja
Co-orientador

Porto Alegre, dezembro de 2015.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Nedel, Werner Mauricio

Analise dos Efeitos de Falhas Transientes no Conjunto de Banco de Registradores em Unidades Gráficas de Processamento

/ Werner Mauricio Nedel – Porto Alegre: Programa de Pós-Graduação em Microeletrônica, 2015.

71 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica. Porto Alegre, BR – RS, 2015. Orientador: Fernanda Lima Kastensmidt; Co-orientador: José Rodrigo Azambuja.

1.GPU. 2.Processamento Paralelo 3.Tolerância a Falhas. I. Kastensmidt, Fernanda. II. Azambuja, José Rodrigo. III. Analise dos Efeitos de Falhas Transientes no Conjunto de Banco de Registradores em Unidades Gráficas de Processamento.

AGRADECIMENTOS

Agradeço a minha família por toda a ajuda despendida neste ano e principalmente pela compreensão com a minha constante falta de tempo e ausência.

À minha orientadora, Fernanda Lima Kastensmidt, por toda a compreensão nas últimas etapas do trabalho e principalmente por sempre acreditar na nossa pesquisa, incentivando sempre novas abordagens frente aos problemas enfrentados. Ao meu co-orientador, José Rodrigo Furlanetto Azambuja, por abrir as portas para mim nesta minha nova experiência no meio acadêmico e por ter sido de ajuda crucial na conclusão do trabalho.

Finalmente, agradeço a minha esposa, Aline Cunha de Moraes, por toda a compreensão, paciência, força de vontade e, principalmente, por estar sempre ao meu lado me incentivando a sempre buscar o aperfeiçoamento em tudo o que faço. Com certeza este trabalho não existiria sem a tua ajuda.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ABNT	Algorithm Based Fault Tolerance
API	Application Program Interface
ASIC	Application Specific Integrated Circuits
BRAM	Block Random-Access Memory
CC	Computing Capability
CPU	Central Processing Unit
CSV	Comma Separated Values
CUDA	Compute Unified Device Architecture
DOP	Degree Of Parallelism
ECC	Error-Correcting Code
FlexGrip	Flexible Graphics Processor
FIT	Failures In Time
GPGPU	General-Purpose Graphics Processing Unit
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HPC	High Performance Computing
IP	Intellectual Property
ISE	Integrated Synthesis Environment
MEBF	Mean Execution Between Failure
MTBF	Mean Time Between Failure
PC	Program Counter
PTX	Parallel Thread Execution
RAM	Random-Access Memory
RTL	Register-Transfer Level
SASS	Source And Assembly
SBST	Software Based-Self-Test
SEE	Single Event Effect

SEE	Single Event Effects
SEL	Single Event Latchup
SET	Single Event Transient
SEU	Single Event Upset
SFU	Special Function Unit
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprocessor
SP	Scalar Processor
TCL	Tool Command Language
TMR	Triple Modular Redundancy
VHDL	VHSIC Hardware Description Language

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Unidades gráficas de processamento, mais conhecidas como GPUs (*Graphics Processing Unit*), são dispositivos que possuem um grande poder de processamento paralelo com respectivo baixo custo de operação. Sua capacidade de simultaneamente manipular grandes blocos de memória a credencia a ser utilizada nas mais variadas aplicações, tais como processamento de imagens, controle de tráfego aéreo, pesquisas acadêmicas, dentre outras. O termo GPGPUs (*General Purpose Graphic Processing Unit*) designa o uso de GPUs utilizadas na computação de aplicações de uso geral. A rápida proliferação das GPUs com ao advento de um modelo de programação amigável ao usuário fez programadores utilizarem essa tecnologia em aplicações onde confiabilidade é um requisito crítico, como aplicações espaciais, automotivas e médicas. O crescente uso de GPUs nestas aplicações faz com que novas arquiteturas deste dispositivo sejam propostas a fim de explorar seu alto poder computacional. A arquitetura FlexGrip (*FLEXible GRaphIcs Processor*) é um exemplo de GPGPU implementada em FPGA (*Field Programmable Gate Array*), sendo compatível com programas implementados especificamente para GPUs, com a vantagem de possibilitar a customização da arquitetura de acordo com a necessidade do usuário.

O constante aumento da demanda por tecnologia fez com que GPUs de última geração sejam fabricadas em tecnologias com processo de fabricação de até 28nm, com frequência de relógio de até 1GHz. Esse aumento da frequência de relógio e densidade de transistores, combinados com a redução da tensão de operação, faz com que os transistores fiquem mais suscetíveis a falhas causadas por interferência de radiação. O modelo de programação utilizado pelas GPUs faz uso de constantes acessos a memórias e registradores, tornando estes dispositivos sensíveis a perturbações transientes em seus valores armazenados. Estas perturbações são denominadas *Single Event Upset* (SEU), ou *bit-flip*, e podem resultar em erros no resultado final da aplicação.

Este trabalho tem por objetivo apresentar um modelo de injeção de falhas transientes do tipo SEU nos principais bancos de registradores da GPGPU Flexgrip, avaliando o comportamento da execução de diferentes algoritmos em presença de SEUs. O impacto de diferentes distribuições de recursos computacionais da GPU em sua confiabilidade também é abordado. Resultados podem indicar maneiras eficientes de obter-se confiabilidade explorando diferentes configurações de GPUs.

Palavras-Chave: GPU, processamento paralelo, alta performance, tolerância a falhas.

Evaluation of Transient Fault Effect in the Register Files of Graphics Processing Units

ABSTRACT

Graphic Process Units (GPUs) are specialized massively parallel units that are widely used due to their high computing processing capability with respective lower costs. The ability to rapidly manipulate high amounts of memory simultaneously makes them suitable for solving computer-intensive problems, such as analysis of air traffic control, academic researches, image processing and others. General-Purpose Graphic Processing Units (GPGPUs) designates the use of GPUs in applications commonly handled by Central Processing Units (CPUs). The rapid proliferation of GPUs due to the advent of significant programming support has brought programmers to use such devices in safety critical applications, like automotive, space and medical. This crescent use of GPUs pushed developers to explore its parallel architecture and proposing new implementations of such devices. The FLEXible GRaphics Processor (FlexGrip) is an example of GPGPU optimized for Field Programmable Arrays (FPGAs) implementation, fully compatible with GPU's compiled programs.

The increasing demand for computational has pushed GPUs to be built in cutting-edge technology down to 28nm fabrication process for the latest NVIDIA devices with operating clock frequencies up to 1GHz. The increases in operating frequencies and transistor density combined with the reduction of voltage supplies have made transistors more susceptible to faults caused by radiation. The program model adopted by GPUs makes constant accesses to its memories and registers, making this device sensible to transient perturbations in its stored values. These perturbations are called Single Event Upset (SEU), or just bit-flip, and might cause the system to experience an error.

The main goal of this work is to study the behavior of the GPGPU FlexGrip under the presence of SEUs in a range of applications. The distribution of computational resources of the GPUs and its impact in the GPU confiability is also explored, as well as the characterization of the errors observed in the fault injection campaigns. Results can indicate efficient configurations of GPUs in order to avoid perturbations in the system under the presence of SEUs.

Keywords: GPU, parallel processing, high performance, fault tolerance

1 INTRODUÇÃO

Ao longo dos últimos anos, unidades gráficas de processamento, conhecidas como GPUs (*Graphics Processing Units*), tem despertado um grande interesse na área de computação de alto desempenho. O uso de uma arquitetura totalmente voltada para o paralelismo faz com que GPUs superem gradativamente o poder computacional de CPUs (*Central Processing Units*) para aplicações que satisfaçam os requisitos necessários de paralelismo de GPUs. Na figura 1.1, podemos observar a evolução do número de operações de ponto flutuante por segundo que tais dispositivos comportam. É notável o grande salto computacional atingido por GPUs, o que atrai o interesse de seu uso nos mais diversos ramos que necessitam alto processamento de dados.

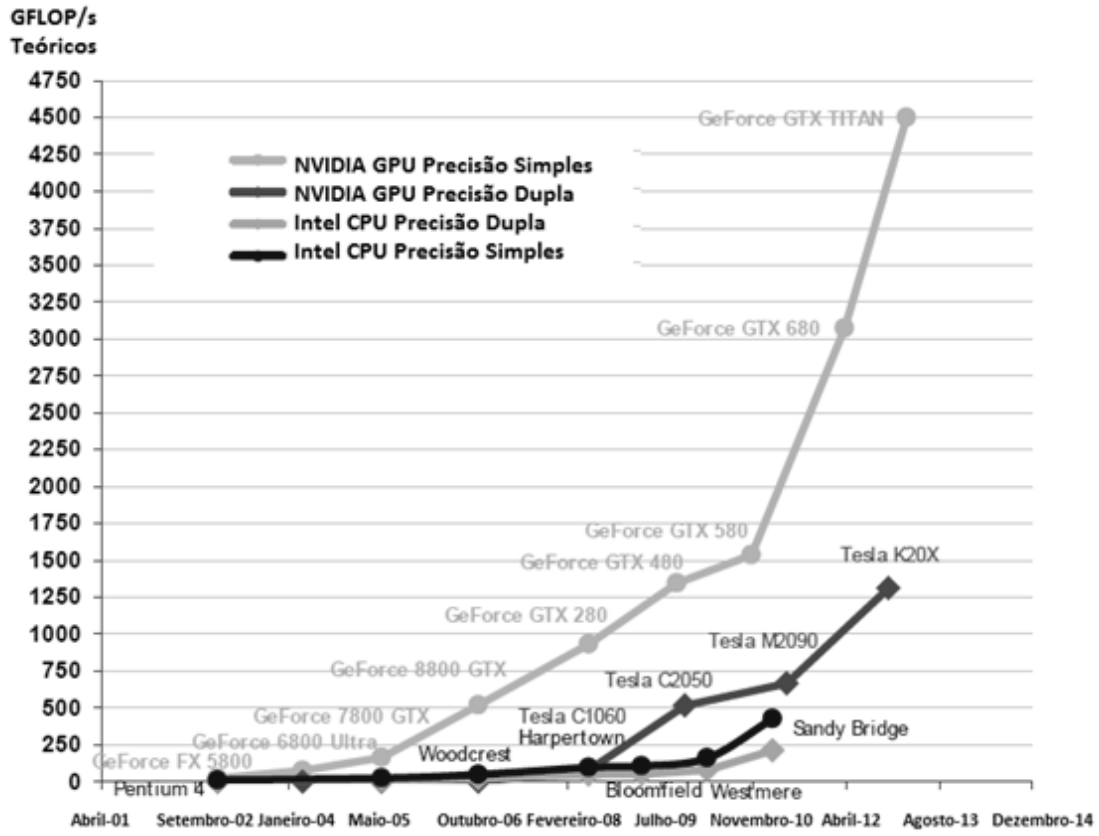

Figura 1.1: Computações por segundo - GPU vs. CPU (NVIDIA, 2015)

A arquitetura de GPUs foi inicialmente desenvolvida com o intuito de realizar processamento dedicado às robustas placas de vídeo da NVIDIA. Seu poder computacional logo chamou a atenção de desenvolvedores, que passaram a explorar a sua arquitetura altamente paralela nas mais diversas aplicações, dando origem ao termo *General Purpose Graphic Processing Unit*, ou GPGPU, que designa o uso de GPUs para algoritmos de propósito gerais. O aumento quase exponencial do poder computacional de GPUs ao longo dos últimos 10 anos dá-se pelo constante aprimoramento de sua arquitetura paralela. O modelo de programação *Compute Unified Device Architecture*, ou simplesmente CUDA, foi concebido a fim de facilitar a programação de GPUs através do uso de linguagens de programação de alto nível já conhecidas, tais como C, C++, Java, Python, dentre outros (NVIDIA, 2009). Neste modelo, o dispositivo hospedeiro é responsável por invocar o uso da GPU para computações dedicadas, dando total liberdade ao desenvolvedor para explorar livremente os recursos da GPU, balanceando-os da maneira que achar mais conveniente.

A constante evolução no modelo de programação de GPUs, combinado com sua eficiente execução de tarefas elementares em paralelo, torna esta arquitetura mais efetiva que CPUs convencionais para a computação de algoritmos onde grandes blocos de dados podem ser processados em paralelo. Exemplos de aplicações onde tais algoritmos podem ser encontrados são encontrados em análise de controle de tráfego aéreo, processamento de imagens médicas, álgebra linear, estatística, reconstrução 3D e determinação de preço de venda de ações no mercado financeiro (KRUGER, 2003).

A rápida proliferação no uso de GPUs, graças ao advento do modelo CUDA de programação, fez com que seu uso seja explorado também em aplicações que requerem alto grau de confiabilidade, tais como aplicações automotivas, espaciais e médicas (HU, 2011)(ZHANG, 2007)(STRANO et al, 2011). Nestas aplicações, um erro no comportamento do sistema pode ser fatal para o usuário final, tornando obrigatória a pesquisa de técnicas de detecção e correção de falhas, visto que estas aplicações necessitam manter seu funcionamento mesmo na presença de falhas.

A crescente demanda por poder computacional fez com que os últimos modelos de GPUs passassem a ser desenvolvidas em processos de fabricação de 28nm, com frequências de relógio de até 1GHz. O aumento na frequência de operação de sistemas digitais, combinado com os respectivos aumentos de densidade de transistor e redução de tensão de alimentação fez os transistores experimentarem cada vez mais falhas induzidas por radiação (BAUMANN, 2001). Estas falhas, causadas principalmente por partículas energizadas, fazem com que sistemas digitais sejam potencialmente sujeitos a experimentar erros induzidos por radiação (SLAYMAN, 2010). Perturbações que se manifestam em memórias do sistema, modificando o valor original de um de seus elementos de armazenamento, são denominadas *Single Event Upset* (SEU), ou bit-flip, e podem resultar em um erro no resultado final da aplicação.

Na tabela 1.1, podemos observar a evolução da taxa de SEUs em diferentes tecnologias de fabricação de circuitos integrados. Podemos observar que a taxa de SEUs por bit de memória em um milhão de horas (*SEU rate in FITs/kbit*) diminui com a evolução da tecnologia de fabricação. Por outro lado, novas tecnologias apresentam um aumento no uso de memórias SRAMs, com o intuito de aumentar a eficiência dos dispositivos. A tabela I demonstra também uma taxa crescente no número de SEUs não corrigidos por modernas técnicas de correção de erros em memórias, como ECC (*Error Correcting Code*), indicando que o uso de técnicas de detecção e correção de falhas

também são necessárias em outros componentes internos de processadores (DIXIT, 2011).

Tabela 1.1: Taxa de SEUs por tecnologia de microprocessadores (DIXIT, 2011)

Tecnologia (nm)	Taxa de SEUs relativos (FITs/kbit)	Mbits por processador (aprox.)	Taxa relativa de SEUs não corrigidos (kFIT)
250	3,2	1,52	5,0
180	3,0	1,52	4,3
130	2,4	3,28	7,9
90	1,0	33,6	33,6
65	0,7	44,3	30,5
40	0,94	71,0	67,0

Como mencionado anteriormente, GPUs modernas vem sendo desenvolvidas em tecnologias de última geração, sendo potencialmente sujeitas a experimentar erros induzidos por radiação mesmo em aplicações sendo executadas ao nível do mar (NORMAND, 1996), sendo GPUs também suscetíveis a tais perturbações, geradas principalmente pelo efeito de nêutrons atingindo o dispositivo (RECH, 2012). Experimentos conduzidos na rede de computação distribuída *Folding@home* (BEBERG, 2009) demonstraram que cerca de dois terços das GPUs utilizadas no experimento apresentaram suscetibilidade a experimentar erros em seus elementos lógicos e de memória (HAQUE, 2010).

Estas perturbações, que GPUs estão sujeitas a enfrentar, combinadas com os requisitos de confiabilidade exigidos por aplicações mencionadas anteriormente, tornam obrigatória a pesquisa do comportamento de GPUs na presença de falhas. Arquiteturas recentes de GPUs já fazem uso de técnicas de detecção e correção de erros em seus elementos de memória (NVIDIA, 2009). Dados corrompidos nestes componentes podem ser detectados e corrigidos pela presença de códigos de correção de erro, também conhecidos como ECC. Porém, elementos primordiais da arquitetura, como seus bancos de registradores, podem ter suas falhas facilmente mascaradas, pois ECCs tem a capacidade de corrigir dados que são corrompidos após sua escrita no registrador, não sendo capazes de corrigir dados que já sejam escritos com valores errados, provenientes da parte combinacional do sistema.

Este trabalho tem por objetivo avaliar o comportamento da GPGPU FlexGrip (*FLEXible GRaphIcs Processor*) (ANDRYC, 2013) descrita em linguagem de hardware (HDL), sintetizável em ASIC (*Application Specific Integrated Circuits*) ou circuito programável FPGA (*Field Programmable Array*), na presença de falhas transientes em seus principais bancos de registradores. Um modelo de injeção de falhas transientes é proposto a fim de simular estas falhas sem comprometer o fluxo de execução de cada algoritmo analisado, além de gerar resultados expressivos para a análise do comportamento de GPGPUs em presença de falhas. Diferentes algoritmos paralelos são

utilizados nas campanhas de injeção de falhas, sendo sua confiabilidade avaliada para distribuições distintas de recursos computacionais da GPGPU.

Embora muitos trabalhos tenham analisado o comportamento de GPGPUs sob efeito de radiação, como irá ser relatado capítulo de estado da arte, a injeção de falhas em uma GPGPU descrita em linguagem de hardware permite um maior controle do processo, além da maior facilidade na análise dos erros produzidos, permitindo assim um diagnóstico preciso dos componentes mais sensíveis do dispositivo. A facilidade na implementação deste método o credencia no teste de diferentes técnicas de detecção de erros por proteções em software.

O restante do texto está organizado da seguinte forma. No capítulo 2, são apresentados conceitos básicos sobre GPUs e confiabilidade de sistemas digitais com a finalidade de um melhor entendimento deste trabalho, além do estado da arte nesta área. O capítulo 3 apresenta conceitos sobre a arquitetura FlexGrip, enquanto o capítulo 4 demonstra o modelo de injeção de falhas adotado. No capítulo 5 são apresentados os resultados experimentais, e o capítulo 6 conclui este trabalho.

2 CONCEITOS BÁSICOS E TRABALHOS RELACIONADOS

Neste capítulo, serão apresentados conceitos básicos sobre GPUs, necessários para o entendimento do estado da arte sobre confiabilidade nestes dispositivos. Na primeira seção, é apresentado o modelo computacional utilizado por GPUs e como ele se relaciona com sua arquitetura paralela. Na sequência, uma breve abordagem sobre SEUs é apresentada, com o estado da arte em confiabilidade de GPUs sendo abordado na última seção.

2.1 Descrição da Arquitetura de uma GPU

Uma GPU é um dispositivo com múltiplos núcleos de processamento e uma substancial capacidade de processamento paralelo. A grande parte da área de uma GPU é dedicada às unidades de processamento de dados, sem a necessidade do uso de grandes mecanismos de controle, como pode ser observado na figura 2.1. Este modelo de arquitetura habilita GPUs a resolverem problemas com alto grau de computação de maneira muito mais eficiente do que processadores de propósito geral, ou GPPs (*General Purpose Processors*). Estes dispositivos são projetados para minimizar o tempo de execução de determinadas tarefas, atingindo esse objetivo através do uso de unidades de controle robustas, além de grandes memórias caches a fim de minimizar a latência de instruções e acessos a dados.

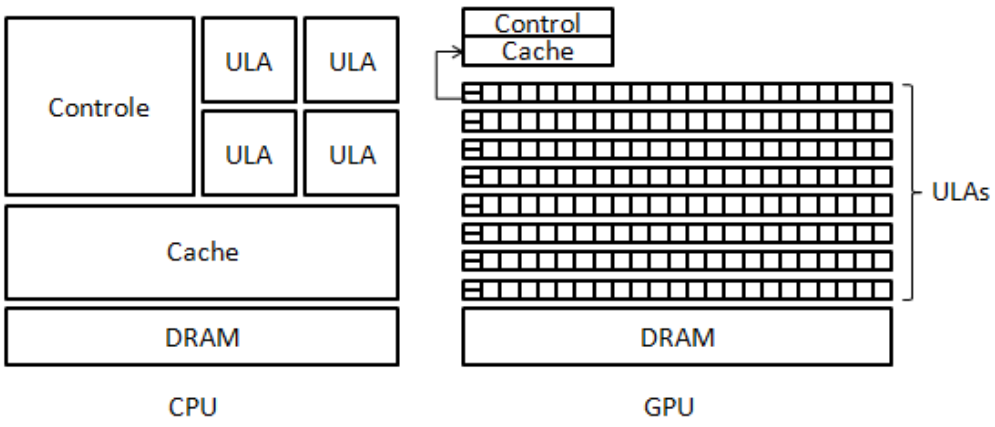

Figura 2.1: Representação simbólica das diferenças de distribuição de recursos entre GPUs e CPUs

A arquitetura G80 foi a primeira a contar com o modelo de programação CUDA (LINDHOLM, 2008) que introduziu o conceito de *Computing Capability* (CC), que diferencia modelos distintos de arquitetura de GPUs a fim de serem facilmente programáveis pelo mesmo modelo de programação. A GPGPU FlexGrip foi desenvolvida baseada nos recursos disponíveis desta arquitetura, implementando seu mesmo conjunto de instruções. Na próxima seção, serão abordados conceitos básicos do modelo de programação CUDA, relacionando-o com os aspectos gerais da arquitetura de GPUs. Esta breve explicação serve como guia para a apresentação dos trabalhos relacionados na área, preparando o leitor para os conceitos mais específicos que constituem a elaboração deste trabalho.

2.2 Modelo de programação CUDA

O modelo de programação CUDA foi concebido com o objetivo de facilitar a programação de GPUs através do uso de linguagens de programação de alto nível. O modelo de computação *Single Instruction Multiple Data* (SIMD) descreve um método de operação onde a mesma instrução é aplicada simultaneamente em diversos conjuntos de dados, proporcionando um alto grau de paralelismo (NVIDIA, 2011). A arquitetura de GPUs foi concebida a partir deste modelo, sendo constituída por um conjunto de multiprocessadores, denominados *Streaming Multiprocessors* (SM), que são responsáveis por realizar operações simultâneas em diferentes conjuntos de blocos de dados. Os multiprocessadores, por sua vez, são constituídos por um conjunto de processadores escalares denominados *Scalar Processors* (SPs), que realizam operações elementares sobre *threads*, que determinam a menor operação possível de ser realizada por uma GPU.

Um bloco de *threads* representa um conjunto de operações que pode ser executado em paralelo pela GPU. Blocos de *threads* são particionados em unidades lógicas menores, denominados *warps*, que representam o menor conjunto de operações simultâneas que são realizadas por um SM. Em outras palavras, um *warp* conterá um conjunto de informações necessárias para a sua execução paralela pelas unidades de processamento residentes no SM, e será escalonado em diferentes SPs que realizarão a execução de sua instrução paralelamente. Múltiplos *warps* podem ser atribuídos para um único SM e escalonados ao longo do tempo. Adicionalmente, blocos de *threads* são agrupados em *grids*, que representam a totalidade de operações a serem executadas na GPU. Denomina-se *kernel* a invocação da execução de programas em GPUs através de um dispositivo hospedeiro, que é responsável por mudar o contexto de execução para a GPU realizar suas computações.

Resumidamente, blocos de *threads* são escalonados ao longo das unidades de processamento da GPU da seguinte maneira:

1. Cada bloco de *threads* é atribuído a um SM pelo escalonador de blocos.
2. No bloco escalonado, o SM irá selecionar *warps* livres para execução.
3. No *warp,* cada *thread* é executada em um SP individual.

Na figura 2.2 podemos observar um apanhado geral de como blocos de *threads* e *warps* são distribuídos ao longo da arquitetura da GPU. Neste exemplo hipotético, a GPU é constituída por três multiprocessadores com oito processadores escalares cada.

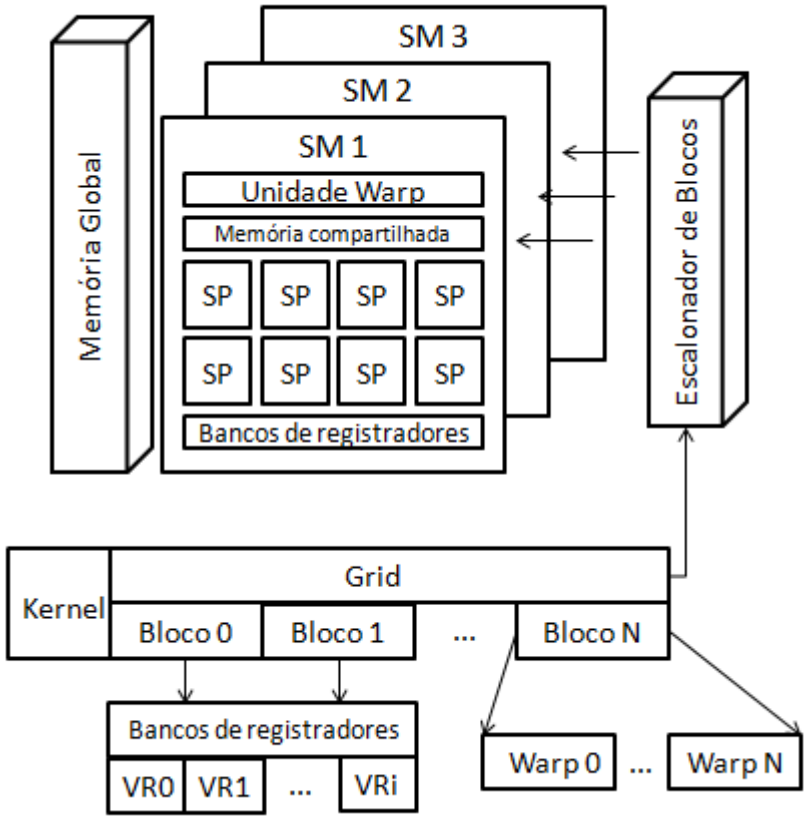

Figura 2.2: Representação da arquitetura de uma GPU

Os bancos de registradores da arquitetura são igualmente divididos entre os SPs de cada multiprocessador. Esta divisão permite que os processadores escalares utilizem o seu próprio conjunto de registradores para realizar suas computações, removendo assim qualquer dependência de dados possível entre operações realizadas por estes componentes. A memória compartilhada é visível por todas as *threads* residentes no bloco de *threads*, sendo igualmente particionada entre elas.A memória global é visível por todas as *threads* residentes no *grid*, servindo como meio de comunicação entre *threads* de diferentes SMs, caso necessário. A memória constante atua como uma memória cache para cada multiprocessador e é acessível por todas as *threads* residentes no bloco em execução. Unidades de funções especiais (SFUs) executam operações aritméticas especiais, tais como seno, cosseno, logaritmos, dentre outras.

Um exemplo de algoritmo de adição de matrizes descrito em C, para o modelo de programação CUDA, é apresentado na figura 2.3. O hospedeiro é responsável por invocar a execução da GPU através da chamada de funções marcadas com a diretiva *__global__*, que indica que a função será executada na GPU. Configurações de tamanho de bloco e número de blocos a serem utilizados pela GPU são passados como parâmetro, como podemos observar na instrução *matrixAdd<<<numBlocks, threadsPerBlock>>>*. Na instrução *threadsPerBlock* é possível observar que tais configurações tomam por base um bloco de dimensão 16x16, totalizando 256 *threads*, e que o número de blocos será determinado pela dimensão das matrizes de entrada.

```
__global__ void matrixAdd(float A[dim][dim], float B[dim][dim], float R[dim][dim])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i < dim && j < dim)
        R[i][j] = A[i][j] + B[i][j]
}

int main(void)
{
    // Code running in host
    ...
    // Kernel invocation to be run in the GPU
    dim3 threadsPerBlock(16,16)
    dim3 numBlocks(dim / threadsPerBlock.x, N / threadsPerBlock.y);
    matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, R);
}
```

Figura 2.3: Adição de matrizes em CUDA

Como dito anteriormente, *threads* designam operações elementares a serem executadas nos processadores escalares. Cada bloco de *threads* a ser executado por um multiprocessador terá um valor *x* e *y* associado indicando sua posição relativa dentro do *grid*. Adicionalmente, *threads* possuem um índice associado à sua posição dentro do bloco de *threads*. No algoritmo apresentado, blocos possuem dimensão 16x16, portanto cada bloco será responsável pela adição de 256 elementos das matrizes de entrada. A determinação de quais elementos das matrizes serão computados por cada bloco é dada pela atribuição das variáveis *i* e *j* no algoritmo apresentado. Cada bloco de *threads* selecionará uma matriz elementar quadrada de dimensão 16x16, tendo sua posição na matriz original determinada pela multiplicação dos índices *x* e *y* da posição do bloco no *grid* pelas dimensões *x* e *y* do bloco. Os índices *x* e *y* da *thread* indicarão exatamente qual elemento de cada matriz de entrada *A[]* e *B[]* será computado por esta *thread* no processador escalar designado para ela. Na figura 2.4 é demonstrada a indexação de blocos e *threads* para duas invocações do algoritmo de adição de matrizes em CUDA para matrizes de entrada de dimensão 64x64.

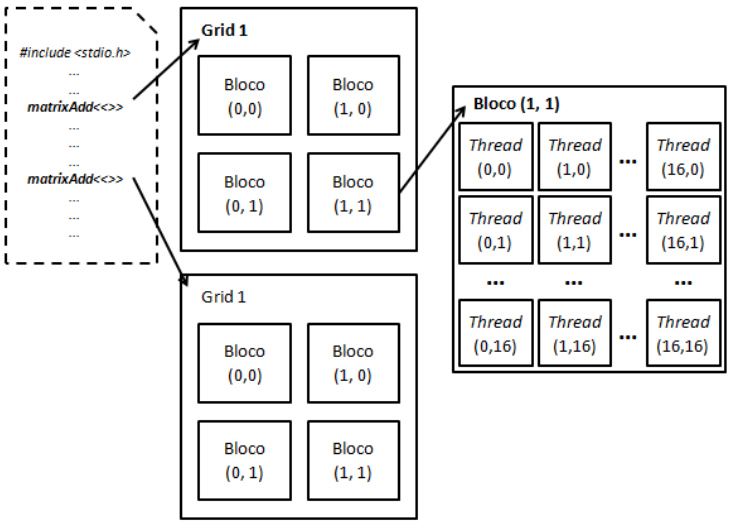

Figura 2.4: Distribuição de threads na GPU para matrizes de dimensão 64x64 aplicadas no algoritmo da figura 2.3

Na tabela 2.1 são apresentas as especificações da arquitetura G80.

Tabela 2.1: Especificações da GPU G80

Número máximo de *threads* residentes por multiprocessador	256
Número máximo de *warps* residentes por multiprocessador	24
Tamanho do *warp*	32
Número máximo de blocos residentes por multiprocessador	3
Número máximo de *threads* por SM	768
Dimensionalidade máxima de um bloco	3
Dimensionalidade máxima de um *grid*	2
Número de registradores de 32 bits por multiprocessador	8192
Quantidade máxima de memória compartilhada por multiprocessador	16KB
Número de bancos de memória compartilhada	1
Tamanho da memória constante	8KB

A seguir, apresentaremos conceitos básicos sobre confiabilidade em sistemas digitais, mais especificamente em GPUs, a fim de facilitar o entendimento sobre trabalhos relacionados nessa área.

2.3 Confiabilidade em GPUs

Efeitos não destrutivos são definidos como falhas transientes provocadas pela interação de uma partícula energizada com a junção PN do dreno do transistor *off-state*. Esse pulso pode afetar um nodo do circuito e assim gerar um pulso transiente de tensão que pode ser interpretado como um sinal interno e levar o circuito a produzir um resultado diferente do esperado (DODD, 2003).

Quando um pulso transiente ocorre em um elemento de memória, como por exemplo um registrador, este evento é classificado como *Single Event Upset* (SEU), ou simplesmente *bit-flip*. Perturbações que induzem um pulso de tensão em elementos combinacionais do circuito são chamados de *Single Event Transient* (SET). Perturbações transientes podem ser mascaradas pela aplicação ou se manifestarem até atingir um *flip-flop* do sistema, induzindo um SEU neste elemento.

A simulação dos efeitos destas perturbações em circuitos digitais é necessária a fim de determinar a confiabilidade de sistemas em presença de falhas. Estes efeitos podem ser facilmente simulados através da criação de modelos de injeção de falhas eficientes que obtenham uma boa cobertura da execução de aplicações previamente escolhidas para campanhas de injeção de falhas. Falhas transientes são facilmente simuladas a nível RTL (*Register Transfer Level*), ou seja, num nível onde o circuito é descrito em termos de fluxo dos dados, não deixando módulos de sistema fixos a componentes em específico.

Uma simples perturbação em um sinal intermediário do sistema pode ser interpretado como um SET, porém é muito difícil obter uma boa cobertura de injeção nos componentes ativos do sistema em arquiteturas complexas como GPUs, que são constituídas por milhões de sinais internos. Desta maneira, o estudo de SEUs é muito mais atrativo, afinal uma perturbação em um elemento de memória do sistema é muito mais suscetível a manifestar um erro no resultado final do que uma perturbação em um sinal aleatório do dispositivo. Porém, seu modelamento é muito mais complexo que um SET, visto que o processo de injeção de falhas deve ser o menos intrusivo o possível, a fim de não comprometer as características de execução dos algoritmos. Uma maneira de realizar tal tarefa é escolher pontos estratégicos da arquitetura para injetar perturbações que posteriormente possam ser capturadas por elementos de memória.

Uma GPU possui milhões de elementos de memória que são suscetíveis a SEUs. Neste trabalho, estamos interessados em analisar o efeito de SEU nos bancos de registradores. Na figura 2.5 apresentamos uma representação da arquitetura de GPUs, dando enfoque para os bancos de registradores onde serão injetadas falhas transientes. Vale lembrar que há milhares de outros elementos de memória presentes na GPU que não serão explorados neste trabalho, tais como elementos no circuito de controle, máquinas de estado, *datapath* e memórias embarcadas.

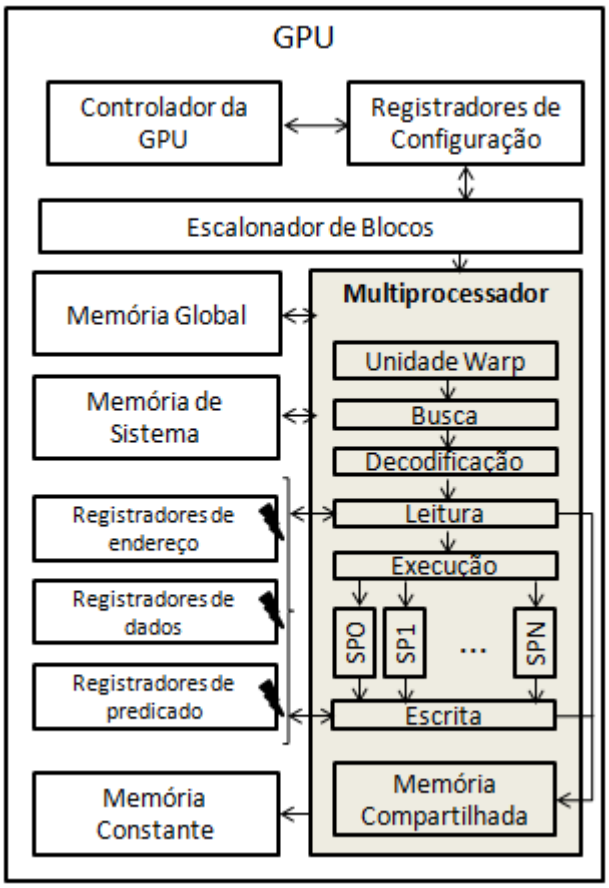

Figura 2.5: Banco de registradores da GPU suscetíveis SEU

Neste trabalho serão realizadas campanhas de injeção de SEUs nos diferentes bancos de registradores da GPGPU FlexGrip, posteriormente apresentada na sessão 3, produzindo resultados que servirão de guia para futuros estudos de técnicas de tolerância a falhas em registradores de GPUs. Com estes resultados, pretendemos demonstrar as regiões mais sensíveis da GPU, além de qualificar os erros através de analise dos resultados de injeção de falhas produzidos. Na próxima seção, serão brevemente discutidos trabalhos relacionados na área, apresentado o estado da arte em confiabilidade de GPUs.

2.4 Estado da Arte

Existem diversos trabalhos no campo de injeção de falhas em GPUs. Experimentos com aceleradores de partículas são bastante comuns na literatura, verificando o comportamento de GPUs comerciais na presença de falhas induzidas pelo bombardeamento direcionado destas partículas no dispositivo em teste. Adicionalmente, modelos de injeção de falhas por simulação também são propostos, porém a intrusividade de cada modelo permanece como um problema a ser contornado. Por fim, simuladores de GPUs também são utilizados a fim de avaliar a confiabilidade deste dispositivo em presença de falhas, porém tais metodologias podem não descrever o real comportamento de uma GPU real. Nas próximas seções, diferentes áreas de pesquisa

em confiabilidade de GPUs são apresentadas, com breve descrição dos resultados obtidos.

2.4.1 Taxas de *Failure in Time* em GPUs

Nestes experimentos, aceleradores de partículas são utilizados a fim de verificar o comportamento de diferentes componentes internos da GPU na presença de falhas induzidas por radiação com neutrons. Rech 2012 apresenta resultados iniciais sobre a probabilidade de falha por tempo, ou FIT (*Failure in Time*) na memória compartilhada e registradores internos de GPUs comerciais. Diferentes padrões de dados são armazenados nestes dispositivos e depois lidos para uma memória DDR, não afetada pela radiação, onde são comparados com os valores esperados. Na figura 2.6, apresentamos os resultados para SEUs em registradores internos da GPU, indicando a suscetibilidade destes dispositivos a tais falhas.

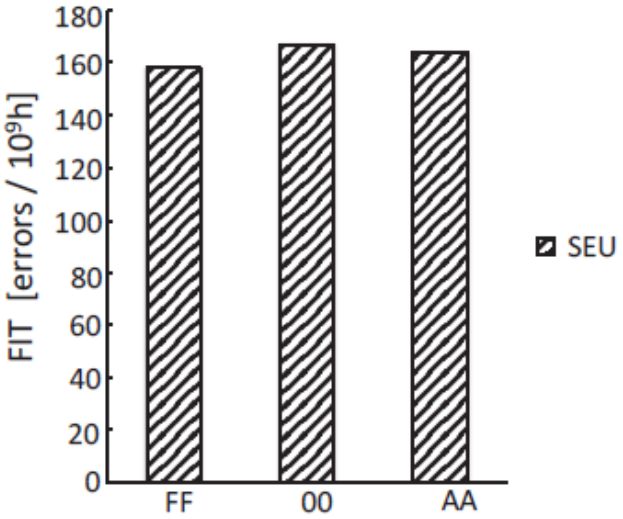

Figura 2.6: Taxa FIT causada por SEUs em registradores internos na cidade de Nova Iorque (RECH, 2012)

Com metodologia de injeção de falhas semelhante, Rech 2013 avalia a suscetibilidade a falhas de GPUs operando sobre diferentes representações de dados. Os resultados demonstram que, sob efeitos de radiação, operações em dados de ponto flutuante são a melhor opção, visto que apresentam maior confiabilidade que operações em dados inteiros. Este resultado é explicado pelos autores pela menor complexidade necessária para realizar operações de multiplicação nesta representação, onde a mantissa dos números necessita somente de uma simples operação de soma, diminuindo a complexidade desta instrução. Resultados para soma em *doubles* apresentam uma sensibilidade muito maior devido ao maior intervalo de representação compreendido, necessitando de mais operações para normalizar os resultados nos expoentes. Na figura 2.7 são apresentados resultados de FIT para instruções de soma e multiplicação operando em dados com diferentes representações numéricas.

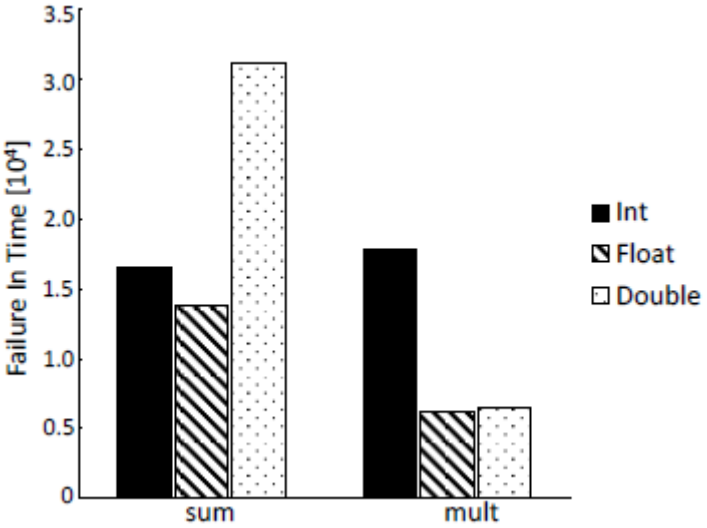

Figura 2.7: FIT para instruções de soma e multiplicação em diferentes representações numéricas (RECH, 2013)

2.4.2 Efeito de distribuição de *threads* na confiabilidade de GPUs

Ainda no âmbito de injeção de falhas por bombardeamento de nêutrons, alguns trabalhos apresentam uma estimativa da confiabilidade de GPUs adotando-se diferentes configurações de distribuição de recursos computacionais para determinados algoritmos. Rech 2013 apresenta um estudo sobre o impacto do número de *threads* utilizados na execução de instruções de soma e multiplicação na confiabilidade de GPUs. Resultados experimentais, observados na figura 2.8, demonstram que o aumento no número de *threads* responsáveis pela computação destes algoritmos resulta em uma maior probabilidade da GPU experimentar falhas induzidas por radiação. Em contrapartida, tais configurações exploram de maneira mais eficiente os recursos computacionais da GPU, sendo necessário determinar um ponto ótimo entre eficiência e confiabilidade nos resultados produzidos.

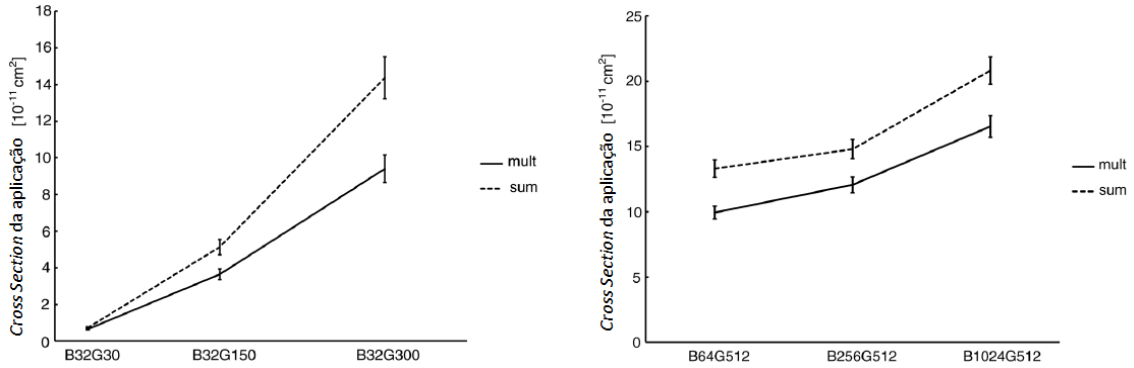

Figura 2.8: Cross section de operações de soma e multiplicação. À esquerda, é mantido fixo o número de threads por bloco. À direita, é mantido fixo o número de blocos por grid (RECH, 2013)

O aumento do número de *threads* foi explorado através da variação do valor do tamanho do *grid*, mantendo-se fixo o tamanho do bloco, e através da variação do valor do tamanho do bloco, mantendo-se fixo o tamanho do *grid*. Em ambos os casos, a confiabilidade da GPU diminui, a partir da medida do *cross-section* de cada caso. As diferentes configurações escolhidas irão impactar diretamente no escalonamento dos recursos internos da GPU, sendo a primeira responsável pelo aumento da carga no escalonador de blocos e a segunda responsável pelo aumento da carga no escalonador de *warps*. Esta diferença induz resultados interessantes com relação à eficiência na execução dos algoritmos para cada configuração. O aumento no número de blocos irá impactar em um maior número de mudanças de contexto para cada novo bloco que é encaminhado para o SM, sendo necessária a verificação de estado do bloco corrente, leitura de resultados, escalonamento de blocos pendentes, busca de dados para o SM e finalmente a sincronização de cada nova instrução. Estas tarefas adicionais geram um impacto muito maior no tempo de execução dos algoritmos do que o aumento no tamanho dos blocos de *threads*, onde o escalonamento interno das *threads* no SM é feito de maneira muito mais eficiente, resultando em tempos de execução consideravelmente menores para estes casos. Resultados ainda demonstram que o aumento na taxa de erros para estes casos é compensado pela maior eficiência obtida, como é demonstrado na figura 2.9, onde os resultados de *cross-section* são normalizados em termos de volume de dados processado e carga de processamento.

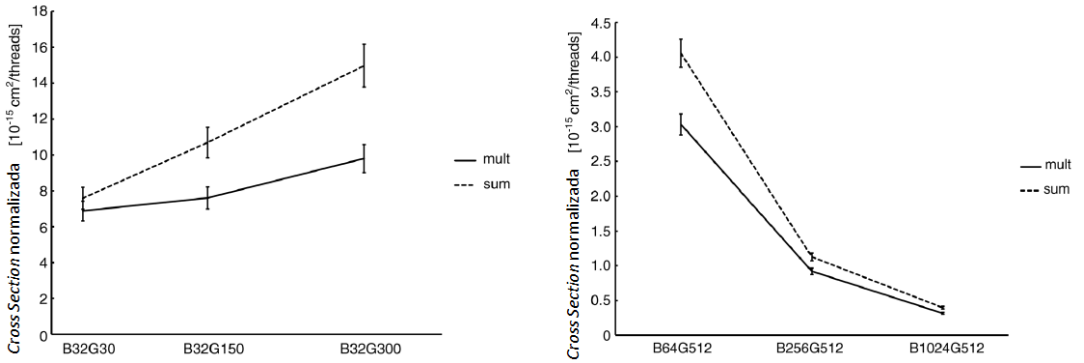

Figura 2.9: Cross section normalizada de operações de soma e multiplicação. À esquerda, é mantido fixo o número de threads por bloco. À direita, é mantido fixo o número de blocos por grid. (RECH, 2013)

Estes resultados são estendidos em Rech, 2014, onde resultados obtidos no trabalho anterior guiaram os autores a expandirem o estudo de diferentes distribuições de *threads* na análise do impacto de confiabilidade. O conceito de DOP (*Degree of Parallelism*) é explorado nas diferentes configurações, sendo obtido pelo contínuo aumento do número de *grids* com tamanho de bloco fixo até o ponto em que o tamanho do bloco começa a ser reduzido, a fim de explorar o impacto da diminuição do DOP na confiabilidade da GPU. Esta diminuição no DOP é aplicada no algoritmo de multiplicação de matrizes, apresentando também resultados em termos de MTBF (*Mean Time Between Failure*), que é definido como o tempo médio necessário para que ocorram duas falhas induzidas por radiação em um algoritmo sendo executado continuamente em uma GPU, e MEBF (*Mean Executions Between Failure*), que corresponde ao número de execuções completas entre duas corrupções de dados induzidos por radiação, como pode ser

observado na tabela 2.2. Resultados experimentais demonstram que, no caso do algoritmo de multiplicação de matrizes, a configuração B128 apresenta-se como a melhor opção em termos de confiabilidade, sendo capaz de combinar reduzidas *cross section* e tempo de execução. É interessante notar que, para este algoritmo, a distribuição ótima P apresenta resultados muito próximos do resultado ótimo de B128.

Tabela 2.2: MTBF e MEBF para diferentes distribuições de threads no algoritmo de multiplicação de matrizes (RECH, 2014)

Distribuição	MTBF [10^4h]	MEBF [10^7exec.]
MxM-G1024	2,37	7,87±1,10
MxM-G4096	2,46	9,84±1,38
MxM-G8192	2,73	13,91±1,95
MxM-P	2,89	15,92±2,23
MxM-B128	3,19	16,80±2,35
MxM-B64	3,19	10,44±1,46
MxM-B16	4,96	7,59±1,06

Resultados semelhantes são encontrados em Rech 2014, onde esta análise é estendida para aplicações HPC (*High Performance Computing*), sendo o algoritmo de transformada de Fourier também avaliado sob as mesmas condições.

2.4.3 Modelos de injeção de falhas

No âmbito de modelos de simulação de falhas por injeção, Fang, 2014 apresenta o GPU-Qin, que introduz uma metodologia de avaliação da resiliência de erros em aplicações pra GPUs. O modelo descrito faz uso da ferramenta de depuração *cuda-gdb*, que é capaz de interromper a execução do programa CUDA na GPU e fazer uso deste breve intervalo de tempo para realizar a injeção da falha. Após agrupar as *threads* da aplicação por similaridade e eleger uma de cada grupo para ser objeto de injeção, o injetor interrompe a execução no exato *breakpoint* escolhido para a *thread*, escolhe uma das instruções desta *thread* e então realiza a injeção da falha. Um ponto positivo desta implementação é que ela é capaz de simular falhas diretamente em GPUs, atingindo um alto grau de representatividade de dados visto que o tempo de execução de cada rodada de injeção é extremamente rápido. Por outro lado, o injetor necessita interromper a execução do algoritmo para injetar a falha, não representando o exato comportamento do dispositivo. Além disso, o método de injeção de falhas baseia-se na modificação da instrução a fim de injetar *bit-flips* em elementos de memória da GPU, o que também altera o algoritmo original utilizado para os testes.

O comportamento de GPUs em presença de falhas em seus registradores é abordado em Tselonis, 2013, porém os resultados apresentados são baseados em experimentos conduzidos no simulador GPGPU-Sim que, apesar de apresentar uma boa cobertura de

instruções de GPUs reais, ainda representa uma simulação do comportamento de GPUs, comprometendo uma análise real dos dados apresentados.

2.4.4 Detecção e correção de falhas em GPUs

A pouca liberdade na exploração dos recursos internos de GPUs, aliado com a perda de desempenho que técnicas de fácil implementação podem causar, fazem com que a área de detecção e correção de falhas em GPUs tenha ainda poucos trabalhos no estado da arte. Di Carlo, 2013, apresenta a metodologia SBST (*Software Based-Self-Test methodology*) aplicada em GPUs, com o intuito de localizar falhas em hardware nestes dispositivos. Esta metodologia baseia-se na definição de diferentes *kernels* de teste, a serem executados na GPU, que criam assinaturas de execução para os componentes internos de cada SM. As SFUs, CUDA cores e o distribuidor de *threads* são avaliados por esta metodologia, tendo suas assinaturas posteriormente checadas por processos monitores que, identificando uma assinatura diferente da assinatura livre de falhas, irá identificar o SM com faltoso.

Di Carlo, 2013, faz uso desta técnica a fim de detectar e corrigir falhas transientes em GPUs. O método descrito anteriormente é executado periodicamente na GPU a fim de encontrar SMs defeituosos, para então evitar a execução de blocos de *threads* nesses componentes. Dois métodos são apresentados neste trabalho. O primeiro, denominado *sleep*, coloca o SM faltoso em modo dormente, evitando assim a execução de um bloco de *threads* por um componente comprometido. O segundo método, denominado *wait*, coloca o bloco de *threads* em estado dormente, aguardando por um SM livre de falhas terminar a sua execução. Resultados experimentais podem ser observados na figura 2.10, onde nota-se o aumento gradativo no tempo de execução do algoritmo com relação ao número de SMs com presença de falhas, o que demonstra a limitação deste método em presença de múltiplas falhas nos SMs da GPU.

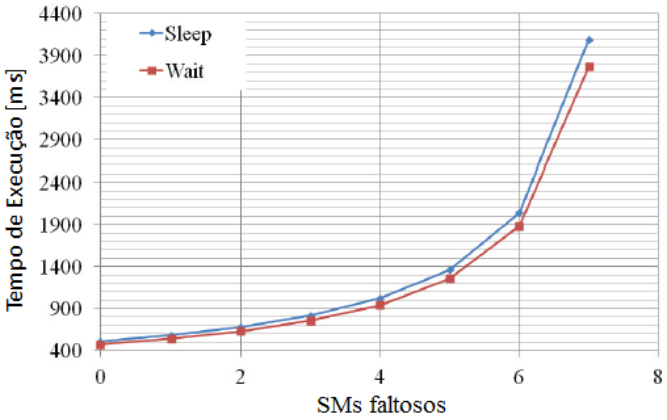

Figura 2.10: Resultados para multiplicação de matrizes (DI CARLO, 2013)

Finalmente, em Rech 2013, três conhecidas técnicas de proteção em software são avaliadas em GPUs: TMR (*Triple Modular Redundancy*), ABFT (*Algorithm Based*

Fault Tolerance) e extABFT, que é uma extensão da técnica ABFT proposta pelos autores. A metodologia de cada técnica não será discutida, visto que não abrangem o escopo deste trabalho. Entretanto, resultados são apresentados na figura 2.11, demonstrando o *overhead* em tempo de computação para cada método, indicando uma eficiência considerável no método proposto.

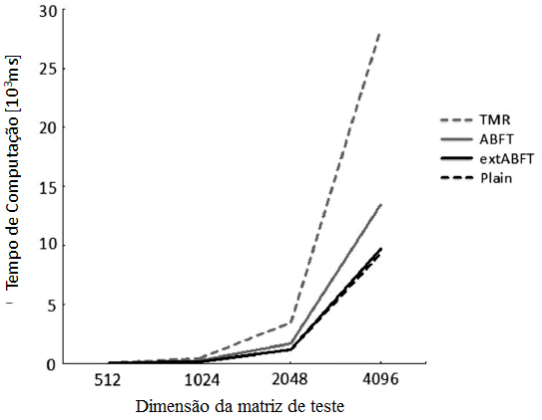

Figura 2.11: Overhead em tempo de computação para cada método proposto

3 ARQUITETURA FLEXGRIP

Neste capítulo iremos apresentar um panorama geral da arquitetura da GPGPU FlexGrip, objeto de estudo deste trabalho. O caminho de execução, em *hardware*, de um *kernel* é apresentado, seguido por uma descrição detalhada dos diferentes componentes presentes no sistema. No final da seção são apresentadas as instruções CUDA que foram implementadas para a arquitetura FlexGrip, bem como uma breve descrição da funcionalidade de cada uma.

3.1 Arquitetura FlexGrip

A GPGPU FlexGrip foi desenvolvida baseada na arquitetura da GPU G80 da NVIDIA, com capacidade computacional 1.0. A arquitetura foi implementada HDL para uma plataforma ML605 com FPGA Virtex-6. A GPGPU FlexGrip é utilizada, no nosso ambiente de testes, em conjunto com um microprocessador MicroBlaze (XILINX, 2008). A comunicação entre o MicroBlaze e a GPGPU é feita através de um barramento AXI, como pode ser visto na figura 3.1.

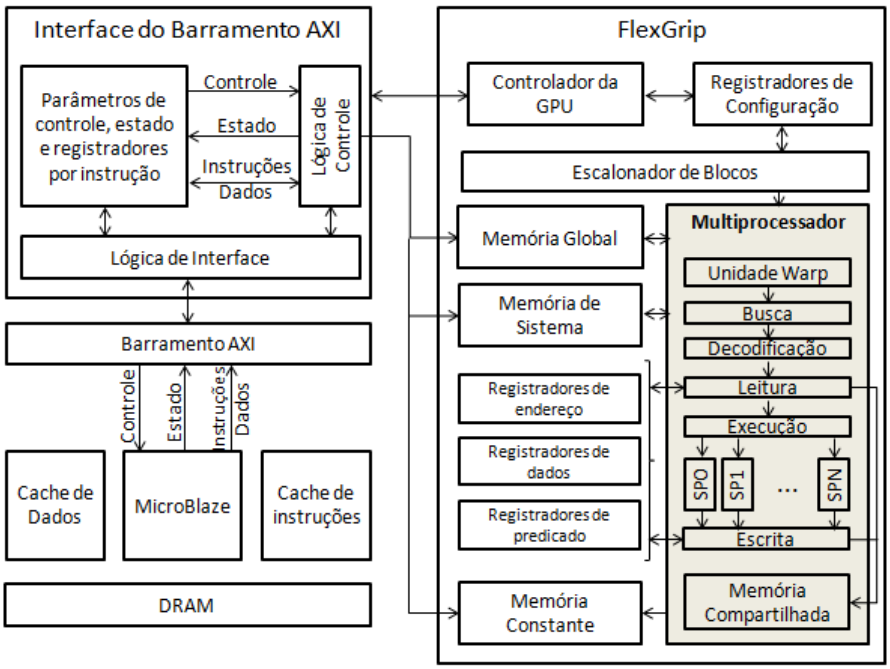

Figura 3.1: Arquitetura FlexGrip

Durante a execução do *kernel*, o processador MicroBlaze carrega um *driver* que envia informações de controle, estado e dados através do barramento. A lógica de controle atua como uma interface entre o barramento AXI e a GPGPU FlexGrip, executando funções que dependem dos valores escritos no registrador de controle. Ao ser carregado pelo processador, o *driver* carrega instruções CUDA e dados de programa nas memórias de sistema e global, respectivamente. Adicionalmente, o *driver* também atualiza o controlador da GPU com parâmetros associados ao *kernel* CUDA, como dimensões de bloco e *grid*, número de *threads* por bloco, número de registradores utilizados por *thread* e tamanho da memória compartilhada. Todos esses parâmetros são salvos nos registradores de configuração da GPGPU. Após esta inicialização, o controle de fluxo é passado para a GPGPU com a finalidade de executar o *kernel* CUDA.

A arquitetura FlexGrip segue o modelo SIMT, no qual uma instrução é lida e mapeada simultaneamente em múltiplos processadores escalares. O escalonador de blocos é responsável por escalonar blocos de *threads* através do método *round-robin*. Após encaminhar o bloco para o respectivo SM, o escalonador sinaliza a unidade *warp* a fim de iniciar o escalonamento dos *warps*. O número máximo de blocos que podem ser escalonados para um SM é restringido pela quantidade de memória compartilhada, e de registradores, disponível.

Nos próximos subcapítulos, serão discutidos elementos essenciais da arquitetura FlexGrip. Iniciaremos pelo multiprocessador, indicando todos os elementos de sua arquitetura e o fluxo de uma instrução CUDA pelo *pipeline* do multiprocessador. Por fim, apresentaremos o conjunto de instruções disponíveis na arquitetura FlexGrip.

3.2 Multiprocessador (SM)

O multiprocessador FlexGrip é composto basicamente por uma unidade *warp* e por um *pipeline* de cinco estágios: busca, decodificação, leitura, execução e escrita. Na figura 3.2, podemos observar o diagrama de blocos de um SM da arquitetura FlexGrip.

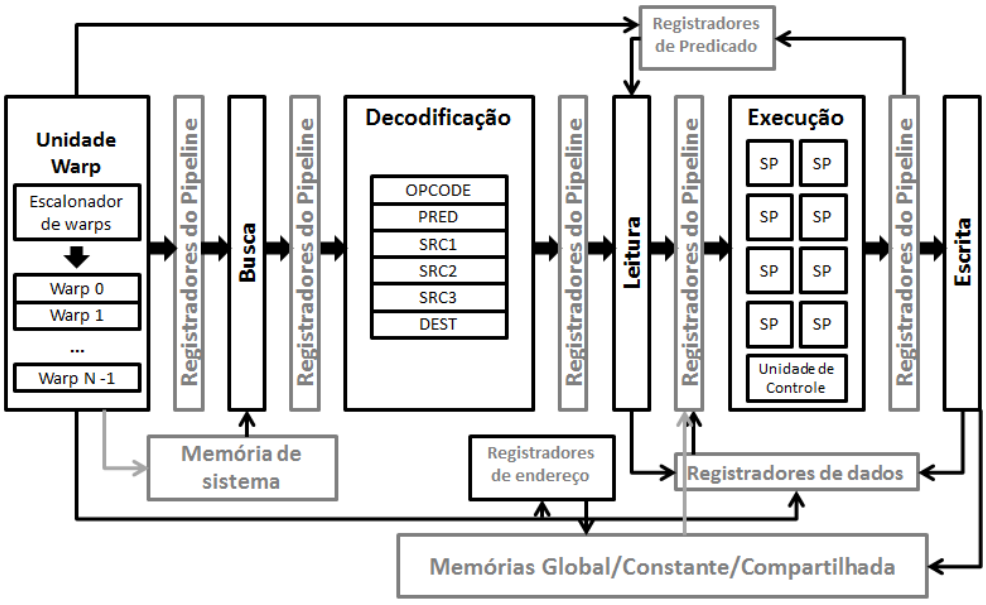

Figura 3.2: Multiprocessador da arquitetura FlexGrip

Nas próximas seções, os componentes básicos do SM serão discutidos individualmente.

3.2.1 Unidade *warp*

Como mencionado anteriormente, *threads* são agrupadas, por similaridade de operações, em grupos chamados *warps.* A unidade *warp* é responsável pela geração destes *warps* e escaloná-los através do método *round-robin. Warps* são constituídos por um estado e dados associados. Os dados do *warp* são constituídos por um identificador, o *program counter* (PC), e uma máscara de *threads.* A máscara de *threads* é tem a finalidade de mascarar *threads* que não façam parte do caminho corrente de execução da instrução CUDA. Cada *warp* possui seu próprio PC e, assim, é livre para seguir o seu próprio caminho de execução.

A sincronização de *warps* é feita explicitamente por uma função CUDA, chamada *__syncthreads.* Ao atingir essa instrução, *warps* têm de esperar a execução de outros *warps* e são assim marcados com o estado *Waiting*. Quando todos os *warps* terminam de executar o *kernel*, eles são marcados com o estado *Finished*. *Warps* ativos no *pipeline* são marcados como *Active*, enquanto que *warps* disponíveis para execução são marcados como *Idle*.

Dentro de um *warp*, *threads* são agrupadas em linhas, que dependem do número de processadores escalares instanciados no SM. A arquitetura FlexGrip permite que SMs sejam configurados com 8, 16 ou 32 SPs, a fim de minimizar o consumo de potência da GPU. Em uma configuração de 8 SPs, por exemplo, um *warp* é agrupado em quatro linhas contendo 8 *threads* cada. Analogamente, para uma configuração de 32 SPs, o *warp* será agrupado em 1 linha contendo 32 *threads* cada, atingindo assim o maior paralelismo possível.

Na arquitetura FlexGrip, os dados e estados dos *warps* são salvos em uma *block RAM* de duas portas e são indexados através do seu identificador de *warp.* Todos os *warps* são armazenados incialmente com o PC apontando para a primeira instrução, ou seja 0x00000000, e a máscara de instruções com todas as *threads* ativas, ou seja, com o valor 0xFFFFFFFF. Todos os *warps* são incializados com o estado *idle.* Assim que a geração dos *warps* é concluída, os dados do primeiro *warp* são lidos, seu estado é verificado como *idle* e todas as suas linhas são escalonadas, uma após a outra, para serem executadas nos processadores escalares. Da mesma maneira, novos *warps* são escalonados a cada ciclo de relógio através do escalonador de *warps.* A cada vez que a execução do *warp* é concluída pelo *pipeline* do SM, seus dados associados são atualizados e ele fica livre para ser executado novamente por um SM livre.

Este fluxo de execução é interrompido no caso de uma instrução de barreira de sincronização. Um *warp* que esteja executando essa instrução é marcado como *waiting* ao atingir o fim do *pipeline.* O registrador *fence register* é utilizado com a finalidade de registrar *warps* que estejam nesse estado. Na figura 3.3 é apresentada funcionalidade do *fence register* a fim de sincronizar a execução dos *warps* em estado *waiting*. A largura do *fence register* é determinada pelo número total de *warps* contidos no bloco. Cada *warp* com o estado *waiting* é registrado no *fence register* indexado pelo seu *warp ID.* Após esta etapa, o registrador é lido novamente para verificar se os *warps* restantes já

foram marcados com o estado *waiting*. Em caso afirmativo, todos os *warps* são sincronizados, ou seja, marcados como *idle*, e a barreira é liberada na unidade *warp*.

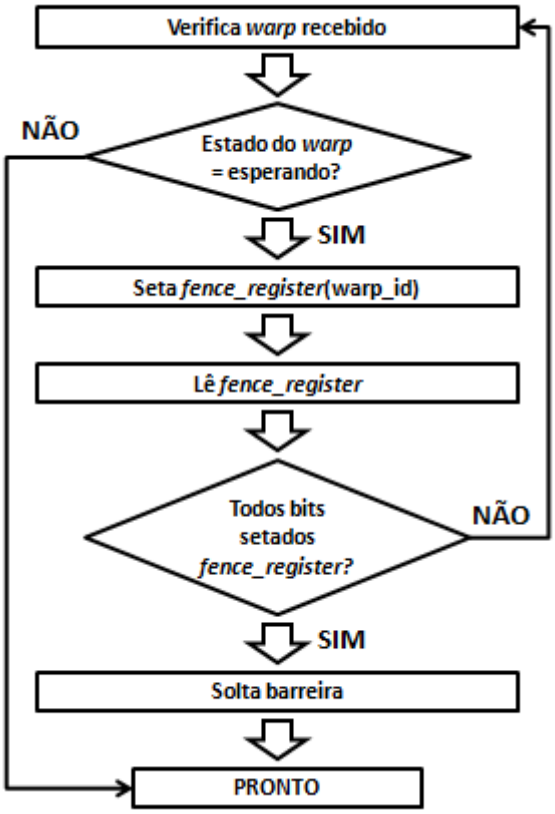

Figura 3.3: Sincronização de warps pela unidade warp

3.2.2 Estágio de busca e decodificação

O estágio de busca é responsável por realizar a busca da instrução binária referenciada pelo *Program Counter* do *warp* associado. A instrução é então registrada para todas as *threads* residentes no *warp*, no estilo *Single Instruction Multiple Data* (SIMD). Instruções CUDA podem ser de 4 bytes (*short*) ou de 8 bytes (*long*). Após realizar a busca da instrução, o PC é incrementado, por 4 ou 8 bytes, apontando assim para a próxima instrução a ser executada. O estágio de decodificação é responsável por interpretar a instrução e gerar diversos dados de saída, ou *tokens*, tais como: tipo de instrução, tamanho de instrução, operandos de origem e destino, tipos de dados, entre outros.

3.2.3 Estágio de leitura

No estágio de leitura, operandos de origem são lidos de bancos de registradores ou de memórias, dependendo do tipo de instrução decodificada. O banco de registradores é particionado de uma maneira que cada *thread* possua o seu próprio conjunto de registradores. O mapeamento do banco de registradores entre as *threads* do *warp*, para uma arquitetura FlexGrip implementada com 8 SPs é demonstrado na figura 3.4.

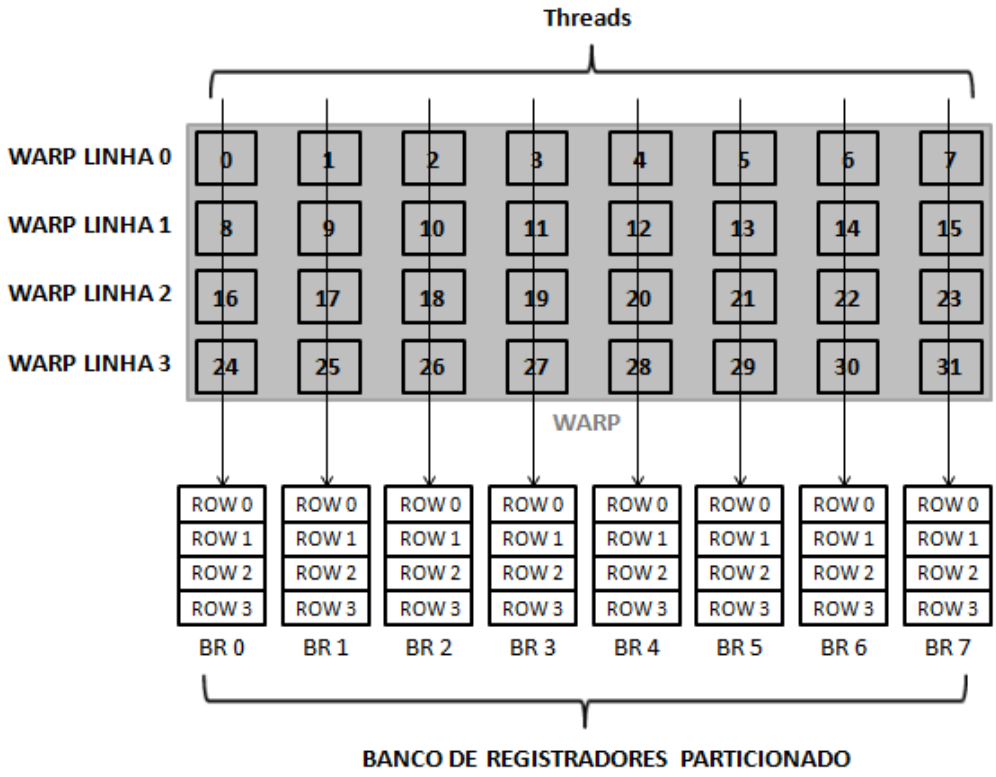

Figura 3.4: Distribuição de registradores entre threads do warp

Cada *thread* pertencente a uma mesma linha do *warp* é mapeado para um banco de registradores diferentes. A diferenciação entre *threads* que pertencem à uma mesma coluna é feita através do particionamento de cada banco de registradores por quatro, o que corresponde ao número de linhas por *warp* configurado. Cada banco é implementado como uma memória de duas portas, uma para leitura e uma para escrita, que é indexada pelo identificador de linha do *warp*. Desta maneira a leitura paralela dos registradores é garantida para todas as *threads* residentes no *warp*, independentemente da configuração de número de SPs por SM escolhida.

A arquitetura FlexGrip conta com três bancos de registradores principais: banco de registradores de dados, de endereço e de predicado. O banco de registradores de dado é utilizado para guardar dados intermediários para a execução das instruções. O banco de registradores de endereço guarda os *offsets* de memória para operações *gatter-scatter* em memória, que nada mais são que operações de leitura e escrita, porém os dados são lidos em blocos ao invés de sequencialmente. Os registradores de endereço são divididos em grupos de quatro, um para cada *thread*. O banco de registradores de predicado guarda sinalizações de predicado, ou *flags*, que são utilizadas em instruções

que geram execuções condicionais. Estas *flags* podem conter as mais diversas condições, como zero, não zero, sinal, estouro, vai um, dentre outras.

As memórias compartilhada, constante e global são implementadas com BRAMs de duas portas, com uma porta para o estágio de leitura e outra para o estágio de escrita do *pipeline*. Isso garante que esses estágios podem acessar as memórias do sistema simultaneamente. A memória compartilhada é dividida entre diferentes blocos de um SM e tem 16 KB de tamanho total. A memória constante, de leitura somente, possui 8 KB de memória utilizados para gravar dados constantes, atuando como uma cache do sistema. A memória global é responsável por armazenar dados de entrada e saída e tem um tamanho total de 256 KB. Controladores de memória encontram-se embutidos nos estágios de leitura e escrita, com a presença de unidades de cálculo de endereço dedicadas para acessar rapidamente os dados de memórias.

3.2.4 Estágio de controle e execução

Este estágio é responsável por realizar todo o processamento de dados, aritméticos e lógicos, com a ajuda de processadores escalares (SPs). Cada *thread* contida em um *warp* é mapeada para um SP, possibilitando assim uma execução paralela do algoritmo. Na arquitetura FlexGrip, como mencionado anteriormente, o número de SPs pode ser configurado a fim de se consumir mais ou menos potência. As configurações disponíveis, atualmente, são 8, 16 ou 32 SPs. Atualmente, os SPs tem suporte apenas para operações inteiras, tais como adição, subtração, multiplicação, multiplicação e adição, conversão de tipos de dados, deslocamento de bits e operações lógicas como AND, OR, NOR, XOR, dentre outras.

Na tabela 3.1 podemos ver um resumo das instruções interpretadas pela unidade de controle da GPGPU.

Tabela 3.1: Instruções suportadas pela unidade de controle

Instrução	Descrição
Branch	Caminhos diferentes de execução, condicional ou não.
Return	Final da execução do *kernel*
Set	Marca ponto de convergência de uma ramificação de execução
Barrier	Marca ponto de sincronização de *threads* no SM

A instrução de barreira já foi previamente discutida e indica um ponto de convergência para os diferentes *warps* de um bloco em execução no SM. Outra instrução de sincronismo, o *set*, é utilizada antes de instruções de *branch* potencialmente divergentes. Um exemplo de *branch* divergente pode ser observado na figura 3.5. Dependendo do valor do índice *x* da *thread*, que corresponde ao termo *threadIdx.x*, diferentes caminhos de execução podem ser tomados pela *thread*, resultando em operações de *store* em *offsets* distintos da memória global.

```
__global__ void teste(float *global_mem) {

    /* CUDA code */

    if ( threadIdx.x < 20 )
        global_mem[threadIdx.x] = /* something */;
    else
        global_mem[20 - threadIdx.x/20] = /* something */;

    /* CUDA code */
}
```

Figura 3.5: Exemplo de código CUDA com a presença de threads divergentes

Um *warp* é denominado divergente se o resultado final do *branch* não é o mesmo para todas as *threads* contidas no *warp*. A instrução *set* marca um ponto inicial onde as *threads* começam a divergir o caminho de execução. Cada caminho é executado por um conjunto diferente de *threads*, cada conjunto por vez, a partir do ponto inicial marcado pela instrução *set*. Ao concluírem suas execuções, *threads* divergentes passam a apontar seu PC para o ponto de reconvergência, retomando assim o paralelismo completo do SM.

3.2.5 Estágio de escrita

O estágio de escrita preenche o banco de registradores de dados com dados temporários, o banco de registradores de endereços com *offsets* de memória, o banco de registradores de predicado com sinalizações de predicado, a memória compartilhada com dados temporários e/ou resultados e a memória global com resultados finais. A sequência de operações para realizar a escrita em memória e em registradores é exatamente o oposto das operações do estágio de leitura. Ao terminar a sua execução no estágio de escrita, o *warp* retorna para a pilha, atualizando seus dados associados e seu estado.

Todos os estágios do *pipeline* tem um sinal de saída denominado *stall* que alimenta o estágio anterior. O sinal *stall* indica que o estágio está ocupado e não está apto a receber novos dados. Todo o estágio do *pipeline* deve garantir que o sinal *stall* está baixo antes de passar os seus próprios dados para o estágio seguinte. Isso garante um fluxo de dados adequado de um estágio para o outro evitando corrupção de dados ao longo do *pipeline*.

3.3 Instruções CUDA

A arquitetura FlexGrip suporta o conjunto de instruções da NVIDIA G80, com capacidade de computação 1.0. Na tabela 3.2 é apresentada a lista das instruções aceitas

pela arquitetura FlexGrip, bem como uma breve explicação da funcionalidade de cada uma delas.

Tabela 3.2: Instruções aceitas pela arquitetura FlexGrip

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

4 MODELO DE INJEÇÃO DE FALHAS

Um injetor de falhas deve sempre ser capaz de satisfazer os seguintes requisitos: representatividade, eficiência e mínima interferência. As falhas injetadas devem representar o mesmo comportamento de falhas reais que ocorrem durante a execução do sistema. Em outras palavras, falhas devem ser injetadas uniformemente ao longo do *hardware* realmente utilizado pela aplicação, cobrindo todas as instruções executadas pela aplicação. Além disso, o modelo de injeção de falhas deve ser rápido o suficiente para completar o algoritmo em um tempo razoável, possibilitando assim a execução de milhares de injeções a fim de obter dados estatisticamente significantes. Por fim, o método de injeção deve interferir minimamente no comportamento na aplicação original, não modificando as suas características. Em outras palavras, o injetor de falhas não deve mudar nem dados nem código da aplicação original que não sejam relacionados com o ato da injeção de falha.

Uma maneira eficiente de atingir a representatividade necessária no processo de injeção de falhas é utilizando o método de bombardeamento de nêutrons no dispositivo, através do uso de aceleradores de partículas. Para a execução deste método, é necessário um dispositivo que faça o bombeamento de nêutrons direcionado no dispositivo em teste. O IP da arquitetura FlexGrip é facilmente sintetizável em FPGA, o que é ótimo quando realizamos estudos sobre métodos eficientes de detecção e correção de erros, pois é possível adicionar componentes relevantes na descrição RTL do sistema e propor mecanismos dedicados de tolerância e detecção de falhas para a arquitetura de GPUs.

Um dos problemas desta arquitetura é a grande quantidade de recursos necessários para sintetizá-la em FPGAs, a opção mais viável para realizar testes de radiação nesta arquitetura. Na tabela 4.1 podemos observar a estimativa de recursos para o FlexGrip implementado e sintetizado para o FPGA Virtex-6 XC6VLX240T-1FFG1156. É possível observar a grande diferença em área para configurações de 8 SPs, 16 SPs e 32 SPs. Essa medida de ocupação indica a necessidade do uso de plataformas de desenvolvimento robustas para a sintetização da arquitetura FlexGrip em FPGAs, o que aumenta drasticamente o custo de testes por radiação.

Tabela 4.1: Área de diferentes implementações do FlexGrip implementado em um FPGA da família Virtex-6 XC6VLX240T-1FFG1156.

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

A liberdade proporcionada pela descrição RTL da arquitetura FlexGrip proporciona a definição de um modelo de injeção de falhas transientes por simulação, surgindo como uma alternativa ao alto custo atrelado a testes de injeção de falhas por radiação. Nos subcapítulos seguintes, o modelo de injeção de SEUs por simulação é apresentado, com uma explicação detalhada de cada um de seus componentes.

4.1 Modulo do Injetor de Falhas

Inicialmente, a ferramenta de simulação escolhida para conduzir os experimentos de injeção de falhas foi o ISim (*ISE Simulator*), embarcado no pacote ISE 14.7 da Xilinx. A longa demora na execução dos *benchmarks*, aliada com a impossibilidade de paralelizar as execuções dos ambientes de teste, além das constantes falhas do simulador inviabilizaram o seu uso no modelo de injeção. Muitas etapas do desenvolvimento do injetor foram realizadas nesta ferramenta, porém as campanhas de injeção eram constantemente interrompidas por problemas do ambiente de teste, sendo posteriormente abandonado o seu uso em detrimento da ferramenta *Modelsim SE* (MODELSIM, 2010). Com esta ferramenta, foi possível controlar execuções paralelas de scripts de injeção, bem como execuções limpas e sem falsos alarmes, como problemas da própria ferramenta refletindo falsos comportamentos errôneos da GPU.

Um levantamento inicial de dados foi feito sobre a GPGPU FlexGrip, a fim de escolhermos o melhor método de injeção de falhas, levando em conta o tempo total de execução por rodada de injeção que, apesar de ser muito melhor com a mudança de ferramenta de simulação, continua relativamente alto para atingir dados estatisticamente representativos. Neste levantamento, foi constatada a presença de 440.321 sinais intermediários na arquitetura, considerando cada posição de um vetor de sinais como um sinal independente, com a finalidade de já identificarmos os sinais da arquitetura no formato adequado para comandos específicos do simulador, utilizados no processo de injeção.

A grande quantidade de sinais inviabiliza uma injeção de perturbações em sinais aleatórios do sistema, pois seriam necessárias repetições consideráveis de injeções em cada sinal, o que é inviável quando cada rodada de injeção apresenta um tempo considerável de execução. Desta maneira, a simulação de SETs foi posta de lado em detrimento da simulação de SEUs nos diversos bancos de registradores presentes na arquitetura. A escolha do SEU foi feita por se tratar de uma falha extremamente sensível para o sistema, pois altera de fato um valor alocado em um elemento de memória do sistema, que fica muito mais suscetível a apresentar um erro no resultado final do que uma simples perturbação em um sinal aleatório da arquitetura. Na figura 4.1 podemos observar um diagrama de blocos do modelo de injeção de falhas adotado, destacando os diversos elementos da GPU e componentes do modelo de injeção de falhas utilizado.

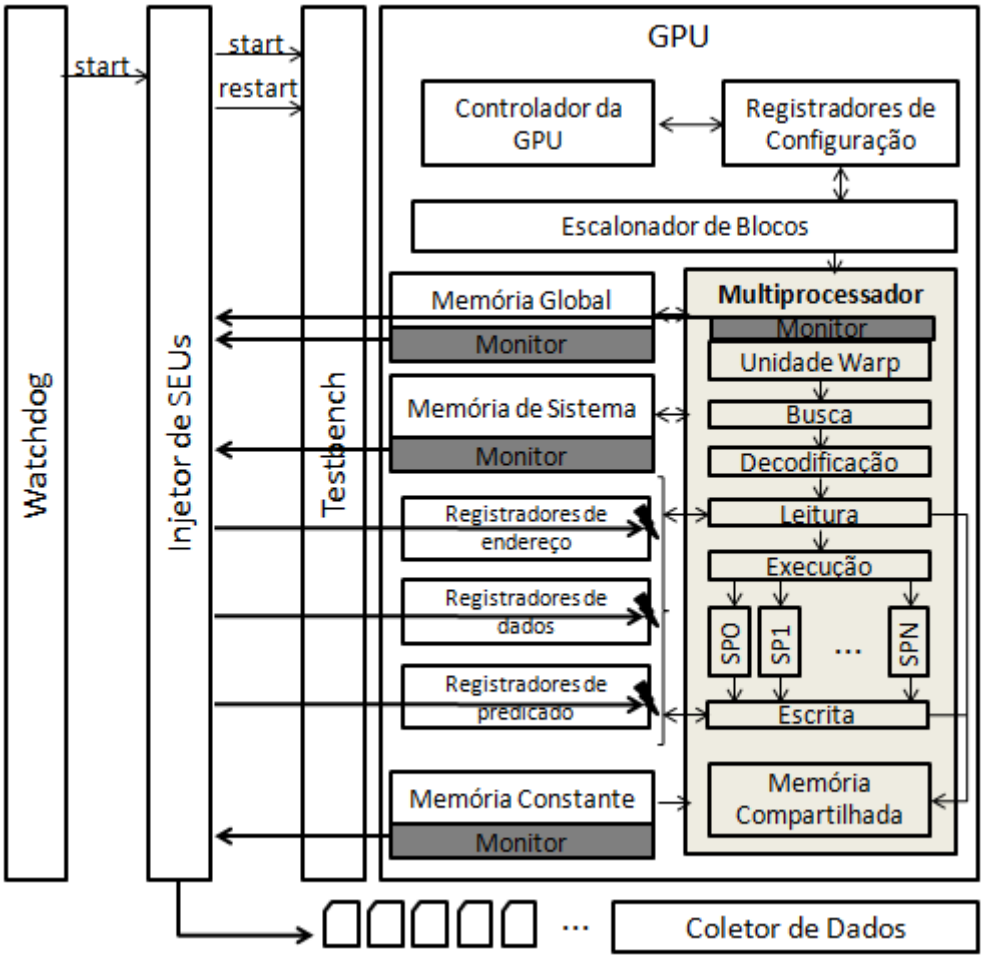

Figura 4.1: Arquitetura FlexGrip e componentes do módulo injetor de falhas

O injetor de falhas é descrito em *scripts* na linguagem *Tool Command Language* (TCL), que tem sua sintaxe reconhecida pelo simulador *modelsim*. O processo de injeção aguarda o processo de transferência de informações do *host* para a GPU, iniciando assim a sua ação. O *host* neste ambiente é formado por processos executados no *testbench* e pelo microprocessador MicroBlaze. Vencida esta etapa, o injetor irá executar o algoritmo normalmente até atingir um tempo aleatório de injeção pré-determinado. Caso as condições necessárias para a injeção não sejam atingidas, o injetor irá descartar a rodada. Caso contrário, o resultado parcial da injeção será salvo em formato *Comma Separated Values* (CSV), o que facilita muito a análise dos resultados finais das campanhas de injeção de falhas no programa *Excel*, além de ser facilmente criado, bastando separar as informações de cada coluna por ponto-e-vírgula. Na tabela 4.2 são descritas as informações parciais de injeção fornecidas pelo injetor.

Tabela 4.2: Informações parciais produzidas pelo processo de injeção

Tempo de injeção	Tempo transcorrido da execução até o momento da injeção do SEU.
Banco	Indicador de banco de registradores afetado, que pode ser: de endereço, de dados e de predicado
ThreadId	Identificador da *thread*
LaneId	Identificador da linha do *warp*
Bit	Identificador do bit afetado
Classificação do Erro	SDC – Erro no resultado final. Simples ou múltiplo. HANG – Sistema teve sua execução interrompida ou entrou em *loop* UNACE – Falha mascarada
Monitores	Acessos à memória diferentes da execução normal; *Program Counter* diferente da execução normal.

Para atingirmos dados estatisticamente significativos na campanha de injeção, foi necessária a adição dos módulos coletor de dados e *watchdog*, o primeiro sendo responsável pelo controle dos resultados intermediários de cada execução paralela de cada ambiente de injeção de falhas, e o segundo responsável por garantir o funcionamento correto do injetor, sem ocorrer interrupções geradas pelo computador hospedeiro ou pela própria ferramenta de injeção. Mais detalhes de cada módulo serão abordados nas próximas seções.

4.2 Algoritmo de Injeção de Falhas

Um dos principais problemas encontrados no modelamento de SEUs por simulação diz respeito à intrusividade da injeção no comportamento normal do sistema. Para injetar um *bit-flip* em um elemento de memória é necessário que esse elemento esteja com o sinal de escrita alto, caso contrário qualquer alteração na porta de escrita será desprezada. Com isso em mente, a principal questão a ser levantada é como injetar uma falha em um tempo aleatório da execução e garantir que essa falha irá satisfazer as seguintes condições: alterar apenas um bit do valor final do registrador e não interromper alguma operação pendente no controlador de registradores.

Para entender o método de injeção utilizado, é necessário recapitular brevemente a distribuição do banco de registradores ao longo processadores escalares. A arquitetura FlexGrip pode ser configurada com oito, dezesseis ou trinta e dois SPs, distribuídos ao longo de um único SM. Um *warp* contém trinta e duas *threads*, e pode ser divido em uma, duas ou quatro linhas, que irão determinar o particionamento do banco de registradores de dados. Na figura 4.2 podemos verificar a distribuição dos bancos de registradores ao longo das *threads* contidas em cada linha do *warp*. Cada *thread* ao longo da linha do *warp* contém o seu próprio conjunto de registradores em seu próprio

banco, habilitando assim escrita e leitura paralelas para todas as *threads* contidas no *warp.*

Ao entrar no estágio de leitura ou escrita, o controlador dos bancos de registradores irá acessar os bancos paralelamente, sendo cada página de memória dos bancos endereçada pelo índice da linha do *warp* e pelo próprio índice da *thread* dentro de cada linha. É conveniente lembrar que as operações nos bancos de registradores não irão levar em conta os limites do tamanho do bloco de *threads*, realizando a operação para todas as 32 *threads* contidas no *warp,* independentemente da configuração escolhida.

Dada a complexidade do multiprocessador da GPGPU, garantir intrusividade mínima na injeção de um SEU torna-se uma tarefa muito difícil. Qualquer instante de tempo de execução que for escolhido para a injeção poderá ser um momento em que o controlador já esteja realizando uma operação. Seria necessário introduzir uma máquina de estados que termine a operação pendente no controlador, garanta que nenhuma outra operação seja realizada no instante da injeção, realize a injeção e aí libere o controlador para a sua execução normal. Essa abordagem, além de claramente modificar o curso natural da execução do programa também não viabilizaria uma campanha de injeção de dados significativa, pois contamos com um elevado número de registradores por operação e um tempo de execução relativamente elevado para a maioria dos *benchmarks* utilizados.

Dessa maneira, foi feito uso de uma característica da arquitetura, que mantém alto o sinal de escrita para todas as *threads* do *warp* a cada ciclo de escrita ou leitura no controlador de registradores. Isso acontece porque o particionamento dos registradores por *thread* é dado pelas dimensões do *warp* que, independentemente das dimensões de bloco ou do número de linhas do *warp*, irá sempre contar com trinta e duas *threads* em sua constituição.

Como nem todas as instruções dos *bencharmks* vão operar sobre todas as unidades de processamento, fazemos uso desse breve período em que os bancos de registradores têm o seu sinal de escrita ativo para injetarmos um *bit-flip* em um banco de registradores aleatório de uma *thread* do *warp*. Dessa maneira, garantimos que o *bit-flip* irá perturbar de alguma maneira o sistema, sendo essa perturbação percebida de fato ou não pela aplicação. Na figura 4.2 podemos observar o diagrama de blocos do processo de injeção proposto. Neste exemplo, a *thread* escolhida para receber o *bit-flip* em seu registrador pré-selecionado pelo controlador foi a de número 11. Desta maneira, o injetor irá injetar a falha na primeira linha do banco de registradores de número 3. O injetor irá executar um período de relógio por vez até encontrar um intervalo de tempo em que o controlador esteja preparado para escrever no banco de registradores desejado. Caso esse intervalo nunca seja encontrado e a injeção atingir o tempo total de execução do algoritmo, a execução será descartada e a injeção reiniciada.

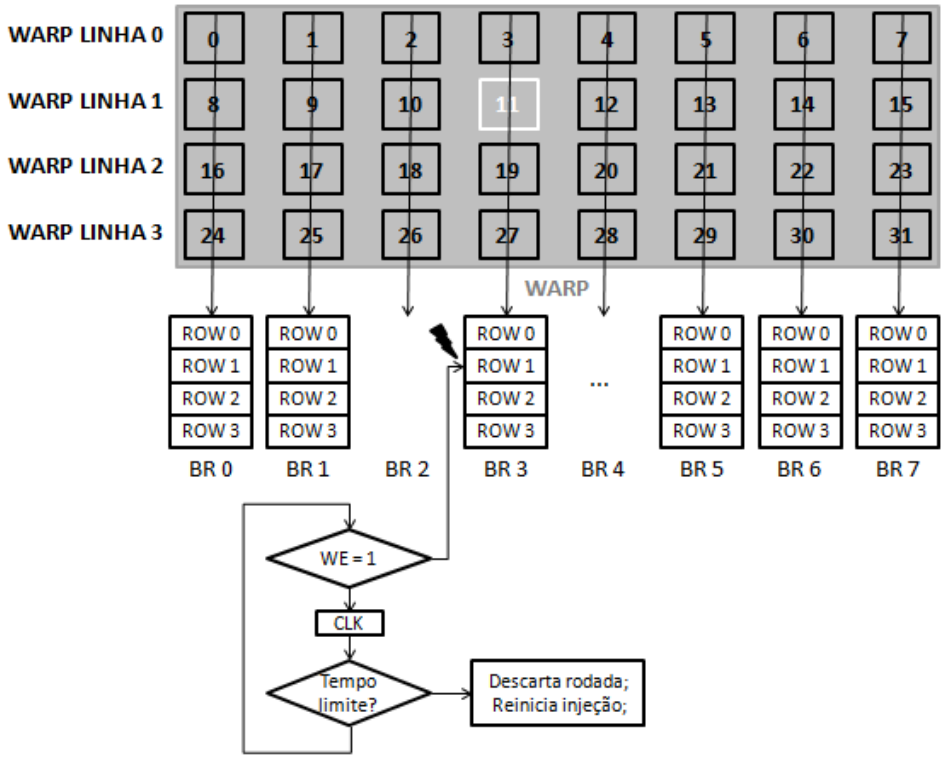

Figura 4.2: Injeção de SEU no banco de registradores

A tática abordada não reflete perfeitamente um comportamento de um injetor de nêutrons, por exemplo, onde a aleatoriedade da falha é garantida, visto que o bombardeamento é feito em toda a unidade de processamento avaliada. No nosso método, não conseguimos apresentar uma aleatoriedade real, porém podemos estudar claramente os efeitos de uma perturbação em um elemento de memória extremamente importante da arquitetura, e assim indicar a sensibilidade de GPUs a esse tipo de falha, bem como propor técnicas eficazes que podem ser facilmente avaliadas pelo método de injeção de falhas proposto.

4.3 Coletor de Dados e Watchdog

O coletor de dados tem a função de realizar o controle do número total de injeções desejadas na campanha. Seu uso foi motivado pelo tempo necessário para o simulador realizar uma rodada de execução do algoritmo, que nesta arquitetura é consideravelmente elevado para uma campanha de injeção de falhas. Na tabela 4.3 apresentamos o tempo total de execução do algoritmo de multiplicação de matrizes em diferentes dimensões de matrizes de entrada, bem como o tempo médio real de execução do simulador.

Tabela 4.3: Tempo de execução do algoritmo multiplicação de matrizes

Dimensão das matrizes de entrada	Tempo de execução em ciclos de relógio	Tempo de simulação (segundos)
8x8	33220	40
16x16	231996	280
32x32	1758058	2121

Como podemos observar o tempo de simulação do algoritmo para matrizes de entrada de dimensão 32x32 leva aproximadamente 35 minutos para completar uma única rodada de injeção. Matrizes de dimensão 32x32 possuem um volume de dados considerável para ser processado pela GPGPU, garantindo a possibilidade de explorar os recursos computacionais da GPGPU em sua totalidade. Porém, para atingirmos dados estatísticos suficientes para uma análise fiel do comportamento da GPGPU, são necessárias diversas rodadas de injeção de falhas. Considerando-se três casos diferentes de testes para cada algoritmo, como são conduzidos os experimentos deste trabalho, estas matrizes de entrada necessitariam de muito tempo hábil para completar a sua campanha.

Desta maneira, foi necessário introduzir um modelo de execução paralela dos processos de injeção de falhas. O modelo faz uso do gerenciador de tarefas do sistema operacional *Windows*, que permite o disparo de tarefas específicas, como programas, ações de sistema, execução de *scripts*, dentre outros. Outra grande facilidade proporcionada por esta ferramenta é a possibilidade de agendar periodicamente a execução da tarefa, além de determinar ações de término ou reinício de processos após um determinado intervalo de tempo.

A figura 4.3 apresenta um diagrama geral do processo de injeção de falhas. O processo inicia pela ação do *watchdog*, que é definido como uma regra habilitada no gerenciador de tarefas do *Windows*. Essa regra é disparada toda vez que o sistema hospedeiro do ambiente de injeção de falhas sofre um *start*, e é responsável por disparar os processos de injeção dos K ambientes de injeção. Cada ambiente de injeção conta com o seu próprio *script* descrito em *batch*, que será responsável por disparar N processos do simulador *Modelsim.* A execução paralela dos simuladores é dada pelo comando *START*, que dispara um processo novo para cada chamada realizada.

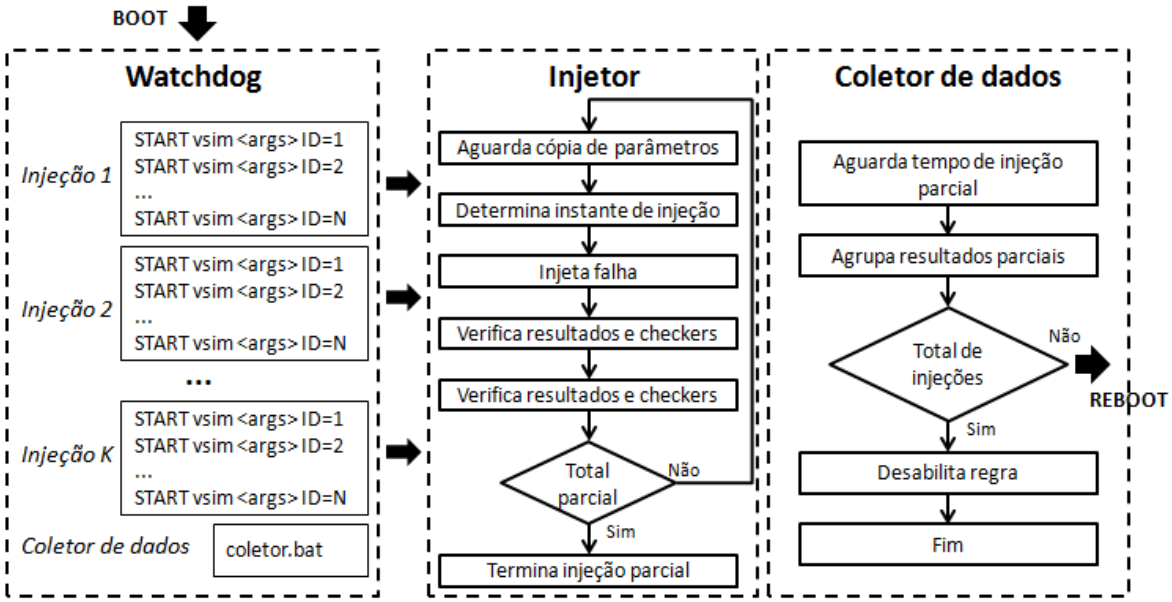

Figura 4.3: Processo de injeção de falhas

Ambientes com processos paralelos atuando no mesmo conjunto de dados possuem o problema do compartilhamento de recursos, que deve ser controlado para que um processo não interfira no processamento normal do outro. No nosso modelo de injeção, garantimos a sincronização dos dados gerados pelos processos injetores fazendo com que cada processo de injeção tenha o seu próprio conjunto de dados associado. A cada rodada de injeção, um arquivo *csv* contendo informações sobre a rodada de injeção é criado. Tomando por exemplo um arquivo da campanha de injeção de falhas deste trabalho, *17 Apr 2015 – 12:31:15 @1.csv*, podemos extrair o horário em que a injeção terminou e o identificador do processo responsável por esta injeção específica.

Após o disparo dos processos de injeção para cada *benchmark*, o *Watchdog* então dispara o processo coletor de dados. Este processo é dimensionado para aguardar o tempo necessário para que as injeções de todos os ambientes sejam concluídas. Para evitar que um processo de injeção de um determinado ambiente de injeção fique inerte aguardando longos períodos de tempo até que um ambiente mais lento termine a sua execução, todos os processos tem seu tempo médio de execução pré-determinado, levando em conta todo o cenário com diversos processos de outros *benchmarks* sendo executados paralelamente. Este balanceamento é determinado executando o ambiente completo de injeção por um tempo bastante reduzido e esperando o sistema estabilizar, isto é, quando todos os processos de simulação já tiverem iniciado e produzido os primeiros resultados parciais de injeção. Desta maneira, é possível determinar o intervalo médio de execução de cada simulador em cada ambiente de injeção através do horário contido em cada resultado parcial de simulação. Foi determinado de maneira empírica que o ponto ótimo de desempenho é atingido com até dez processos simultâneos do módulo injetor.

4.4 Monitores e configuração do ambiente

A configuração do ambiente de injeção é feita em duas etapas, descritas na figura 4.4. Na primeira etapa, é necessário configurar a arquitetura FlexGrip, definindo a quantidade de *threads* por bloco (I) e o número de blocos por *grid* (II) da aplicação, que são definidos pelas fórmulas:

I. Número de threads = dimensão_bloco_x * dimensão_bloco_y

II. Número de blocos = dimensão_grid_x * dimensão_grid_y

A memória global do sistema deve ser inicializada com os vetores de entrada do *benchmark*. A recompilação do sistema é necessária para validar a nova configuração, sendo seguida pela execução da simulação com o *script* gerador dos componentes monitores.

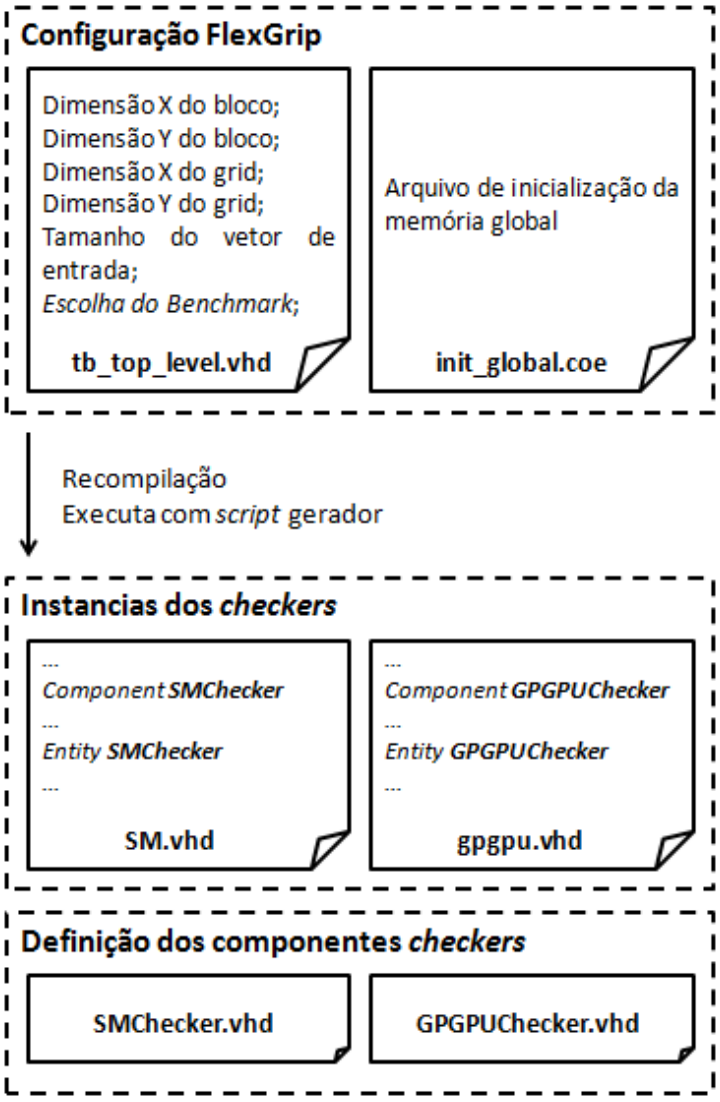

Figura 4.4: Fluxo de configuração do ambiente de injeção

Os monitores de memória são desenvolvidos através de uma análise prévia do número de acessos a cada endereço utilizado em cada memória do sistema. Em outras palavras, o *script* irá gerar um gabarito de acessos a posições de memória realizadas pela execução do algoritmo. Esse gabarito dará origem aos componentes monitores de memória, que são posteriormente adicionados à arquitetura e são monitorados a cada rodada de injeção a fim de verificar se algum acesso inesperado ocorreu em alguma das memórias da GPGPU. Todas as memórias da arquitetura são verificadas: memória compartilhada, memória global e memória constante.

Analogamente, o monitor do PC irá salvar um rastro de execução do PC, sendo verificado se os acessos realizados pelo programa ao PC condizem com a execução livre de falhas, descrita na figura 4.4. Na figura 4.5 é representada a metodologia utilizada para a criação dos componentes monitores de memória e PC. Endereços de memória sem acesso algum são descartados das tabelas de contadores a fim de economizar espaço na arquitetura.

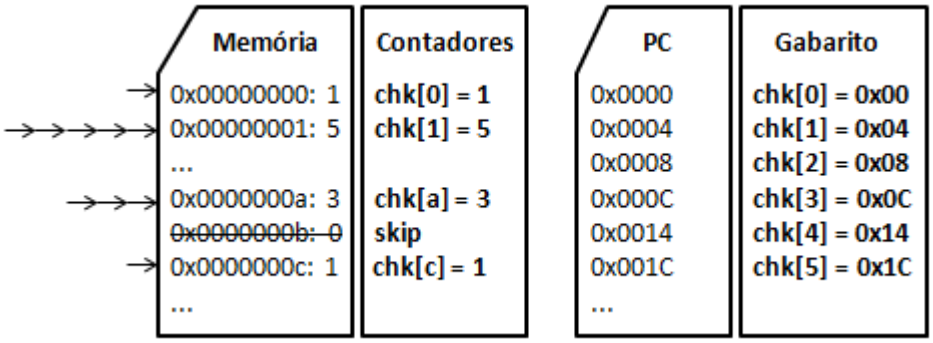

Figura 4.5: Metodologia utilizada para a criação dos monitores de memória e PC

O *script* gerador irá armazenar os dados gerados a partir da consulta dos sinais de endereço de memória e de PC de cada estágio para daí formatá-los em descrição HDL para que possam ser adicionados automaticamente no *design.* Uma recompilação é necessária para adicionar os monitores na arquitetura, finalizando a criação do ambiente de injeção. Na tabela 4.4 listamos as configurações necessárias para montar o ambiente de injeção.

Tabela 4.4: Configurações do módulo de injeção

Configuração	**Descrição**
blockDimx	Dimensão no eixo *x* do bloco de *threads*
blockDimy	Dimensão no eixo *y* do bloco de *threads*
gridDimx	Dimensão no eixo *x* do *grid*
gridDimy	Dimensão no eixo *y* do *grid*
Algoritmo	Algoritmo a ser utilizado na simulação
Número de injeções total	Número total de injeções a serem executadas
Número de injeções parcial	Número de injeções parciais de cada processo de cada ambiente
Tempo limite do coletor de dados	Tempo que o coletor de dados irá esperar para agir nos dados parciais da execução. Deve ser dimensionado juntamente com o número de injeções parcial.
Número de processos de simulação por ambiente	Número de processos paralelos de simulação a serem disparados

5 RESULTADOS EXPERIMENTAIS

Neste capítulo, serão apresentados os resultados das campanhas de injeções realizadas neste trabalho. Cada cenário demonstrado contou com a injeção de 1000 SEUs em cada banco de registradores analisado. As simulações foram realizadas utilizando o modelo de injeção de falhas descrito no capítulo 4 e foram analisadas através da ferramenta *Microsoft Office Excel*. Seu uso é claramente justificado pelo formato CSV escolhido para a geração dos relatórios de injeção de falhas. A GPU FlexGrip foi simulada com referência a um FPGA Virtex 6 da Xilinx, utilizando elementos de pré-sintetizados de memória (IPs) disponíveis para plataformas de desenvolvimento da Xilinx. Analises técnicas do modelo de injeção de falhas também foram extraídas com o intuito de melhorar o processo para futuros experimentos. Medidas precisas de configurações eficientes do injetor podem ser extraídas através da análise destes dados, viabilizando ainda mais futuras pesquisas nesta área.

A totalidade do processo de injeção de falhas, incluindo as rodadas de injeção, preparação de ambientes e agrupamento de resultados levou aproximadamente 43 horas para ser completa. Vetores pequenos de dados de entrada foram escolhidos a fim de diminuir o tempo total da campanha de injeção. Essa escolha causa certo impacto na aplicabilidade dos resultados, visto que os algoritmos escolhidos podem ser resolvidos em tempo aceitável até mesmo por CPUs. GPUs são exploradas na resolução de algoritmos para grandes volumes de dados, fazendo uso desta maneira de todas as características de paralelismo da arquitetura. Porém, para fins de validação do modelo de injeção, pouco importa o tamanho do vetor de entrada, contanto que ele não omita características primordiais de paralelismo da GPU e esteja de acordo com resultados apresentados no estado da arte.

Todos os algoritmos foram compilados através do compilador CUDA *nvcc* (NVIDIA, 2015). O processo de compilação de algoritmos CUDA é demonstrado na figura 5.1. Durante o tempo de compilação, o compilador *nvcc* é abastecido com o kernel CUDA, convertendo-o em um código denominado *Parallel Thread Execution* (PTX). O PTX é uma linguagem de programação *assembly* que explora a GPU como um dispositivo de computação paralela, definindo um modelo de programação estável e um conjunto de instruções virtuais interpretadas por GPUs da NVIDIA. A linguagem PTX não representa diretamente o conjunto de instruções de máquina da GPU, sendo apenas uma linguagem intermediária que é compilada para instruções específicas *assembly*, dependendo da capacidade computacional de cada GPU. Durante o tempo de execução, o código PTX passa pelo driver CUDA da API (*Application Programming Interface*), que é o responsável por gerar o binário CUDA, ou *.cubin*, que é então direcionado para a GPU que será responsável por sua execução. Como a campanha de injeção de testes não é realizada em uma GPU comercial, a ação do *driver* é substituída pelo uso de bibliotecas disponibilizadas pela NVIDIA.

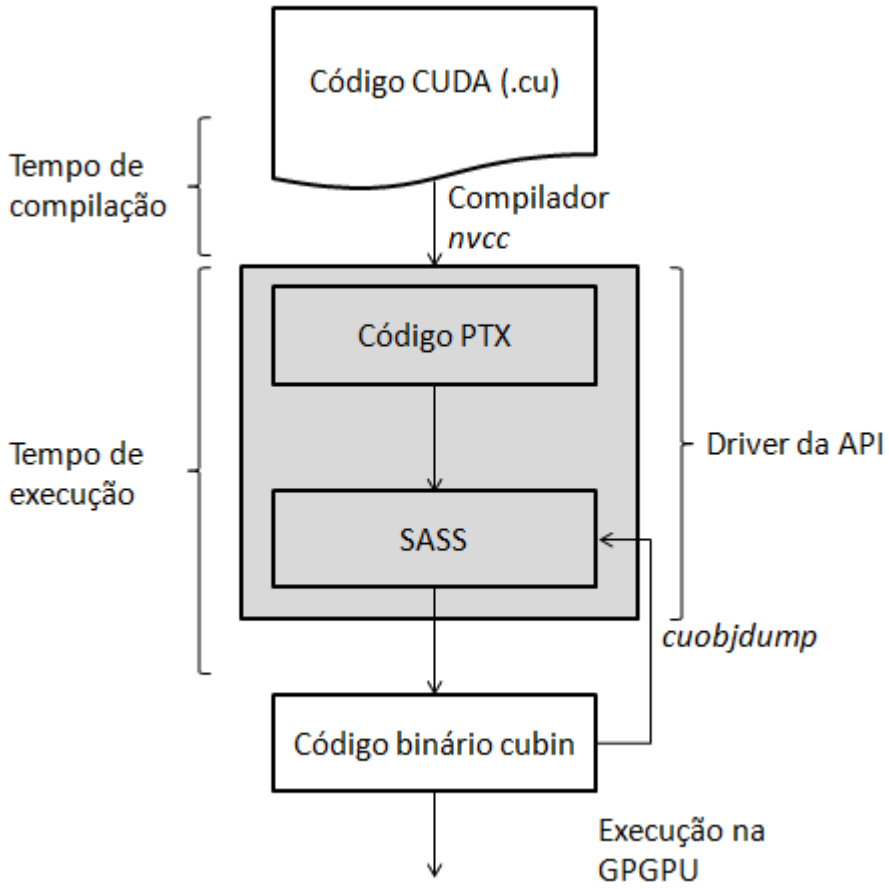

Figura 5.1: Fluxo de software de uma GPU

Para realizar a campanha de injeção na GPGPU FlexGrip, é necessário o uso de instruções *assembly* que de fato correspondam ao binário gerado pelo processo de compilação. Como mencionado anteriormente, o código PTX não corresponde a instruções binárias reais a serem executadas pela GPU. O código PTX, durante o tempo de compilação, é utilizado para a geração de outro formato de código denominado *Source and Assembly* (SASS) (AAMODT, 2012), como demonstrado dentro do processo realizado pelo driver CUDA. Este formato é específico para a arquitetura destino de GPU, representando instruções nativas que são interpretadas por hardwares da NVIDIA. É interessante mencionar que essa conversão não é visível para o usuário final, sendo necessário o uso da ferramenta *cuobjdump* (NVIDIA, 2012), disponibilizada pelo *toolkit* CUDA da NVIDIA, que compreende diversas ferramentas necessárias para o modelo de programação CUDA, além do compilador *nvcc* previamente mencionado.

O restante deste capítulo é organizado da seguinte maneira. No primeiro subcapítulo, apresentaremos os algoritmos utilizados na campanha, explicando as características de cada um deles bem como a maneira que é feito o processamento paralelo em cada um deles. A seguir, são apresentados resultados de validação do injetor e por fim, no subcapítulo três, é apresentado um breve estudo sobre a manifestação das falhas ao longo dos componentes da GPU.

5.1 Algoritmos utilizados nas campanhas de injeção

A seleção das aplicações a serem utilizadas nas campanhas de injeção é uma etapa primordial no estudo de confiabilidade de sistemas. Cada algoritmo possui características próprias, fazendo uso de diferentes recursos do dispositivo em teste. Idealmente, o uso de *benchmarks* é o mais indicado para este tipo de validação. Entretanto, a campanha de injeção de falhas descrita nesse capítulo envolve diversas etapas de processamento de falhas e resultados, sendo algumas delas realizadas manualmente, o que torna muito dispendioso a simulação de diversas aplicações. Por esta razão, três aplicações paralelas foram escolhidas neste trabalho: um algoritmo de multiplicação de matrizes, um algoritmo de *bitonic sort* (BATCHER, 1968) e finalmente um algoritmo de autocorrelação de sinais. Os dois primeiros algoritmos são submetidos ao processo completo de análise de falhas, sendo realizados experimentos com diferentes distribuições de *threads* e a verificação do seu efeito na confiabilidade da aplicação. Para o algoritmo de autocorrelação somente a análise dos erros é realizada, com uma configuração de ocupação ótima de número de *threads* por bloco a fim de analisar apenas o comportamento do algoritmo em presença de falhas.

5.1.1 Multiplicação de matrizes

O algoritmo de multiplicação de matrizes foi escolhido por realizar uma massiva computação de instruções aritméticas em grandes porções de dados. Esta característica do algoritmo, aliada com o seu uso constante em aplicações que requerem processamento de um grande volume de dados, torna mandatória a sua presença em pesquisas sobre avaliação de confiabilidade em GPUs. Além disso, este algoritmo faz acesso constante a elementos de memória da arquitetura, sendo excelente para fins de análise de comportamento de falhas transientes em registradores. O mapeamento das matrizes de entrada do algoritmo para as diferentes unidades computacionais da GPU é direto, com intrínseca relação entre o índice da *thread* no bloco e como próprio índice do bloco do *kernel* a ser executado. Na figura 5.2 é exibido o algoritmo utilizado na campanha de injeção de falhas. O índice de coluna e linha da matriz é determinado pela relação entre o índice da *thread* dentro do bloco, o índice do bloco dentro do *grid* e a dimensão do bloco de *threads*. Sendo assim, cada bloco de *threads* será responsável pela computação de uma sub-matriz da matriz de entrada. Na figura 5.3 podemos observar as diferentes distribuições do bloco de *threads* que realizarão a computação de sub-matrizes da matriz original. Diferentes configurações de número de *threads* por bloco e de número de blocos por *grid* são escolhidas com o intuito de observar a relação entre confiabilidade e distribuição de *threads* por bloco em GPUs.

```
__global__ void
matrixMul(int* d_M,int* d_N,int* d_P,int Width)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int P_val = 0;

    for (int k = 0; k < Width; ++k) {
        int M_elem = d_M[row * Width + k];
        int N_elem = d_N[k * Width + col];
        P_val += M_elem * N_elem;
    }

    d_P[row*Width+col] = P_val;

    return;
}
```

Figura 5.2: Programa CUDA de multiplicação de matrizes

Esta descrição de multiplicação de matrizes em CUDA é bastante eficaz pois torna possível a configuração exata do número de *threads* operando sobre um conjunto definido de dados de entrada para cada SM da GPU, atingindo-se assim um grande grau de paralelismo. Dito isso, cada configuração diferente de *threads* por bloco terá uma característica de escalonamento e ocupação de SPs diferentes, tornando necessário o estudo do comportamento de cada configuração em presença de falhas. Configurações onde a ocupação de SPs é máxima são possivelmente sujeitas a experimentar uma sensibilidade maior a falhas pois utilizam mais elementos de memória a cada *warp* executado, sendo mais suscetíveis a apresentar erros derivados de SEUs.

A escolha do tamanho da matriz de entrada foi feita basicamente pela estimativa de tempo real de computação necessária para obtermos dados significativos de injeção de falhas. Como demonstrado anteriormente na tabela 4.2, a dimensão 8x8 da matriz de entrada foi a mais viável para uma campanha inicial de validação do modelo de injeção. O principal problema dessa escolha é a falta de volume de dados o suficiente para atingir uma ocupação ideal de recursos da GPU. Futuras campanhas de injeção de falhas são necessárias após a consolidação e validação do modelo de injeção proposto.

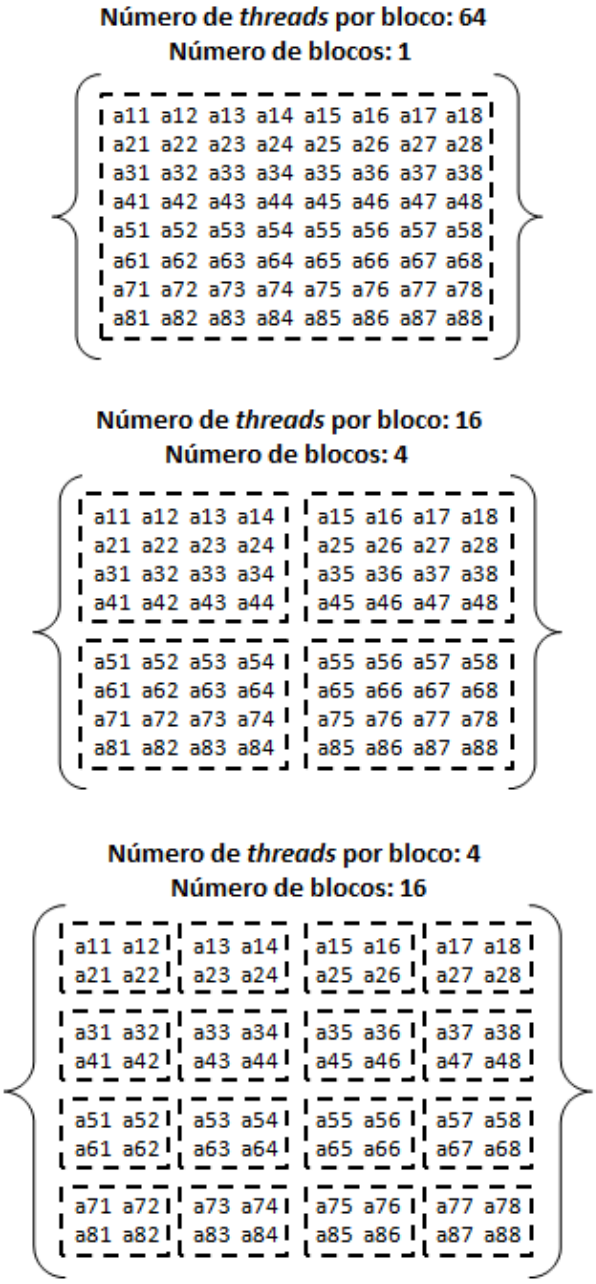

Figura 5.3: Distribuição de threads nos vetores de entrada do algoritmo de multiplicação de matrizes

5.1.2 Algoritmo *Bitonic Sort*

O algoritmo *bitonic sort* é um algoritmo paralelo de ordenamento de dados. O algoritmo monta redes de *sort* que nada mais são do que sequências de comparações que não possuem dependência entre si, permitindo assim o uso de arquiteturas paralelas para a sua eficiente execução. A rede ascendente do *bitonic sort* é apresentada na figura 5.4. Para um vetor de tamanho *n*, a rede *bitonic* consiste de $\Theta(n.log(n)^2)$ operações de comparação entre $\Theta(log(n))$ estágios, sendo que cada estágio realiza $n/2$ comparações.

A ponta da flecha aponta para o valor mais alto entre as duas linhas. Ao passar pela rede, todos os valores de entrada são ordenados em uma ordem ascendente na saída. Considerando a estrutura da rede, cada operação de comparação em cada estágio pode ser paralelizada.

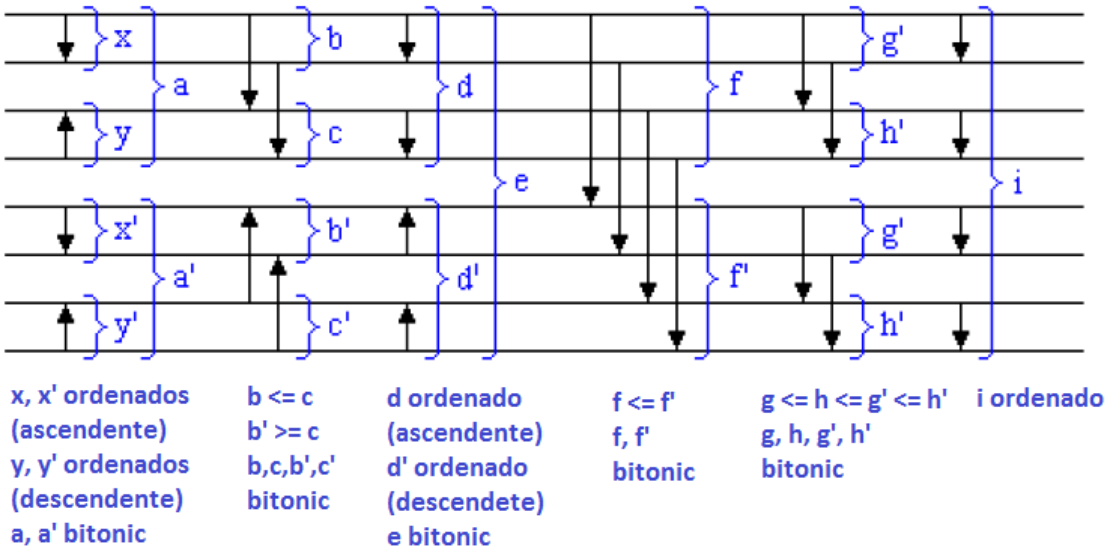

Figura 5.4: Rede de comparações do algoritmo bitonic sort

Como o algoritmo de *bitonic sort* opera em um tempo de simulação bem inferior ao de multiplicação de matrizes, optamos por utilizar um vetor de entrada de 32 valores distintos. As configurações de número de *threads* por bloco escolhidas para as campanhas de injeção de falhas são de 32, 16 e 8, sendo o número de blocos por *grid* configurado como um, dois e quatro, respectivamente.

A figura 5.5 apresenta a descrição do algoritmo *bitonic sort* em CUDA. Podemos observar a presença da instrução de sincronização de *threads*, visto que neste algoritmo há a presença de *threads* divergentes, ou seja, que tomam caminhos de execução diferentes residindo em um mesmo *warp*. Podemos notar também o uso da memória compartilhada, o que é interessante para os resultados de injeção de falhas a fim de analisar o comportamento do algoritmo quando uma perturbação atinge este componente, resultado de um SEU em um dos bancos de registradores.

```
__global__ static void bitonicSort(int * values)
{
    extern __shared__ int shared[];
    const unsigned int tid = threadIdx.x;

    shared[tid] = values[tid];
    __syncthreads();

    // Parallel bitonic sort.
    for (unsigned int k = 2; k <= NUM; k *= 2) {
        // Bitonic merge:
        for (unsigned int j = k / 2; j>0; j /= 2) {
            unsigned int ixj = tid ^ j;

            if (ixj > tid) {
                if ((tid & k) == 0) {
                    if (shared[tid] > shared[ixj])
                        swap(shared[tid], shared[ixj]);
                } else {
                    if (shared[tid] < shared[ixj])
                        swap(shared[tid], shared[ixj]);
                }
            }
            __syncthreads();
        }
    }

    // Write result.
    values[tid] = shared[tid];
}
```

Figura 5.5: Programa CUDA do algoritmo bitonic sort

Este algoritmo foi escolhido para as campanhas de injeção por se tratar de uma aplicação que faz uso constante de instruções de controle em sua execução. Desta maneira, procuramos comparar os resultados de algoritmos com diferentes características, avaliando qual tipo de aplicação é mais sensível a SEUs em GPUs.

5.1.3 Algoritmo de autocorrelação

Consistido basicamente por operações de multiplicação e adição, o algoritmo de autocorrelação realiza a correlação de um sinal consigo mesmo, sendo bastante utilizado para encontrar padrões de repetição, como em sinais periódicos mascarados por ruídos (DUNN, 2005). Este algoritmo foi escolhido por executar instruções semelhantes às do algoritmo de multiplicação de matrizes, porém com dependência entre os dados intermediários produzidos pelas computações da GPGPU. O paralelismo é atingido neste algoritmo pela designação de cada *thread* do bloco a computar um elemento do

array de autocorrelação. Na figura 5.6 podemos observar a descrição do algoritmo em CUDA. Neste algoritmo, podemos observar que o índice da *thread* é determinante no número de operações a serem realizadas por laço do *loop*, indicando uma forte dependência entre o comportamento do algoritmo e a *thread* a ser executada.

```
__global__ void autoCor(int * x, int *R, int size){

    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int sum = 0;

    if(tid < size) {
        for (int j=0;j<size-tid;j++)
            sum+=x[j]*x[j+tid];

        __syncthreads();

        R[tid]=sum;
    }
}
```

Figura 5.6: Descrição CUDA do algoritmo de autocorrelação

5.2 Análise da distribuição de *threads* na confiabilidade de GPUs

Esta seção apresenta o impacto da distribuição de threads na taxa de erros produzidos por SEUs na GPGPU FlexGrip. As campanhas de injeção foram realizadas nos principais bancos de registradores da arquitetura: de dados, de predicado e de endereço, lembrando que o último não é utilizado no algoritmo de multiplicação de matrizes. Os resultados das campanhas de injeção de falhas, apresentados nesta seção, remetem a três possíveis resultados da execução. Denomina-se HANG o erro que ocorre quando o sistema permanece em um estado inesperado de execução, que pode ser um loop infinito ou uma exceção de sistema, fazendo com que o algoritmo não termine sua execução. *Silent Data Corruption* (SDC) remete a erros em que a execução ocorreu normalmente, porém um ou mais posições do vetor de resultados do algoritmo apresentou um valor inesperado, diferente dos resultados de uma execução livre de falhas. Por fim, UNACE remete a execuções em que os SEUs foram mascarados pelo sistema, ou seja, não produzem erros.

A seguir são apresentados os resultados para cada banco de registrador, além da análise dos resultados obtidos.

5.2.1 Taxa de Erros devido a Falhas nos Registradores de dados

Nesta seção serão apresentados os resultados de injeção de SEUs nos registradores de dados da GPU. A figura 5.7 apresenta os resultados para o algoritmo de multiplicação de matrizes. Como esperado, configurações que utilizam mais

eficientemente os registradores do SM apresentam uma sensibilidade maior a SEUs. Pegando como exemplo a configuração de 4 *threads*, podemos observar que o índice de erros de dados, ou SDCs, corresponde a aproximadamente 10%, o que é um resultado ótimo se avaliarmos que as injeções são realizadas em registradores no seu ciclo ativo de armazenamento. A configuração de 16 *threads* por bloco apresenta aproximadamente o dobro da taxa de erros em relação à configuração anterior. Este resultado é explicado pela ocupação de *threads* no *warp* a ser executado pelo SM. Como o número de *threads* é o dobro do que a configuração de 4 *threads*, analogamente o dobro de registradores são utilizados para comportar o número maior de *threads* ativas contidas no *warp*. Assim, uma falha tem o dobro de chances de ser percebida pela computação do algoritmo, resultando em uma taxa de erros duplamente maior. Finalmente, no caso de 64 *threads*, observamos uma variação menos abrupta na taxa de erros quando comparada com a configuração de 16 *threads.* Neste caso, não temos a mesma relação entre taxa de erros e ocupação de registradores pois as 64 *threads* do bloco são acomodadas em 2 *warps* de 32 *threads* cada. Assim, temos uma maior eficiência na execução de instruções críticas, como adição e multiplicação, pois o algoritmo não contará com a carga extra de tempo de execução do escalonador de blocos, visto que apenas um bloco será designado para o SM. Analogamente, teremos uma maior carga do escalonador de *warps*, porém este demonstra ser mais eficiente, contribuindo menos para a taxa de erros final.

A taxa de erros de controle, ou HANG, não apresentam o mesmo comportamento dos SDCs. Em uma configuração com ocupação mínima, as instruções de controle são mais espaçadas ao longo da execução do algoritmo, fazendo com que essas instruções estejam mais suscetíveis a serem afetadas por um SEU quando comparada com a configuração de 16 *threads*. O mesmo comportamento não é observado na configuração de 64 *threads*, possuindo uma taxa de HANG ainda maior que a configuração intermediária. Como explicado anteriormente, as instruções de dados são executadas mais eficientemente, distribuindo de uma maneira mais uniforme as instruções de controle ao longo da execução do algoritmo, justificando assim a alta taxa de HANGs neste caso.

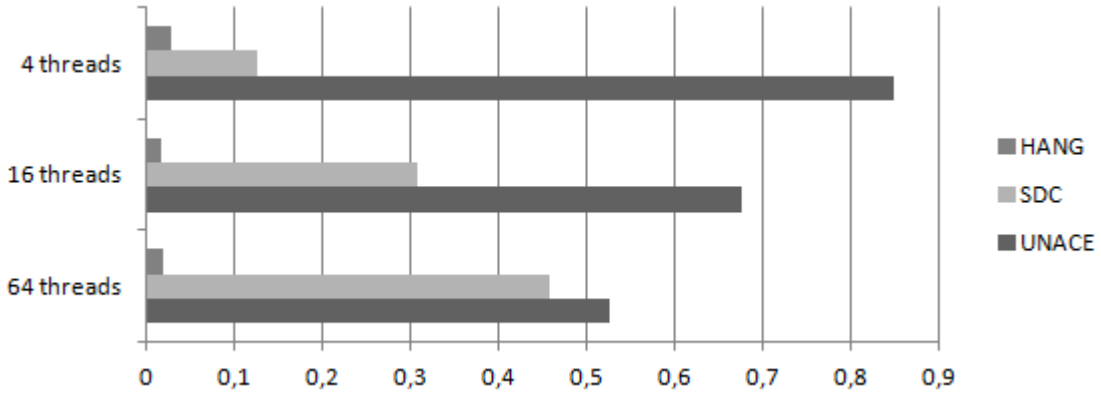

Figura 5.7: Efeitos de SEUs no banco de registradores de dados para o algoritmo de multiplicação de matrizes

A figura 5.8 demonstra esta relação de distribuição de falhas ao longo da execução do algoritmo. Podemos notar que as *threads* afetadas pela perturbação em seus registradores estão mais agrupadas quando o algoritmo é configurado para operar com

um número maior de *threads*. Analogamente, temos intervalos de tempo bastante grandes no caso de blocos com 4 *threads*, tornando ainda mais improvável que uma falha resulte em um erro no resultado final da aplicação.

Figura 5.8: Momento da injeção da falha nos registradores de dados que resultou em erro no algoritmo multiplicação de matrizes

Na figura 5.9 são apresentados os resultados da campanha de injeção de falhas no algoritmo *bitonic sort*. Podemos observar um comportamento semelhante na taxa de SDCs para este algoritmo quando comparado à multiplicação de matrizes. Uma ocupação maior do *warp* com *threads* ativas irá utilizar mais recursos de registradores por *warp*, aumentando a sensibilidade da GPU a experimentar erros causados por SEUs. Como todas as configurações ocupam apenas um *warp* por bloco de *threads*, a taxa de erros varia de acordo com o número operações de comparação realizadas pelo algoritmo, que é ditado pela fórmula $\Theta(n.log(n)^2)$. Como as operações de comparação tem relação direta com o índice da *thread* no warp, a taxa de erros pode ser derivada, da mesma maneira, pelo índice $\Theta(n.log(n)^2)$ operado sobre o número de *threads* por bloco. Esta relação explica o salto abrupto nos SDCs para a configuração com o maior número de *threads*, onde o número de operações de comparação totais realizadas por *warps* é menor do que em configurações com número menor de *threads* por bloco.

Já os erros de HANG apresentam uma maior taxa para configurações com menor número de *threads*. Neste caso, temos uma maior quantidade de instruções mais sensíveis a SEUs sendo executadas, visto que necessitamos de uma maior quantidade de *warps* para completar a execução do algoritmo. Esta característica faz com que o algoritmo de *bitonic sort* apresente uma relação intrínseca entre o número de *threads* por bloco e a taxa de erros produzidos pelo sistema em presença de falhas, especialmente quando o tamanho do bloco de *threads* é menor que o tamanho do *warp*; neste caso 32.

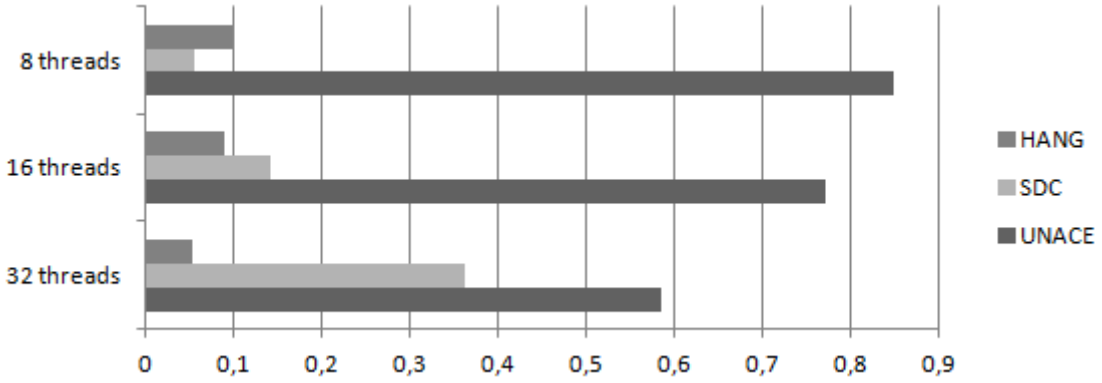

Figura 5.9: Efeitos de SEUs no banco de registradores de dados para o algoritmo de bitonic sort

Na figura 5.10 apresentamos a distribuição de falhas que resultaram em erros ao longo do tempo de execução do algoritmo de *bitonic sort*. Analogamente á multiplicação de matrizes, configurações com maior número de *threads* tendem a agrupar mais as operações mais sensíveis a SEUs que operam em uma maior quantidade de *threads* do sistema, fazendo com que estas configurações sejam mais sujeitas a experimentar erros.

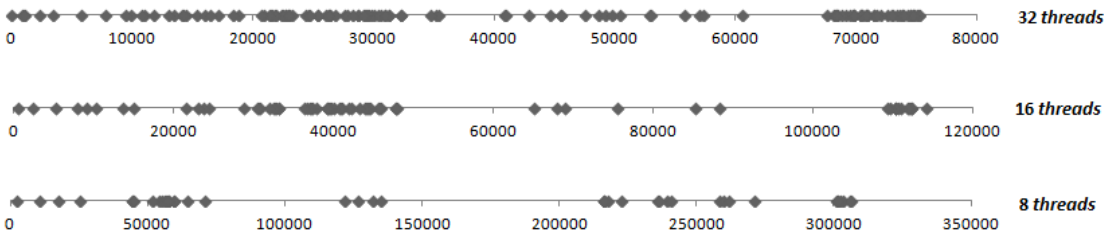

Figura 5.10: Momento da injeção da falha nos registradores de dados que resultou em erro no algoritmo bitonic sort

5.2.2 Taxa de Erros devido a Falhas nos Registradores de predicado

Nesta seção, serão apresentados os resultados de injeção de SEUs no banco de registradores de predicado. As figuras 5.11 e 5.12 apresentam, respectivamente, os resultados para os algoritmos de multiplicação de matrizes e *bitonic sort*, respectivamente.

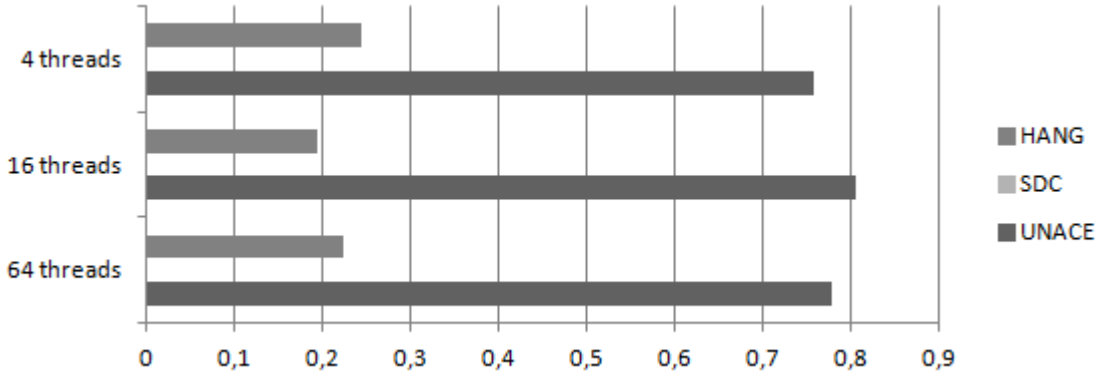

Figura 5.11: Efeitos de SEUs no banco de registradores de predicado para o algoritmo de multiplicação de matrizes

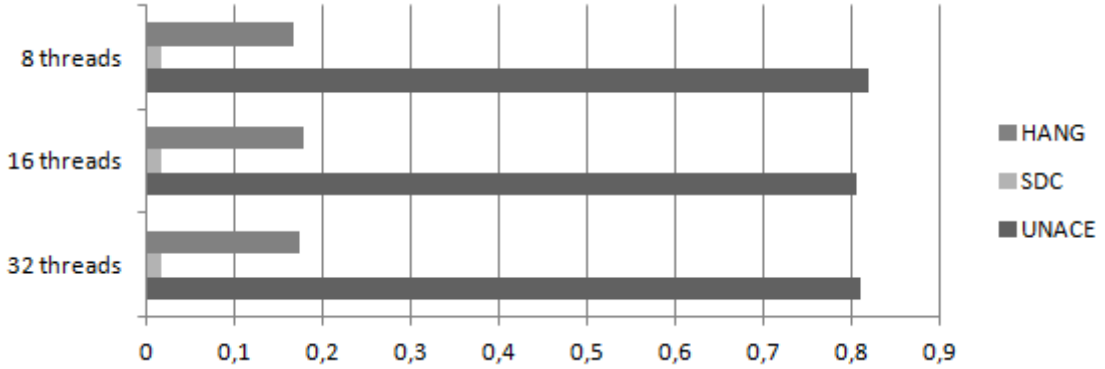

Figura 5.12: Efeitos de SEUs no banco de registradores de predicado para o algoritmo de bitonic sort

Podemos observar que a variação do número de *threads* por bloco não tem um impacto tão significativo na taxa de erros produzidos por SEUs nos registradores de predicado. Este resultado pode ser explicado pela própria finalidade destes registradores na execução das instruções. Registradores de predicado armazenam informações relevantes para instruções de *branch*, sendo deste modo menos sensíveis a variações no número de *threads* operativas residentes no *warp*. Podemos observar que no caso de multiplicação de matrizes a configuração com 16 *threads* apresentou a menor sensibilidade a SEUs nos registradores de predicado. Apesar da diferença para os outros casos ser muito pequena, este comportamento pode ser explicado novamente pelo número de instruções sensíveis a SEUs executadas na configuração de 64 *threads*. Um número menor destas instruções indica maior probabilidade de que um SEU atinja uma instrução de controle, resultando em um índice maior de taxas de HANG. A configuração de 4 *threads*, por outro lado, apresenta um maior índice de instruções que utilizam registradores de predicado, devido à sua baixa ocupação de *threads* por bloco. Mais operações de movimentação entre memória e registradores são necessárias, aumentando o uso dos registradores de predicado, tornando configurações com número pequeno de *threads* por bloco mais suscetíveis a erros, apresentando um valor ótimo com meia ocupação do *warp* para os casos de estudo apresentados.

As figuras 5.13 e 5.14 representam o momento da injeção que resultou em erros para os algoritmos de multiplicação de matrizes e *bitonic sort*, respectivamente. Comparando com a distribuição nos casos de injeção nos registradores de dados, podemos observar que as falhas ocupam posições bem distintas ao longo do tempo de execução. Esta observação confirma o comportamento diferente observado por injeções em cada banco de registradores, afetando instruções distintas, em sua maioria, para cada algoritmo.

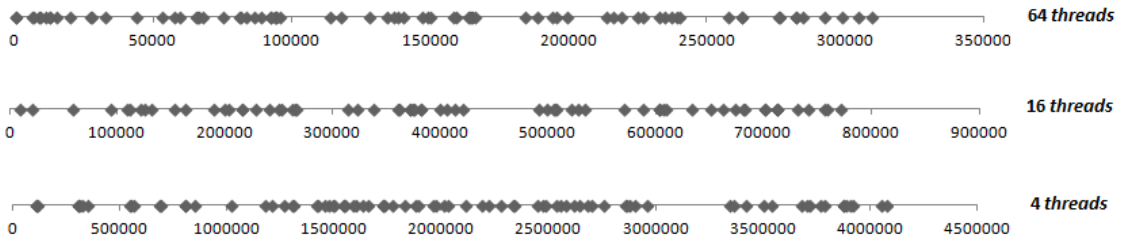

Figura 5.13: Momento da injeção da falha nos registradores de predicado que resultou em erro no algoritmo multiplicação de matrizes

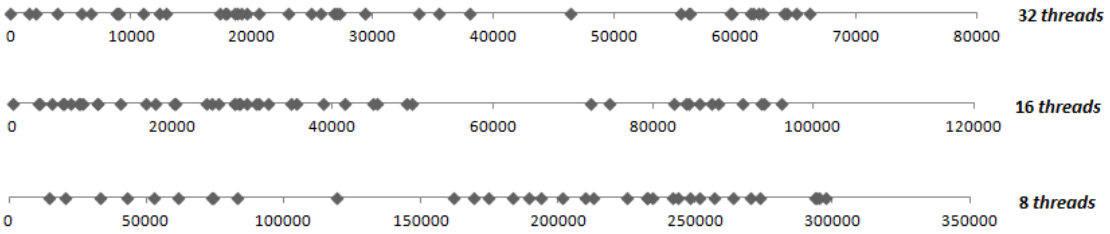

Figura 5.14: Momento da injeção da falha nos registradores de predicado que resultou em erro no algoritmo bitonic sort

5.2.3 Taxa de Erros devido a Falhas nos Registradores de endereço

A figura 5.15 apresenta os resultados de injeção de falhas nos registradores de endereço. Registradores de endereço armazenam *offsets* de memória para operações em memória. Desta maneira, a taxa de erros observada será determinada pelo volume de instruções de acesso à memória utilizadas pelo algoritmo. Como este número também é ditado pelo número de comparações do algoritmo, a configuração de 32 *threads* por bloco irá apresentar um salto na taxa de erros quando comparada com as configurações de 16 *threads* e 8 *threads*. Estes resultados se assemelham com o comportamento observado no banco de registradores de dados, porém apresentando uma taxa de erros significativamente menor do que o outro banco de registradores.

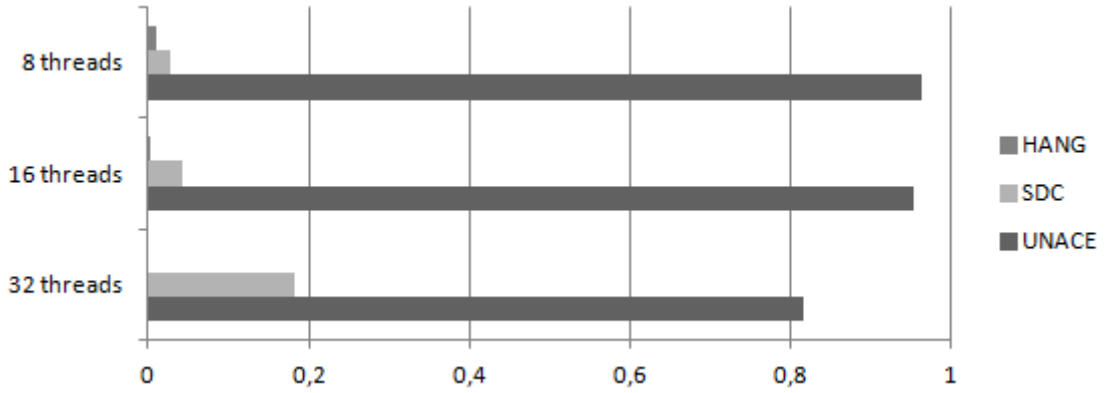

Figura 5.15: Efeitos de SEUs no banco de registradores de endereço para o algoritmo de bitonic sort

5.2.4 Discussão de Resultados

Os resultados apresentados nesta etapa de injeção de falha condizem com os resultados apresentados no estado da arte. O aumento do paralelismo de GPGPUs, alcançado pela escolha de diferentes configurações de número de blocos e *threads* a serem utilizadas na computação dos algoritmos, tem impacto direto na confiabilidade do sistema. A utilização menos eficiente dos recursos computacionais de GPGPUs pode ser explorado a fim de obter sistemas mais confiáveis em aplicações que requerem alto grau de confiabilidade, sendo necessária a determinação de um ponto ótimo em relação à eficiência na execução do algoritmo e na confiabilidade dos resultados apresentados, como demonstrado em Rech, 2013.

O modelo de injeção de falhas apresentado neste trabalho indicou ser bastante preciso com relação às taxas de erro observadas como resultado dos SEUs injetados nos bancos de registradores. Novas campanhas de injeção são necessárias a fim de explorar esta característica de confiabilidade de GPUs. As metodologias utilizadas para a aceleração do processo de injeção tornam viáveis os testes em vetores de entrada mais significativos, simulando de maneira mais real o comportamento de GPUs em aplicações paralelas.

5.3 Impacto de SEUs nos componentes da GPU

Nesta segunda campanha de injeção de falhas, monitores de acessos à memória e aos *Program Counters* da GPU são adicionados com a finalidade de analisar o impacto da falha no sistema. A cada rodada de injeção, o sinal de erro de cada monitor é avaliado, indicando que um acesso não esperado foi feito no componente. Adicionalmente, os resultados de cada aplicação são avaliados em termos de erros simples ou múltiplos, além da taxa de erros de controle que foram observados na campanha de injeção. Erros de controle remetem a indicações de acessos fora de ordem ao *Program Counter* do multiprocessador, geralmente levando a GPU a estados inesperados, tais como: atraso no término da execução do algoritmo, laços infinitos, dentre outros.

Na tabela 5.1 podemos observar os resultados de campanhas de injeção em três algoritmos: autocorrelação, *bitonic sort* e multiplicação de matrizes. Os resultados serão individualmente discutidos para cada cenário de injeção.

Tabela 5.1: Taxa de Erros na execução dos algoritmos de autocorrelação, bitonic sort e multiplicação de matrizes

Banco de registradores	Algoritmo	Correto	Erro simples	Erro múltiplo	Erro de controle	Memória Global	Memória Compartilhada	Memória Constante
Dados	Autocorrelação	77,4%	18,4%	1,2%	3%	3,4%	3%	0%
	Bitonic Sort	58,5%	21,6%	14.6%	5,3%	11,2%	5,9%	0,6%
	Multiplicação de Matrizes	59,6%	38,7%	0%	1,7%	13,2%	0%	0%
Endereço	Autocorrelação	-	-	-	-	-	-	-
	Bitonic Sort	81,7%	6,5%	11,8%	0%	5,3%	17,1%	0%
	Multiplicação de Matrizes	-	-	-	-	-	-	-
Predicado	Autocorrelação	76,8%	0%	12,7%	10,5%	22,4%	0,6%	0%
	Bitonic Sort	81%	0%	1,6%	17,4%	1%	1,3%	5,5%
	Multiplicação de Matrizes	77,4%	1,9%	16,2%	4,4%	16%	0,4%	0,5%

5.3.1 Discussão dos resultados

As campanhas de injeção nos registradores de dados resultaram nas maiores taxas de erros na GPU. No algoritmo de multiplicação de matrizes, podemos observar uma alta incidência de erros simples, com respectiva ausência de erros múltiplos em seus resultados. Esse resultado é facilmente explicado pela presença de instruções de multiplicação e adição simultâneas no *assembly* deste algoritmo, eliminando a dependência de dados entre as operações aritméticas presentes neste algoritmo. Adicionalmente, a taxa de 1,7% de erros de controle indica a principal característica deste algoritmo, que é o grande número de operações aritméticas sobre os dados de entrada do algoritmo. A baixa ocorrência de instruções de controle reflete a taxa de erros de controle do algoritmo de multiplicação de matrizes, afetando muito pouco o PC do multiprocessador. Outra característica interessante deste algoritmo é a falta de acessos inesperados à memória compartilhada, que tem seu uso limitado neste caso. Por outro lado, 13,2% dos erros da campanha de injeção refletiram acessos inesperados à memória global.

O algoritmo de *bitonic sort* apresentou a maior sensibilidade a falhas transientes em seus registradores de dados. O grande número de instruções de movimento de dados a fim de ordenar os vetores de entrada, somado com a presença de laços de controle e, consequentemente, instruções de controle, faz com índices maiores de erros de controle e de erros de dados sejam observados. Além disso, erros múltiplos apresentaram uma taxa de 14,6%, indicando a alta dependência entre dados neste algoritmo. A

dependência de dados entre *threads* também é observada pela presença de acessos inesperados às memórias compartilhada e constante, indicando comunicação entre *threads* do mesmo bloco de *threads*. A memória global, como era de se esperar, apresentou o maior índice de acessos inesperados, sendo a mais utilizada neste algoritmo. Outro resultado interessante diz respeito ao tipo de erro produzido quando o SEU manifesta um acesso inesperado à uma das memórias da GPU. Erros múltiplos são observados quase que na sua totalidade nestes casos, em detrimento aos erros simples que ocorrem na maioria das injeções em que a memória realiza somente acessos esperados.

Já o algoritmo de autocorrelação demonstrou ser o algoritmo menos sensível a SEUs em registradores de dados. Verificando a descrição do algoritmo, na figura 5.6, podemos observar que o número de operações a serem realizadas por laço do *loop* principal é limitado pelo número de *therads*, fazendo com que tenhamos casos com *threads* inoperantes residindo no *warp*. Assim, como já explicado em resultados anteriores, a taxa de erros cai, devido ao número menor de registradores a serem de fato utilizados por cada *warp*. Por fim, acessos inesperados na memória global produziram erros simples e múltiplos, enquanto que acessos inesperados na memória compartilhada produziram somente erros de controle do sistema. O *assembly* deste algoritmo não apresenta instruções explícitas de acesso à memória compartilhada, sendo que seus acessos são gerados por instruções de controle, como a ISET, perturbando desta maneira o fluxo de controle do sistema.

As injeções de SEUs nos registradores de predicado demonstram uma maior sensibilidade para algoritmos que realizam um volume maior de instruções de controle. Por suas características, o algoritmo de *bitonic sort* possui mais instruções de controle que os outros algoritmos avaliados, apresentando uma taxa maior de erros de controle. Os testes com o algoritmo de autocorrelação também apresentaram uma taxa elevada de erros de controle pelo mesmo motivo. Erros de dados, por outro lado, se manifestam mais nos algoritmos de autocorrelação e multiplicação de matrizes por contarem com mais instruções algébricas e de movimentação nos seus dados intermediários, sendo mais afetadas por uma falha causada em um dos laços de execução. Assim, temos uma presença bastante elevada de erros múltiplos nestes algoritmos, pois a falha irá afetar um conjunto muito maior de dados do que quando presente nos registradores de dados.

O mesmo padrão é encontrado nos acessos inesperados às memórias da GPU. Algoritmos com presença massiva de instruções de controle irão afetar mais as memórias compartilhada e constante, que são utilizadas respectivamente para comunicações entre *threads* e caches de dados, justificando seus altos índices no algoritmo de *bitonic sort.* Adicionalmente, os algoritmos de multiplicação de matrizes e autocorrelação apresentam altos índices de acessos errôneos à memória global, justificado pela maior presença de instruções que utilizam operandos desta memória.

Por fim é feita a analise do comportamento do sistema em presença de falhas transientes nos registradores de endereço. Somente o algoritmo de *bitonic sort* possui instruções de acesso à estes endereços, sendo a única que possui dados de injeção de falhas nesta etapa. Como era de se esperar, erros múltiplos de dados são os mais presentes, com nenhum erro de controle sendo observado. Uma perturbação no valor destes registradores irá causar um acesso errado na busca de dados na memória, o que causa efeitos evidentes na computação do algoritmo. A memória compartilhada é a mais afetada justamente pelas constantes comparações realizadas entre seus valores pelas

threads responsáveis por sua computação, tornando este algoritmo extremamente suscetível a experimentar erros quando perturbações ocorrem em seus registradores.

6 CONCLUSÃO

Os avanços tecnológicos na indústria de semicondutores permitiram a fabricação de transistores integrados cada vez mais densos, consequentemente produzindo chips de tamanhos cada vez menores. Essa tendência faz com que transistores tenham sua confiabilidade diminuída por consequência de características da tecnologia: menores tensões de operação, menor dimensão dos transistores e menor distância entre os transistores no silício. Estas características implicam uma maior sensibilidade destas tecnologias a experimentarem falhas transientes em sua lógica, que podem resultar em perturbações em seus elementos de memória.

A constante pesquisa em termos de redução da tecnologia de fabricação de semicondutores faz com que arquiteturas tradicionais de processamento também atinjam um limite de sua capacidade computacional. Desta maneira, tornou-se necessária a pesquisa de arquiteturas paralelas com eficiente processamento de grandes volumes de dados, dando origem a arquitetura de GPUs, amplamente utilizadas em um grande número de aplicações onde a confiabilidade no resultado computado é obrigatória.

Este trabalho teve como principal objetivo a avaliação do comportamento de GPGPUs em presença de falhas transientes nos principais bancos de registradores presentes na arquitetura. Para isso, foi necessário definir um modelo de injeção de falhas com a capacidade de perturbar o sistema com tais falhas, não interferindo no fluxo de execução normal do dispositivo. A representatividade dos resultados também é um fator relevante no processo de injeção, sendo necessária a adição de componentes no modelo que garantam a execução ininterrupta e eficiente das campanhas de injeção, gerando um volume estatisticamente aceitável para a análise dos resultados.

As campanhas de injeção de falhas foram realizadas na GPGPU FlexGrip executando três aplicações distintas: multiplicação de matrizes, *bitonic sort* e autocorrelação. Resultados experimentais demonstram a forte relação entre o tipo de erro gerado pela GPU com a característica de cada algoritmo. Uma análise sobre o impacto de diferentes configurações de distribuição de recursos computacionais da GPU na confiabilidade do dispositivo é avaliada, demonstrando uma forte relação entre o grau de paralelismo atingido pelo algoritmo com a taxa de erros observada em presença de SEUs nos seus bancos de registradores. Configurações que atinjam um grande grau de paralelismo, ou seja, que utilizem eficientemente os recursos computacionais da GPU, apresentam uma alta sensibilidade a SEUs em seus registradores. Por outro lado, configurações com um baixo índice de ocupação de recursos demonstram ser mais confiáveis, produzindo menos erros em suas campanhas de injeção.

Por fim, uma análise da manifestação de SEUs em diferentes componentes da GPU demonstra que memórias podem ser afetadas por falhas transientes em registradores, sendo que uma falha que manifeste o armazenamento de um resultado errôneo em uma

das memórias da GPU pode produzir resultados ainda mais errôneos do algoritmo em execução, como demonstrado nos resultados do capítulo 5. Técnicas de detecção e correção de erros utilizadas nestas memórias, como o ECC, utilizado em modernas arquiteturas de GPU, não irão corrigir estes erros, visto que o resultado já é armazenado com um valor errado na memória, sendo que uma futura leitura nesta posição de memória não acusará erros. Sendo assim, o estudo de técnicas de tolerância a falhas em registradores de GPUs torna-se necessário.

Um dos principais problemas enfrentados na elaboração deste trabalho foi a grande demora na simulação lógica de programas para a arquitetura FlexGrip, utilizada nas campanhas de injeção de falhas. A elaboração do modelo de injeção de falhas, proposto neste trabalho, fez com que este tempo fosse eficientemente reduzido, tornando viável novas pesquisas nesta área. Novos trabalhos nesta área serão explorados no futuro, com a avaliação de técnicas de tolerância a falhas em registradores dedicadas à arquitetura de GPUs.

REFERÊNCIAS

NVIDA Corporation, 2009. Nvidia CUDA Programming guide, Version 2.3.1, Available at: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>. Acessed: November 12th, 2014.

KRUGER, J.; WESTERMANN, R. Linear algebra operators for GPU implementation of numerical algorithms. **ACM Transactions on Graphics**. vol. 3, no. 22, pp. 908–916, 39–55, 2003.

HU, Q.; XIAO, B.; FRISWELL M. Robust fault-tolerant control for spacecraft attitude stabilisation subject to input saturation. **Control Theory Applications.** vol. 5, no. 2, pp. 271–282, 2011.

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

STRANO, A.; et al. Exploiting structural redundancy of SIMD accelerators for their built-in self-testing/diagnosis and reconfiguration. **IEEE International Conference on Application-Specific Systems, Architectures and Processors.** pp. 141-148, 2011.

BAUMANN, R. C. Soft errors in advanced semiconductor devices-part I: the three radiation sources. **IEEE Electron Devices Society.** pp 17-22, 2001.

SLAYMAN, C.; LA CARTE, O. A. Soft errors-past history and recent discoveries. **IEEE Integrated Reliability Workshop Final Report**. pp. 25–30, 2010.

DIXIT, A.; WOOD, A. The impact of new technology on soft error rates. **IEEE International Reliability Physics Symposium**. pp. 5B.4.1 – 5B.4.7, 2011.

NORMAND, E. Single event upset at ground level. IEEE Transactions on Nuclear Science. vol 43, no 6. pp. 2742-2750, 1996.

RECH P.; et al. Neutron radiation test of graphic processing units. **IEEE International On-line Testing Symposium**. pp. 55-60, 2012.

BEBERG, A. L. et al. Folding@home: Lessons from eight years of volunteer distributed computing. **IEEE International Symposium on Parallel & Distributed Processing**. pp. 1-8, 2009.

HAQUE, I. S.; PANDE, V. S. Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu. **IEEE International Conference on Cluster, Cloud and Grid Computing**. pp. 691–696. 2010.

NVIDA Corporation, 2009. Next Generation CUDA compute architecture: Fermi, Version 1.1, Available at: < https://docs.nvidia.com/cuda/cuda-c-programming-guide/>. Acessed: August 20th, 2014.

ANDRYC, K.; MERCHANT, M.; TESSIER, R. FlexGrip: A Soft GPGPU for FPGAs. **IEEE International Conference on Field-Programmable Technology**. pp 230-237, 2013.

LINDHOLM, E. NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro. vol.28, no.2. pp.39-55, 2008.

NVIDIA Corporation, 2011, PTX: Parallel Thread Execution ISA, Version 2.3. Available at: <https://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf >.

DODD, P.; MASSENGILL, L. W. Basic mechanism and modeling of single-event upset in digital microelectronics. **IEEE Transactions on Nuclear Science**. Vol. 50. pp. 583-602, 2003.

RECH, P. et al. Neutron Sensitivity of Integer and Floating Point Operations Executed in GPUs. **IEEE Latin American Test Workshop**. pp 1-6, 2013.

RECH, P. et al. Neutron-Induced Soft Errors in Graphic Processing Units. **IEEE Radiation Effects Data Workshop**. pp 1-6, 2012.

RECH, P. et al. Threads Distribution Effects on Graphics Processing Units Neutron Sensitivity. IEEE Transactions on Nuclear Science. vol 60, no 6. pp 4220-4225, 2013.

RECH, P. et al. Impact of GPUs Parallelism Management on Safety-Critical and HPC Applications Reliability. **IEEE International Conference on Dependable Systems and Networks**. pp 23-26, 2014.

RECH, P. et al. GPUs Reliability Dependence on Degree of Parallelism. **IEEE Transactions on Nuclear Science**. vol 61 no 4. pp 1755-1762, 2014.

FANG, B. GPU-Qin: A Methodology for Evaluating the Error Resilience of GPGPU Applications. **IEEE International Symposium on Performance Analysis of Systems and Software**. pp 221-230, 2014.

TSELONIS, S.; DIMITSAS, V.; GIZOPOULOS, D. The functional and performance tolerance of gpus to permanent faults in registers. **IEEE International On-Line Testing Symposium**. pp 236-239, 2013.

DI CARLO, S. et al. Fault mitigation strategies for CUDA GPUs. **IEEE International Test Conference**. pp 1-8, 2013.

DI CARLO, S. et al. A Software-Based Self Test of CUDA Fermi GPUs. IEEE European Test Symposium. pp 1-6, 2013.

CHONG, D. et al. Matrix Multiplication on GPUs with On-line Fault Tolerance. IEEE International Symposium on Parallel and Distributed Processing with Applications. pp 311-317, 2011.

RECH, P. et al. An Efficient and Experimentally Tuned Software-Based Hardening Strategy for Matrix Multiplication on GPUs. IEEE Transactions on Nuclear Science. vol 60 no 4. pp 2797-2804, 2013.

XILINX, 2008. MicroBlaze Processor Reference Guide, Version 9.0. Available at: < http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf>

MODELSIM SUPPORT, 2010. Mentor Graphics. Available at: <http://www.model.com/content/modelsim-support>

NVIDIA, 2015, Cuda Compiler Driver NVCC, Version 7.5. Available at: < http://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf>

AAMODT, T., 2012, GPGPU-Sim. University of British Columbia. Available at: < http://www.gpgpu-sim.org/ >

NVIDIA Corporation, cuobjdump - Application Note, Version 04. Available at: < http://www.ece.lsu.edu/koppel/gp/refs/cuobjdump.pdf >

BATCHER, K. E. Sorting Networks and their Applications. AFIPS '68 Proceedings of spring joint computer conference. pp 307-314, 1968.

ANEXO 1

Em anexo se encontra o artigo publicado no Latin-American Test Workshop (LATW) no ano de 2014.

Implementation and Experimental Evaluation of a CUDA Core under Single Event Effects

Werner Nedel, Fernanda Kastensmidt
Instituto de Informática - PPGC - PGMICRO
Universidade Federal do Rio Grande do Sul (UFRGS)
Av. Bento Gonçalves 9500, Porto Alegre - RS - Brazil
{wmnedel, fglima} @ inf.ufrgs.br

José Rodrigo Azambuja
Centro de Ciências Computacionais - PPGCOMP
Universidade Federal do Rio Grande (FURG)
Av. Itália Km 8, Rio Grande - RS - Brazil
jrazambuja@furg.br

***Abstract*—Graphic Processing Units have become popular in a broad range of applications due to their high computational power and low prices. Among the applications are the safety critical ones, where fault tolerance is mandatory. This paper presents the implementation of a CUDA core, the main processing core of a GPU and its evaluation under Single Event Transients. Results will be able to help designers to develop the required fault tolerant techniques in an effective fashion.**

Keywords—graphic processing unit, fault tolerance, single event transients

I. INTRODUCTION

Graphic Processing Units (GPUs) [1], [2] are specialized massively parallel many-core processors that take advantage of Thread-Level Parallelism (TLP) to handle computations for computer graphics. GPUs dedicate the majority of the silicon area to data processing with only a small portion assigned to data caching and flow control circuitry, which makes them suitable for solving computer-intensive problems, in spite of general purpose CPUs, which embrace sequential data flow structure with the use of Instruction-Level Parallelism (ILP).

The ability to rapidly manipulate high amounts of memory locations and execute several elementary tasks in parallel at high speeds makes GPUs more effective than CPUs for algorithms in which large blocks of data can be processed in parallel. Examples of algorithms where GPUs excel are oil exploration, analysis of air traffic flow, medical image processing, linear algebra, statistics, 3D reconstruction, and stock options pricing determination [3].

The rapid proliferation of GPUs due to the advent of significant programming support has brought programmers to use them in safety critical applications, like automotive, space and medical [4-6]. In these applications, the use of fault tolerant techniques is mandatory to detect or correct faults, since they must continue to work properly despite the existence of faults. However, the reliability of a GPU system is still an open issue.

The increasing demand for computational has pushed GPUs to be built in cutting-edge technology down to 28nm fabrication process for the latest NVIDIA devices with operating clock frequencies up to 1GHz. The increases in operating frequencies and transistor density combined with the reduction of voltage supplies have made transistors more susceptible to faults caused by radiation interference due to reduced threshold voltages, reduced node capacitances and tightened noise margins [7]. Such faults, mainly caused by energized particles, make the newest GPUs potentially prone to experience radiation-induced errors [8], [9], even on terrestrial applications running at ground level, where neutrons are the main sources of soft errors [10].

Single Event Effect (SEE) is known as one of the major effects that may occur when a single radiation ionizing particles strikes the silicon. This effect can be destructive and non-destructive. An example of destructive effect is Single Event Latchup (SEL) that results in a high operating current, above device specifications, that must be corrected by a power reset. Non-destructive effects are defined as a transient effect fault provoked by the interaction of a single energized particles in drain PN junction of the off-state transistors. This strike may upset a node of the circuit and thus generate a transient voltage pulse that can be interpreted as internal signals and lead to an erroneous result [11]. When the transient pulse occurs in a memory element, such as a register, it is classified as Single Event Upset (SEU). When the particle hits a combinational element, inducing a pulse in the combinational logic, the upset is classified as Single Event Transient (SET).

Compute Unified Device Architecture (CUDA) is a model created by NVIDIA and implemented by its GPUs that gives program developers direct access to the virtual instruction set and memory. The CUDA core, on the other hand, is the hardware module responsible for executing threads in the GPU architecture. A GPU is mainly built on top of a large number of CUDA cores and therefore the CUDA core can be considered as the building block of a GPU.

In this paper, the authors will implement a CUDA core prototype described in Hardware Description Language (HDL) and will perform an experimental fault injection campaign to simulate the effects of SEUs and SETs on it. Results will show the most sensitive parts of the CUDA core that must be protected by fault tolerance techniques in order to be used in safety critical applications, even at ground level.

II. BACKGROUND

A. RELATED WORK

The literature has a few works that have implemented a GPU in HDL languages for FPGAs. The FlexGrip project [14] examines the implementation of a soft G80 GPU in a Virtex-6 FPGA. This architecture is able to run CUDA compiled objects without hardware recompilation. The results presented in the work indicated a considerable gain, up to 30x, when compared to MicroBlaze processor. However, the G80 architecture was the first GPU developed with general computation purpose, still in 2006, and several improvements are taken place in new GPUs architectures [1], [2], such as Fermi, which is the architecture used as reference in the present work.

Some projects have evaluated the neutron sensitivity of GPUs. In [15] the authors analyzed the results of neutrons radiation campaigns on a GPU, presenting a complete guide for accelerated radiation experiments, while in [16] the experiments are focused on the sensitivity of integer and floating point operations executed in GPUs. It shows that there is a strong dependence of the neutron-induced error rate of different codes on data type, and operations are more reliable when executed on floating point data with respect to integer.

Radiation injection campaigns are very important to evaluate the sensitive of GPU internal modules and instructions, but to the best of our knowledge, so far no fault injection campaign has been performed in the HW implementation. This work presents the implementation of a CUDA core and a simulation fault injection campaign in the integer and floating point units.

B. GPGPUs

A GPU is a many-core device with a substantial parallel processing capability. The GPU consists of an array of multiprocessors enabling the device to execute lots of threads in parallel. The majority of the silicon area is dedicated to data processing units with only a small portion assigned to data caching and flow control circuitry. Its data processing units can be seen in Fig. 1.

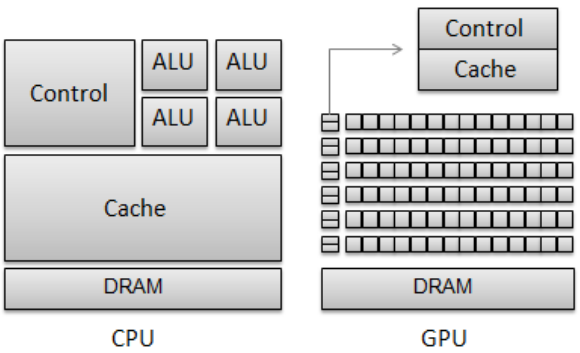

Figure 1: Data processing units in a GPU

Such architecture enables the GPU to solve streaming compute-intensive problems [1] more efficiently than when compared to General Purpose Processors (GPPs). On the other hand, GPUs are designed to optimize the execution time of a given task, and to do so they need robust control units to schedule thread execution sequentially. Large cache memories are also needed to minimize the latency of instruction and data accesses. Conversely, GPUs have several computing units which instantiates and executes a large amount of instructions at the same time, so a very simple control unit is sufficient to schedule the work between them.

CUDA is the name of the programming model created by NVIDIA that enables the GPU to be programmed with a variety of high level programming languages. This programming model is the key to use GPUs as General Purpose Graphics Processing Units (GPGPUs). The host program launches several kernels organized as a grid of thread blocks. The thread blocks are partitioned and grouped into warps, which is the smallest set of simultaneous operations.

GPUs are divided into various computing units, called Streaming Multiprocessors (SM), which are responsible for executing several threads in parallel, in a Single Instruction Multiple Data (SIMD) fashion. Every thread executes with dedicated memory locations, thus avoiding complex resource sharing or long pipelines. A very simple control unit is then sufficient to schedule threads execution [3]. The structure of a GPU is different from the typically CPU one, where sophisticated control units are needed to schedule complex thread execution sequentially or in parallel with other executions, eventually taking advantage of the presence of multiple arithmetic logic units. Moreover, on a CPU, large cache memories must be provided to minimize the instruction and data access latencies of large complex applications.

The SM is composed of various computing cores named CUDA cores, load/store units and Special Function Units (SFU), which can be seen in Fig. 3 as Core, LD/ST and SFU, respectively. Each CUDA core has a pipeline integer Arithmetic Logic Unit (ALU) and a Floating Point Unit (FPU), which can be seen in Fig. 2 as INT Unit and FP Unit, respectively. The load/store units supply memory access to all the threads running in the SM. The SFUs execute special instructions such as sin, cosine, reciprocal, and square root. It is decoupled from the dispatch unit, and therefore does not interfere with the other execution units. These processing cores are then connected by an interconnect network and fed with data from an instruction cache and a register bank.

As one can see in Fig. 2, the CUDA core is the main processing node of a GPU. It is composed not only by the Integer and FP units, but also by a dispatch port, operand collector, and result queue. The dispatch port is responsible for distributing the control signals from the SM into both processing units or, in other words, designating each instruction to its processing unit. The operand collector is a hardware module responsible for interface with the register bank and thus providing the required data to be processed. The result queue performs a sort of temporal register cache for the functional units. The INT unit executes all the integer operations, while the FP unit deals with the floating point instructions, using the IEEE 754-2008 floating-point standard [1]. Finally, the result queue avoids conflicts of writes in the register file. Also, some operations do not need to write back

into the register file, producing only an intermediate result which is consumed by another operation. So, in that situation, if the two operations are scheduled close enough, the output value can be forwarded to the input of next operation, avoiding an extra access into the register file.

In a GPU, each thread may dispose of up to tens of megabits of internal registers and has access to shared memory, which is necessary to avoid multiple accesses to the DRAM performed by threads executing operation on the same data. Usually, on a GPU, threads do not interact with each other to minimize latency and waiting delays and to take full advantage of the GPU parallelism. Thus, a small stand-alone computing unit executes each thread, and the GPU scheduler is needed just to synchronize all threads and to check if execution has been completed. The GPU physical design may then be viewed as a group of several isolated elementary computing units (each one executing a thread) whose corruption will generate an output error in the threads that is being executed but will not affect other units.

III. CUDA CORE IMPLEMENTATION

Field Programmable Gate Arrays (FPGAs) are highly specialized devices that offer application-specific customization to designers, which includes Block RAM (BRAM) memory, Digital Signal Processors (DSPs), and Combinational Logic Blocks (CLBs) configured through Look-Up Tables (LUTs). Such devices offer low time-to-market and have been used in safety-critical applications in the past with success [12]. Due to these characteristics, we have chosen the XC5VLX110T part Virtex5 FPGA to implement the CUDA core, which contains a total of 69,120 slice LUTs and 5,905 slice registers.

In order to implement the CUDA core, we started by selecting the instruction set to be supported. Taking [13] into consideration, we chose for the INT unit the following instructions: IADD, IADD32I, IMUL, IMUL32I, IMAD, ISCADD, ISETP, and ICMP. For the FP unit, we have chosen the following instructions: FADD, FADD32I, FMUL, FMUL32I, FCMP, MUFU, DADD, DMUL, and DFMA. By implementing this set of instructions, we are able to support a large set of applications to be executed, by means of providing addition and multiplication with different types of address mode and floating point precision. The INT unit was implemented in a 5-stage pipeline, while the FP unit was implemented in a 3-stage pipeline.

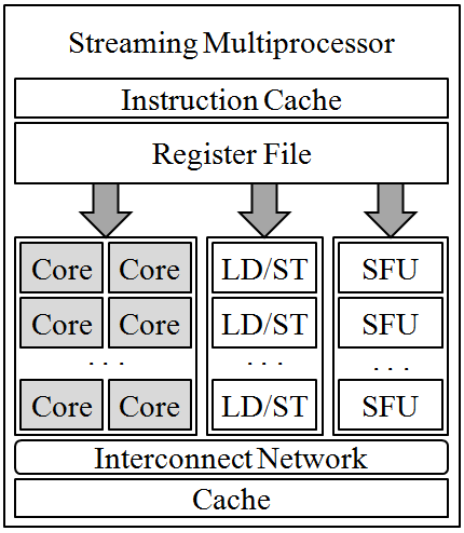

Figure 3: Streaming Multiprocessor Structure

Table I shows the area of the complete CUDA core. The INT Unit requires 2,098 slice LUTs and 1,812 slice registers and, resulting in 2.6% and 3.0% in the FPGA's fabric occupation, respectively. Additionally, the FP Unit requires 1,258 slice LUTs and 1,306 slice registers, resulting in 1.8% and 1.9% in the FPGA's fabric occupation, respectively. The whole system has been implemented using only fabric logic, meaning that no hardware accelerator, such as DSPs, has been used. The complete CUDA core, including dispatch port, operand collector and a two-word result queue requires 3,656 slice LUTs and 3,417 slice registers, which represent 5.3% and 4.9% occupation of the FPGA fabric. Such results mean that the currently used FPGA could support up to 9 CUDA cores.

TABLE I. AREA AND FREQUENCY RESULTS FOR THE VIRTEX5 FPGA IN SLICE LUTS AND REGISTERS

	INT Unit (occupation)	**FP Unit (occupation)**	**CUDA Core (occupation)**
Slice LUTs	2,098 (2.6%)	1,258 (1.8%)	3,656 (5.3%)
Slice Registers	1,812 (3.0%)	1,306 (1.9%)	3,417 (4.9%)
Fequency (MHz)	100	300	100

IV. FAULT INJECTION EXPERIMENTAL RESULTS

In order to evaluate both the effectiveness and the feasibility of the CUDA core, we implemented a benchmark containing all the instruction implemented and supported by the CUDA core with all possible operating modes (IADD with a normal and negated operand, IMULT with signed and unsigned, etc). Instruction's internal signals have been translated to our implementation and used as inputs. It is important to notice the program memory feeds the SM with instruction, which then provides the CUDA core operand and control signals.

In order to perform the fault injection campaign, faults were injected in the signals of the implemented CUDA core (including registered signals), one fault at each program execution. The SEU and SET types of faults were injected directly in the CUDA core VHDL code by using the Xilinx's iSim simulator. SEUs were injected in registered signals and SETs will be injected in combinational signals. Fault durations will last in the signal for one and a half clock cycle, so that we can avoid latch window masking. The fault injection campaign was performed automatically. At the end of each execution, the results were compared to the expected correct values.

The fault injection campaign comprehended 40,000 faults and analyzed the results by dividing faults between the faults that caused an error in the CUDA core (*Incorrect Result*) and the ones masked by the logic. The system was simulated with a clock frequency of 100MHz (period of 10ns) and most of the 5,841 signals describing were be upset.

Results from the fault injection campaign are presented in Table II. As one can see, 18.4% of the faults injected into the integer unit (INT Unit) have resulted in a wrong result, while 21.2% of the faults injected into the floating point unit (FP Unit) have resulted in a wrong result. When combined, 19.8% of fault injected in the CUDA core have corrupted the output results. Such results prove that GPUs must be hardened by means of fault tolerance techniques in order to be used in harsh environments.

TABLE II. INTEGER AND FLOATING POINT UNITS UNDER SET AND SEU FAULTS

	Faults Injected	**Correct Results**	**Incorrect Results**
INT Unit	20,000	16,313 (81.6%)	3,687 (18.4%)
FP Unit	20,000	15,746 (78.8%)	4,254 (21.2%)
Total	40,000	32,059 (80.2%)	7,491 (19.8%)

V. CONCLUSIONS AND FUTURE WORK

In this paper, the authors presented the implementation of a CUDA core in a SRAM-based FPGA with an occupation of 3.2% slice LUTs and 10.1% slice registers of a Virtex5 FPGA. Such results allow future works to implement up to 9 CUDA cores in a single FPGA. At the cost of 5 CUDA cores, one could still implement a GPU with 2 SMs, each with 2 CUDA cores, being able to test case-study applications parallelized and adapted to GPUs.

A fault injection campaign has been proposed to test the proposed implementation under SETs and SEUs with a case-study benchmark. The results will be able to guide designers into developing fault tolerant techniques for GPUs with increased efficiency and lower costs in time and performance degradation.

In the future, the authors intend to implement the entire SM structure and perform several fault injection campaigns, including a fault injection by simulation at RTL level, by injecting faults in the FPGA's configuration bitstream, and by irradiating the FPGA with neutron and Cobalt-60. By doing so, we will be able to run CUDA compiled code and verify how the GPU responds to the effects of radiation.

REFERENCES

[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 Cores 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 evices-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] H. Quinn, P. Graham, K. Morgan, Z. Baker, M. Caffrey, D. Smith, R. Bell, "On-orbit results for the xilinx virtex-4 FPGA," IEEE Radiation Effects Data Workshop, pp. 1-8, 2012.

[13] Asfermi: an assembler for the NVIDIA Fermi instruction set [online]. Available: https://code.google.com/p/asfermi/

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

[15] 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.

[16] P. Rech, C. Aguiar, C. Frost, L. Carro, "Neutron sensitivity of integer and floating point operations executed in GPUs", IEEE Test Workshop (LATW), pp. 1-6, 2013.