FTGPGPU - Genel Amaçlı Grafik İşlemci Birimi Uygulamaları İçin Donanım Hatası Toleransı Analizi

Program Kodu:3501

Proje No:119E011

Proje Yürütücüsü:

Dr. Öğr. Üyesi IŞIL ÖZ

Bursiyer(ler)

ÖMER FARUK KARADAŞ

BUSENUR AKTILAV

EMRE DİNÇER

ERCÜMENT KAYA

BURAK TOPÇU

NİSAN 2022

ANKARA

TÜBİTAK

Bu belge, 5070 sayılı Elektronik İmza Kanununa göre Güvenli Elektronik İmza ile imzalanmıştır.

ÖNSÖZ

TÜBİTAK 3501 - Kariyer Geliştirme Programı, 119E011 numarası ile desteklenen proje kapsamında genel amaçlı grafik işlemci birimi (GPGPU) uygulamalarının bölgesel geçici donanım hatası hassasiyetlerini belirleyen ve hata yayılımını analiz eden bir araç geliştirilmiş, bu araç kullanılarak farklı GPGPU uygulamalarının hata hassasiyet özellikleri detaylı olarak analiz edilmiştir. Projemizin bu aşamasında yapılan çalışmalar bir uluslararası çalıştayda (*WAICA-HIPEAC 2020*) ve bir ulusal konferansta (*BASARIM 2020*) sözlü olarak sunulmuş, SCI endeksli uluslararası bir makalede (*Journal of Supercomputing*) yayınlanmıştır. Ayrıca bölgesel hata hassasiyeti gözleminden yola çıkılarak geliştirilen yaklaşık hesaplama yöntemleri çalışması da SCI endeksli uluslararası bir dergide ilk revizyon sürecini geçmiş, değerlendirme aşamasındadır. Bu analizler neticesinde elde edilen veriler kullanılarak program karakteristiklerinden hata hassasiyetini gösteren değerlerin tahminlenmesini sağlayan regresyon ve sınıflandırma makine öğrenmesi modelleri geliştirilmiştir. Bu çalışma uluslararası bir konferansta (*PDP 2022*) tam metin bildiri olarak kabul edilmiş ve sunulmuştur. Hata enjeksiyon aracımız kullanılarak derleyici tabanlı bir seçimli hata toleransı yöntemi tasarlanmış ve geliştirilmiştir. Güvenilirlik ihtiyacı olan GPGPU uygulamaları için performans-güvenilirlik takas analizi yapılmıştır. Bu kapsamdaki çalışmalar proje yürütücüsünün FTGPGPU projesi ile dahil olduğu CERCIRAS COST aksiyonu tarafından düzenlenen *CERCIRAS* çalıştayında basılmak üzere kabul edilmiştir ve sunulmuştur. Hata toleransı yöntemi olarak sunulan çoklamanın farklı yöntemlerle genişletilmesi ile hazırlanan bir makalenin uluslararası bir dergiye gönderilmesi için çalışmalar devam etmektedir.

Projede bursiyer olarak çalışan 2 lisans öğrencisi mezun olarak akademi ve endüstride çalışmalarını sürdürmekte olup, 3 yüksek lisans öğrencisinin önümüzdeki yıl içerisinde tez çalışmalarını bitirmeleri öngörülmektedir.

Proje kapsamında geliştirilen tüm kodlar, araştırma grubumuzun `https://github.com/parsiyte` github linkinde açık kaynak kodlu olarak yayınlanmıştır.

Dr. Öğr. Üyesi Işıl Öz

İzmir, Nisan 2022

İçindekiler

ÖNSÖZ **i**

ÖZET **vi**

ABSTRACT **vii**

1 GİRİŞ **1**

2 LİTERATÜR ÖZETİ **4**

2.1 GPU Uygulamaları İçin Geliştirilen Hata Hassasiyeti Belirleme Araçları 4

2.2 Uygulamaların Hata Toleransının Tahminlenmesi 7

2.3 Seçimli Hata Toleransı Yöntemleri 8

3 GEREÇ VE YÖNTEM **10**

3.1 GPGPU Uygulamaları İçin Bölgesel Hata Hassasiyet Analizi Aracı Geliştirilmesi . . 10

3.1.1 Bölgesel Hata Hassasiyet Aracı 10

3.1.2 Hata Yayılımı Takibi 13

3.2 GPGPU Uygulamalarının Hata Hassasiyetlerinin Tahminlenmesi 15

3.2.1 Performans Metriklerinin Toplanması 17

3.2.2 Metriklerin Seçilmesi 20

3.2.3 Hata Tahminleme Modellerinin Oluşturulması 21

3.3 Seçimli Hata Toleransı Yöntemi Geliştirilmesi 23

3.3.1 LLVM Derleyici İskeleti 24

3.3.2 Derleyici Direktifi 28

3.3.3 Çıktı Çoklaması 29

3.3.4 Kernel Fonksiyonunun Çoklanması 30

3.3.5 Çoğunluk Oylaması Gerçeklemesi 30

4 BULGULAR **31**

4.1 Test Ortamı 31

4.1.1 GPU Ortamları 31

4.1.2 GPGPU Benchmark Uygulamaları 31
4.2 Test Sonuçları 31
4.2.1 Bölgesel Hata Hassasiyet Analizi Aracı ile GPGPU Uygulamalarının Hata Hassasiyetlerinin Elde Edilmesi 31
4.2.2 GPGPU Uygulamalarının Hata Hassasiyetlerinin Tahminleme Sonuçları . . . 46
4.2.3 Seçimli Hata Toleransı Yöntemi ile Elde Edilen Performans ve Güvenilirlik Sonuçları 50

5 SONUÇ VE TARTIŞMA **56**

Şekil Listesi

1 Bölgesel hata enjeksiyonu aracı. 11
2 Hata enjektörünün akış diagramı. a) Belirli bir kesme noktasındaki hata enjeksiyonu. b) Özel bir koşul altında hata enjeksiyonu. 12
3 3mm uygulamasının akış diyagramı. 15
4 Program değişkenlerindeki bozulmaların görselleştirilmesi. 16
5 Hata tahminleme yapımızın genel işleyişi. 17
6 Çoklamalı çoklu iş parçacığı yapımız. 24
7 Temel LLVM bileşenleri Lattner (2011). 25
8 CUDA kodunun Clang ile derlenmesi. 25
9 Kernel fonksiyonlarının yanlış hesaplanan eleman sayılarına ait boxplot gösterimleri. 35
10 Mutlak hata ortalama (Mean Absolute Error (MAE)) histogram değerleri. 40
11 Mutlak hata ortalama (Mean Absolute Error (MAE)) histogram değerleri. 41
12 Mutlak hata ortalama (Mean Absolute Error (MAE)) histogram değerleri. 43
13 Simülatör metrikleri ve hata oranları arasındaki Spearman ve Pearson korelasyon sonuçları. 47
14 Nsight Compute tool metrikleri ve hata oranları arasındaki Spearman ve Pearson korelasyon sonuçları. 47
15 Kernel fonksiyonları için SDC değerleri. 50
16 Çoklanmış uygulamaların normalleştirilmiş yürütme süreleri. 52
17 Çoklanmış yürütmelerin SDC oranleri ve yürütme sürelerindeki yüzdelik değişim. . . 53
18 Her fonksiyonun yürütülme profili. 54

Tablo Listesi

1 Simülatörden toplanan metrikler. 18
2 Profil oluşturucudan toplanan metrikler. 19
3 Polybench benchmark setinden seçtiğimiz uygulamalar. 32
4 Maskelenmiş hatalar için tahminleme sonuçları. 48
5 2-sınıf sınıflandırma için tahminleme sonuçları. 49
6 3-sınıf sınıflandırma için tahminleme sonuçları. 49
7 Kernel fonksiyonları için yürütme süreleri. 51

ÖZET

Genel amaçlı hesaplamalar için grafik işlemci birimlerinin (GPGPU) kullanımı, donanım hatalarının kritikliğini arttırmakta, programların geçici hata hassasiyetini değerlendirmek ve uygun hata toleransı tekniklerini kullanmak daha önemli hale gelmektedir. Hataya en hassas program bölgelerinin korunması yoluyla, hem performansı, hem de güvenilirliği hedefleyen sistemler için ayrıntılı bölgesel hata hassasiyeti analizi çok önemlidir. Bu projede, GPGPU uygulamalarının geçici donanım hatası hassasiyetinin ölçülmesi, analiz edilmesi ve bu analizlerin sonuçlarının program özellikleri ile ilişkilendirilmesi, seçimli hata toleransı yöntemi geliştirilmesi yoluyla kullanılması amaçlanmıştır.

Projenin ilk katkısı, GPGPU uygulamlarının geçici hata hassasiyetlerinin bölgesel olarak belirlenmesi için yazılım ile donanım ilişkisini sağlayacak şekilde assembly seviyesinde hata ayıklayıcı tabanlı bir hata enjeksiyonu ve hata yayılımı analizi aracı geliştirilmesidir. Bu araç kullanılarak farklı yapıdaki, farklı özelliklere sahip GPGPU programlarının belirlenen kod bölgelerine hata enjeksiyonu sağlayan deneyler yapılmış, kod bölgelerinin hata hassasiyetleri ve oluşan hatanın program süresince farklı veri yapılarına yayılımı incelenmiştir.

Projenin ikinci katkısı, GPGPU program kod parçalarının özellikleri ile bu kodlar çalışırken meydana gelebilecek hatalara hassasiyetleri arasındaki ilişkinin incelenmesidir. GPGPU programlarındaki kod parçacıklarının performans ve mimari özellikleri profilleme ve simulasyon yöntemleriyle elde edilmiş, ilk adımda geliştirilen hata enjeksiyonu aracıyla belirlenen kod parçalarına hata enjekte ederek uygulanan deney sonuçlarında sessiz veri bozunumu, çökme ve doğru çalışma durumları belirlenmiştir. Program özellikleri-hata hassasiyeti ikilisi arasındaki ilişki incelenerek program özellikleri verilen bir GPGPU uygulamasının hata hassasiyet değerleri makine öğrenmesi yöntemleriyle tahmin edilmiştir. Geliştirilen tahminleme modelleriyle sessiz veri bozunumu için %82, çökme durumları için %87, doğru çalışma durumları için %96 doğruluk oranlarıyla tahminleme başarısı sağlanmıştır.

Projenin üçüncü katkısı, hataya daha hassas kod bölgelerinin çoklanmasına dayalı seçimli hata toleransı yöntemi geliştirilmesidir. Program geliştirici veya kullanıcı tarafından kaynak kodda işaretlenen kod bölgelerinin çoklanması şeklinde gerçeklenen derleyici seviyesinde geliştirilen hata toleransı yapısı, belirtilen kernel fonksiyonlarının çoklanmasını artıklı kernel fonksiyonu olarak veya tek kernel fonksiyonu altında artıklı iş parçacığı olarak veya CUDA stream tekniği ile mümkün kılmaktadır. Böylece uygulamanın paralellik ve veri kullanımı özelliklerine göre farklı çoklama yürütme durumları seçilebilmekte, kaba taneli (coarse-grained) bir yapıda çıktı kontrolü ile performanslı bir şekilde çoklama sağlanmaktadır.

Anahtar kelimeler: Geçici donanım hatası güvenilirliği, GPU mimarileri, GPGPU uygulamaları.

ABSTRACT

As the use of graphics processing units for general-purpose calculations (GPGPU) increases the criticality of the hardware errors, it becomes more important to evaluate the transient error vulnerability of the programs and to perform appropriate fault tolerance techniques. Detailed regional soft error vulnerability analysis is essential for systems targeting both performance and reliability, by protecting the most vulnerable program regions. In this project, we aim to measure and analyze soft error vulnerability of GPGPU programs, and based on the analysis results, we correlate error characteristics with program features and develop a selective fault tolerance method.

The first contribution of the project is the development of an assembly-level debugger-based fault injection and error propagation analysis tool that enables regional soft error vulnerability analysis by associating software code regions and hardware components. By utilizing the tool, we carry out fault injection experiments by targeting the determined code regions of GPGPU programs with different structures and different features. We evaluate soft error vulnerability of the target code regions and error propagation through the data structures during the faulty program execution.

The second contribution of the project is the analysis of the relationship between the GPGPU program features and their vulnerability to soft errors. We obtain the performance and architectural features of the code snippets in GPGPU programs, and perform fault injection experiments by utilizing our fault injection tool to collect silent data corruption, crash and correct execution rates. By examining the relationship between program features and error vulnerability rates, we predict the error vulnerability values of a GPU application by machine learning methods. Our prediction models achieve prediction accuracy rates of 96.6%, 82.6%, and 87% for masked fault rates, SDCs, and crashes, respectively.

The third contribution of the project is the development of a selective fault tolerance method based on the redundancy of more vulnerable code regions. Our compiler-level fault tolerance framework performs redundant multithreading for the code regions marked in the source code by the program developer or the user, and enables the redundant execution of the specified kernel functions as a redundant kernel function or as a redundant thread under a single kernel function or with the CUDA stream technique. Thus, the target execution can be configured with different redundant execution schemes according to the parallelism and data usage characteristics of the application, and the redundancy is maintained in a high-performance manner with coarse-grained output control.

Keywords: Soft error vulnerability, GPU architectures, GPGPU applications.

1 GİRİŞ

Modern işlemci teknolojisinde transistör boyutlarının gittikçe küçülmesi ve transistörlerin çok daha hızlı frekanslarda çalışması nedeniyle bilgisayar sistemlerindeki geçici hata oranları artmaktadır. Alfa tanecikleri, kozmik ışınlar, termal nötronlar gibi çevresel faktörlerin etkisiyle ortaya çıkan geçici donanım hatalarının bir veya daha fazla bit yapısında kendini göstermesiyle tekil değer değişimi (Single Event Upset-SEU) veya çok bit değişimi (Multi-bit Upset-MBU) oluşmaktadır (Shivakumar vd. (2002); Mukherjee (2008)). Geçici hata, bellekte tutulan bir bit verisini bozarsa, veri güncellenene kadar bu veriye erişen tüm işlemler de hatalı sonuçlar üretebilecektir.

Düşük maliyet ve güç tüketimleriyle, grafik işlemci birimleri (GPU) son yıllarda oldukça popüler olmuşlardır (Owens vd. (2007)). İşlemlerin kısa sürede tamamlanmasından (latency) ziyade zaman birimde tamamlanan iş miktarının (throughput) yüksek olması prensibine dayalı bu mimariler, özellikle çok sayıda veri içeren ve veriler üzerinden hesaplama gerektiren uygulamaların performansını önemli ölçüde arttırmaktadır. İlk zamanlarda milyarlarca pikselin işlenmesine dayalı olan grafik uygulamalarına yönelik olarak geliştirilseler de kullanım alanları gittikçe artmaktadır. Çok sayıda hesaplama gerektiren yüksek performanslı hesaplama uygulamaları (HPC applications) ve çok sayıda veriyi gerçek zamanlı olarak işleyerek hızlı sonuç üretimine gereksinim duyan gömülü sistem uygulamaları (real-time embedded applications) da GPU mimarilerinde yüksek performanslarla çalıştırılmaktadır. İmge ve video işlemeye dayalı grafik uygulamalarında, tüm piksellerin kullanıcı tarafından fark edilmemesi ve piksellerin sonraki çerçevede (frame) tekrar hesaplanması sebebiyle piksel değerlerinin yanlış hesaplanması kritik bir önem taşımamaktadır. Bu yüzden bilgisayar sistemlerinde oluşan geçici donanım hatalarından kaynaklanan hesaplama yanlışlıkları bu tür uygulamalar için fazla önemli olmayabilmektedir. Bu sebeple ilk üretilen GPU mimarilerinde genel olarak bu tür hatalara karşı mimari üzerinde bir önlem alınması tercih edilmemiştir. Grafik uygulamalarının hata hassasiyetleri genel olarak düşük olmasına rağmen yüksek performanslı hesaplama uygulamaları ve gömülü sistemlerde, güvenilirlik (reliability) oldukça önem taşımaktadır (otomobil, havacılık, uzay, biyomedikal uygulamalar gibi). Ayrıca teknolojik gelişmelerle transistör boyutlarının küçülmesi ve güç tüketimi limitlerinin geldiği nokta düşünülürse, GPU mimarilerindeki donanımsal hatalar artmaktadır (Haque & Pande (2010); Maruyama vd. (2010)). Bununla birlikte, GPGPU uygulamalarının güvenilirliğin önemli olduğu büyük ölçekli sistemlerde de kullanımı artmaktadır (Örneğin, Oak Ridge Ulusal Laboratuvarı'ndaki Titan süperbilgisayarı). GPU'ların kullanım alanlarının artmasıyla ve genel amaçlı grafik işlemci birimi (GPGPU) uygulamaları geliştirildikçe geçici donanım hatalarından kaynaklanan hesaplama yanlışlıkları önem kazanmış ve bunun için hem mimari üz-

erinde değişiklikler yapılmış (Örneğin, Fermi mimarisindeki ECC koruması), hem de hata toleransı belirsiz olan GPGPU uygulamalarının hata hassasiyetinin analizi ve yazılım seviyesinde hata toleransı yöntemleri geliştirilmiştir. Bütüncül hata hassasiyeti analizi ve hata toleransı yöntemleri yerine, uygulamaların hata hassasiyetlerinin bölgesel olarak incelenmesi, alınacak tedbirlerin özel program bölgelerine ve farklı hata hassasiyetlerine göre kararlaştırılması açısından önem taşımaktadır.

FTGPGPU projesinde, GPGPU uygulamalarının geçici donanım hatalarına olan hassasiyetlerinin bölgesel olarak incelenmesini sağlayan bir araç geliştirilerek hedef uygulamaların detaylı hata hassasiyet özellikleri belirlenmiş, hata yayılımları incelenmiştir. Bu hata hassasiyeti analizleri sayesinde elde edilen veriler ışığında, uygulama özellikleri ile hata hassasiyetleri arasındaki ilişki kullanılarak maliyetli hata enjeksiyonları yerine makine öğrenmesi tabanlı hata tahminleme yöntemleri geliştirilmiştir. Ayrıca yüksek performans maliyetli bütüncül hata toleransı yöntemleri yerine, bölgesel hata analizi sonuçları kullanılarak belirlenen, hassasiyeti yüksek kod bölgelerinin çoklanmasına dayalı seçimli hata toleransı yöntemi geliştirilmiştir. İlave olarak, projemizin ana amaçlarından biri olmamakla birlikte projemizin çıktılarının kullanılabilirliğini göstermek amacıyla farklı program bölgelerinde farklı yaklaşık hesaplama yöntemleri uygulayarak, bu yaklaşık hesaplamaların program çıktısına ve program performansına etkileri araştırılmıştır. Projemiz, aşağıdaki bilimsel katkıları sağlamıştır:

- GPGPU uygulamalarının farklı kod bölgelerindeki hata hassasiyetlerini belirlemek için, yazılım ile donanım ilişkisini sağlayacak şekilde assembly seviyesinde derleyici tabanlı bir hata enjeksiyonu ve hata yayılımı analizi aracı geliştirilmiştir. Bu araç kullanılarak benchmark kümelerindeki GPGPU uygulamalarında, farklı kernel fonksiyonları ve aynı kernel fonksiyonlarındaki farklı kod parçalarının hata hassasiyet analizleri yapılmıştır. Bu kapsamda yaptıklarımızın ön çalışması uluslararası *WAICA-HIPEAC* çalıştayında sunulmuş (Azın & Öz (2020)), geliştirdiğimiz aracın ilk hali ulusal *BASARIM* konferansında sunulduktan sonra (Karadaş & Öz (2020)), hata yayılımı analizi ve detaylı hata analiz sonuçları dahil edilerek *Journal of Supercomputing* dergisinde yayınlanmıştır (Öz & Karadaş (2022)).
- Projemizde gerçekleştirdiğimiz hata hassasiyeti analizleri neticesinde, olası hataların uygulamaların farklı bölgelerini farklı şekilde etkiledikleri gözlemlenmiştir. Benzer olarak, güvenilirlikle dolaylı olarak ilgili olan yaklaşık hesaplama yöntemlerinin farklı program bölgelerine uygulanmasının farklı sonuçlara sebep olacağı gözlemiyle, örnek bir uygulama alanı olarak graf algoritmalarına farklı yaklaşık hesaplama yöntemleri uygulanmış, program performansındaki ve çıktılarındaki etkileri analiz edilmiştir. Bu çalışmalarımız kapsamında hazırladığımız dergi makalesi, SCI endeksli uluslararası bir dergide ilk değerlendirmeyi geçerek revize edilmiş

versiyonu gönderilmiştir, hakem değerlendirmesi devam etmektedir.

- Belirli özelliklere sahip kod parçacıklarının belirli hata hassasiyeti seviyelerinin belirlenmesi çalışmaları kapsamında, simulasyon ve profilleme yöntemiyle toplanan program özellikleri kullanılarak hata enjeksiyonu deneyleri neticesinde elde edilen sessiz veri bozunumu, çökme ve doğru çalışma durumları tahminlenmeye çalışılmıştır. Tahminleme yöntemimizde öznitelik seçme, sınıflandırma ve regresyon operasyonları için farklı makina öğrenmesi teknikleri ve algoritmaları kullanılarak farklı modeller oluşturulmuştur. Böylece uzun süreli ve maliyetli hata enjeksiyonu deneyleri yerine az sayıda deney ile farklı uygulamaların hata hassasiyeti değerlerinin tahmin edilmesi mümkün olmuştur. Bu çalışmalarımızın sonuçları *Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)* konferansında tam bildiri olarak kabul edilmiş (Topçu & Öz (2022)), yüksek lisans öğrenci bursiyerimiz tarafından sunulmuştur.

- GPGPU uygulamalarının bölgesel hata hassasiyetlerinin belirlenmesini sağlayan aracımız kullanılarak tespit ettiğimiz hataya daha hassas kod bölgelerinin çoklanmasına dayalı seçimli hata toleransı yöntemi geliştirilmiştir. Program geliştirici veya kullanıcı tarafından kaynak kodda işaretlenen kod bölgelerinin çoklanması şeklinde gerçeklenen derleyici seviyesinde geliştirilen hata toleransı yapımız, belirtilen kernel fonksiyonlarının çoklanmasını artıklı kernel fonksiyonu olarak veya tek kernel fonksiyonu altında artıklı iş parçacığı olarak veya CUDA stream tekniği ile mümkün kılmaktadır. Böylece uygulamanın paralellik ve veri kullanımı özelliklerine göre farklı çoklama yürütme durumları seçilebilmekte, kaba taneli (coarse-grained) bir yapıda çıktı kontrolü ile performanslı bir şekilde çoklama sağlanmaktadır. Bu çalışmamızın ilk hali uluslararası *CERCIRAS* çalıştayında sunulmuş olup (Kaya vd. (2021)) farklı çoklama yöntemleri ve detaylı analizler dahil edilerek SCI endeksli bir dergi makalesi hazırlama çalışmaları devam etmektedir.

Böylece proje kapsamında tanımlanan tüm iş paketleri tamamlanmış olup, geliştirilen kodlar araştırma grubumuzun `https://github.com/parsiyte` github linkinde açık kaynak kodlu olarak yayınlanmıştır. Projenin son aşamasında proje bursiyeri olarak çalışmaya devam eden üç yüksek lisans öğrencisinin *FTGPGPU* projesindeki çalışmaları kapsamında başlayan tez çalışmaları devam etmekte olup önümüzdeki bir yıl içinde tümünün mezun olması beklenmektedir.

2 LİTERATÜR ÖZETİ

Projemizdeki çalışmalarla ilgili literatürdeki çalışmalar üç ana başlıkta toplanabilir: 1) GPU uygulamaları için geliştirilen hata hassasiyeti belirleme araçları, 2) Uygulamaların hata toleransının tahminlenmesi, 3) Seçimli hata toleransı yöntemleri

2.1 GPU Uygulamaları İçin Geliştirilen Hata Hassasiyeti Belirleme Araçları

Sheaffer vd. (2006), genel işlemci birimi tabanlı sistemler için tanımlanmış olan Architectural Vulnerability Factor (AVF) metriğini grafik işlemci birimleri açısından incelemiş ve grafik uygulamaları için hata hassasiyeti analizi için Visual Vulnerability Spectrum (VVS) kavramını önermişlerdir. VVS, grafik hesaplamaların önemli ve gerekli kalite faktörlerini göz önüne alarak üç farklı kriter belirlemiştir: Kapsam (extent), geçici hatanın kaç piksel değerini bozacağını belirlemektedir (fark edilmeyenden tüm ekranın bozulmasına kadar değişebilir). Büyüklük (magnitude), çıktının etkilenen bölgesinin ne kadar kritik olduğunu belirlemektedir (fark edilmeyenden kabul edilemeze kadar değişmektedir). Kalıcı (persistence), hatanın ne kadar süre etkileyici olacağını belirlemektedir (tek bir frame sonra geçmesinden sonsuza kadar değişmektedir). Çalışmada, GPU üzerinde çalışan grafik uygulamalarının hem AVF analizi, hem de önerilen bu kriterler açısından incelenmesi neticesinde kullanıcının hatayı anlayıp çıktıyı etkilediğini belirttiği durumlarının tespitinin VVS kriterleriyle daha anlamlı olduğu gösterilmiştir. Grafik uygulamalarının da donanım hatalarından etkilenebileceği gösterildikten sonra, çalışmanın yapıldığı tarihte henüz desteklenmeyen donanım tabanlı GPU hata toleransı yöntemleriyle ilgili bir ön çalışma yapılmıştır. Bu çalışmada, sadece grafik uygulamalarının hata hassasiyet analizleri yapılarak genel amaçlı GPU uygulamaları düşünülmemiştir.

GPGPU-SODA (Tan vd. (2011)), GPGPU mimarileri için geçici hata hassasiyeti analizi yapan ilk ve etkili çalışmalardan biridir. Çalışmada, hata hassasiyeti analizi için ilk adım olarak güvenilirlik temelli mikro mimari seviyesinde bir GPGPU simulatörü geliştirilmiş ve farklı GPU benchmark uygulamalarının farklı mimari tasarımlar için hata hassasiyetleri değerlendirilmiştir. Yazarlar, bu çalışmada GPGPU mimarileri için geliştirilmiş olan açık kaynak kodlu GPGPU-Sim simulatörüne (Bakhoda vd. (2009)), daha önceki çalışmalarında genel amaçlı işlemci (CPU) mimarileri için AVF tabanlı güvenilirlik analizi yapmak üzere geliştirmiş oldukları Sim-SODA (Fu vd. (2006)) yapısını entegre ederek bir hata hassasiyeti analizi sistemi geliştirmişlerdir. Komut seviyesinde, ince taneli (fine grained) bir güvenilirlik analizi yapılmaktadır. Güvenilirlik analizinde, register dosyası, paylaşımlı hafıza, L1 önbelleği, warp çizelgeleyicisi gibi donanım yapılarının AVF değerleri ölçülmüş, mimari optimizasyon yöntemlerinin (dinamik warp oluşturulması, her bir stream işlemcisi için iş

parçacığı sayısı belirlenmesi, warp çizelgelemesi gibi) hata hassasiyetine olan etkileri incelenmiştir. Yazarlar diğer bir çalışmalarında da bu analizlerini detaylandırmışlardır (Tan vd. (2013)). GUFI (Tselonis & Gizopoulos (2016)) de GPGPU-Sim simulatörü üzerinde geliştirilmiş AVF tabanlı bir GPU hassasiyet analiz aracı sunmaktadır. AVF tabanlı bu yöntemler, hataların uygulama üzerindeki etkilerini ve uygulamanın hata durumundaki davranışını analiz etme konusunda yetersiz kalmaktadır.

HAUBERK'te (Yim vd. (2011)) GPGPU mimarilerinin hata hassasiyeti, geliştirilen hata enjeksiyon aracıyla incelenmiş, performans etkisini azaltacak şekilde özelleştirilmiş hata bulucuları (fault detector) önerilmiştir. Geliştirilen hata enjeksiyon aracı, kaynak kodu seviyesinde mutasyon temelli olup kaynak koduna hata enjeksiyon kodunun eklenmesiyle gerçeklenmiştir. Aracın kullanımı kaynak kodunun varlığını gerektirmektedir. Hem yüksek performanslı hesaplama uygulamaları, hem de grafik uygulamaları için hata hassasiyeti karşılaştırmaları verilmiştir. Uygulamanın döngüsel (loop) özellikleri incelenerek hangi kısımlarına hata bulucuları eklenirse performans kaybının ve hata yayılımının minimum düzeyde tutulucağı, hatadan kurtulmanın (recovery) arttırılacağı belirlenmiştir.

GPU-Qin (Fang vd. (2014); Fang vd. (2016)), HAUBERK'te olduğu gibi kaynak kodlu hata enjeksiyonu yerine, assembly dili seviyesinde hata enjeksiyonu yöntemiyle GPGPU uygulamalarının hata hassasiyetlerini incelemektedir. NVIDIA GPU mimarileri için özelleştirilmiş olan CUDA programlama modelinin hata ayıklayıcısı olan cuda-gdb (Cuda-gdb, 2018) üzerinde gerçeklenen hata enjeksiyonu aracıyla farklı GPGPU benchmark uygulamalarının sessiz veri bozunumu (silent data corruption-SDC)) ve uygulama çökmesi (crash) verileri raporlanmış, hata durumları uygulamaların karakteristikleriyle ilişkilendirilmemiştir.

CAROL-FI (Oliveira vd. (2017a)), GPU-Qin'e benzer olarak hata ayıklayıcısı üzerinde geliştirilmiş bir hata enjeksiyonu aracı sunmaktadır. Ancak bu araç, GPU mimarileri ve uygulamaları için değil, Intel işlemciler için geliştirilmiş olup gdb hata ayıklayıcısı üzerinde gerçeklenmiştir. CAROL-FI yazarları tarafından yapılmış olan diğer bir çalışma (Oliveira vd. (2017b)), güvenilirliğin kaynak koduna ve mimariye göre değiştiğini göstermek için Xeon Phi mimarisini incelerken CAROL-FI aracını kullanmıştır. Bu mimaride çalıştırılan farklı uygulamalara uygulanan hata enjeksiyonları neticesinde, uygulamaların özelliklerine göre farklı sessiz veri bozunumu değerlerine sahip olduğu gözlemlenmiştir. Santos vd. (2019b) tarafından önerilen araç ise benzer olarak cuda-gdb hata ayıklayıcısı üzerinde geliştirilmiş olup CUDA uygulamalarına hata enjeksiyonu yaparak hata toleranslarını tespit etmektedir. Bu aracın uygulanması olarak aynı yazarlar, diğer çalışmalarında (Santos vd. (2019a)), Histogram of Oriented Gradients (HOG) ve You Only Look Once (YOLO)

nesne algılama çerçevelerinin hata hassasiyetlerini incelemiştir.

SASSIFI (Hari vd. (2017)), GPU uygulamaları için mimari seviyede bir hata enjeksiyonu aracı sunmaktadır. NVIDIA GPU mimarileri assembly dili olan SASS seviyesinde ayrıntılı inceleme (profiling) sağlayan SASSI (Stephenson vd. (2015)) üzerinde geliştirilmiş olan SASSIFI, üç temel adımda çalışmaktadır: 1) Hata enjeksiyonu yapılacak uygulama alanlarının belirlenmesi ve ayrıntılı incelenmesi, 2) İstatistiksel olarak hata enjeksiyonu bölgelerinin seçilmesi, 3) Çalışan uygulamalara hataların enjekte edimesi ve hata davranışının incelenmesi. NVIDIA tarafından sağlanan programlama modeli CUDA için ayrıntılı inceleme (profiling) arayüzü olan CUPTI'nin de kullanıldığı bu hata enjeksiyon aracı, GPU uygulamalarının hata hassasiyeti analizlerine imkan sağlamaktadır. Sessiz veri bozulumunun detaylı incelenmesi, program özellikleri ile program hata hassasiyetinin ilişkisi, uygulama seviyesinde hata azaltıcı faktörlerin belirlenmesi gibi analizler açık kaynak kodlu ve geliştirilmesi kolay bir araç olan SASSIFI ile mümkündür. İlgili çalışmada da belirtildiği gibi, SASSIFI, Program Hassasiyet Faktörü (PVF) (Sridharan & Kaeli (2009)) ölçümüne benzer uygulama seviyesinde bir yaklaşım olmakla birlikte hata enjeksiyonu yöntemiyle hata hassasiyetini ölçmektedir. Previlon vd. (2020) tarafından yapılan çalışma, SASSIFI aracını kullanarak çalıştırma parametrelerinin GPU uygulamalarının hata hassasiyetleri üzerindeki etkilerini incelemekte olup aracın kullanım alanına bir örnek olarak verilebilir. Yazarlar gözlemleri sonucunda en önemli hata enjeksiyonu noktalarını belirleyen ve gerekli hata enjeksiyonu testlerinin sayısını azaltan Spoti-Fi yöntemini önermişlerdir. Yakın zamandaki çalışmada, Tsai vd. (2021) SASSIFI'ye benzer yapıda bir hata enjeksiyonu aracı sunmaktadır. Önerilen araç (NVBitFI), güncel GPU mimarileri için kullanılması önerilen ve SASSI'nin yerine geçen NVBit (Villa vd. (2019)) temel alınarak geliştirilmiştir. Santos vd. (2021) gerçek ışın deneyleri ile NVBitFI üzerinde gerçekleştirilen hata enjeksiyonu testlerini karşılaştırmıştır.

LLFI-GPU (Li vd. (2016)), derleyici tabanlı bir hata enjeksiyon aracı sunarak GPU uygulamalarındaki hata yayılımını incelemektedir. NVDIA CUDA programlama modelinin derleyicisi olan nvcc'nin temelini oluşturan LLVM derleyicisi üzerine geliştirilmiş olan LLFI-GPU, çalıştırma süresi ve hafıza durumları olmak üzere hata yayılımını ölçmektedir. Hafıza durumlarını toplam hafıza (Total memory), sonuç hafızası (Result memory) ve çıktı hafızası (Output memory) olmak üzere üç farklı sınıfa bölerek farklı hafıza durumlarındaki sessiz veri bozunumu oranı, hataların sonuç hafızasına yayılım hızı, hataların farklı hafıza durumlarına yayılıp yayılmadığı anlaşılmaya çalışılmaktadır. Ayrıca GPU fonksiyonlarındaki (kernel) hataların yayılıp yayılmadığı, programın çökmesine (crash) sebep olan hataların öncesinde CPU veya diğer fonksiyonlara yayılıp yayılmadığı ve uygulamaların güvenilirlik karakteristiklerinin farklı GPU mimarilerinde farklı olup olmadığı sorularına yanıt aranmaktadır.

Yang vd. (2021c) küçük girdi boyutlarıyla hata enjeksiyonu yaparak verimli bir şekilde hata hassasiyeti belirlemeyi amçlamaktadır. Uzun süren hata enjeksiyonu deneyleri yerine GPU programlarının hata hassasiyetlerini girdi boyutunun bir fonksiyonu olarak tanımlayarak küçük girdilerle gerçekleşen deneylerden programın büyük girdilerle çalışması durumundaki hata hassasiyetini hesaplamaktadır. Diğer bir çalışmada da (Yang vd. (2021b)), hedef programlar iş parçacığı, komut, döngü ve bit seviyesinde incelenerek enjekte edilmesi gereken hataların sayılarının azaltılması hedeflenmiştir.

2.2 Uygulamaların Hata Toleransının Tahminlenmesi

Hata enjeksiyonu, hedef uygulamanın hata hassasiyetini ölçmek için faydalı bir yöntem olsa da, özellikle uzun süren uygulamalar için çok sayıda deney çalıştırmak pratik olmayabilmektedir. Bu nedenle, her bir hedef program için hata enjeksiyon deneyleri yapmadan geçici hata hassasiyetini tahmin etmek cazip bir çözüm haline gelmiştir. CPU uygulamalarının güvenilirliğini veya hata hassasiyetini tahmin etmek için makina öğrenmesi tabanlı birçok çalışma yapılmış olmasına rağmen, GPU mimarisinin karmaşıklığı ve tahminlemesinin zorluğu sebebiyle bu sistemlerde çalışacak uygulamaları hedefleyen tahminleme çalışmaları kısıtlıdır.

PARIS (Guo vd. (2021)), bir dizi küçük kernel fonksiyonu kullanarak çeşitli regresyon modellerini eğitmiş, bir dizi görünmeyen uygulamanın sessiz hata bozunumu (SDC), kesme (interrupt) ve doğru çalışma (success) oranlarını tahmin etmiştir. Modellerinde öznitelik olarak önce çeşitli komut grupları, dayanıklılık kalıpları, dayanıklılık ağırlıkları ve komut sırası bilgileri kullanılmış, daha sonra bir filtreleme yöntemi uygulanarak daha fazla ilgili özellik seçilmiştir. Ayrıca, kullanılan modellerin tahmin doğruluğunu iyileştirmek için hiperparametre ayarlama yöntemlerini gerçekleştirmişlerdir.

Laguna vd. (2016), komut tipi, komutun ait olduğu temel bloğun ve fonksiyonun özellikleri ve komutları etkileyen komutlar gibi komut özelliklerini dikkate alarak komutları hatalı çıktılar üretme olasılıklarına göre sınıflandırmak için makine öğrenmesi yöntemlerini kullanmışlardır. Sessiz hata bozunumu oluşturan komutları tahmin etmek için SVM sınıflandırıcısını kullanmışlardır. Önerilen derleyici çerçevesi, hataların etkisini azaltmak için yalnızca seçilen talimatları çoklamaktadır. Benzer şekilde, Liu vd. (2019) çalışmasında, komuta özgü özellikler kullanılarak belirli bir uygulamada sessiz veri bozunumuna neden olan komutları tahmin etmek için Random Forest temelli bir model eğitilmiştir. Ek olarak, Liu vd. (2018), komuta özgü özelliklere sahip bir LSTM ağ modeline dayalı sessiz veri bozunumu tahmin yöntemi önermişlerdir. Yang & Wang (2019), kısmi deneyler yürüterek daha düşük bir hata enjeksiyonu maliyetiyle talimatların sessiz veri bozunumu oranlarını

tahmin etmek için bir makine öğrenmesi modeli oluşturmuşlardır. Sınıflandırma ve Regresyon Ağacı (Classification and Regression Tree-CART) tabanlı modellerini eğitmek için komut ve hata yayılımı özelliklerini göz önünde bulundurmuşlardır.

Oliveira vd. (2018), HPC uygulamaları için program hata hassasiyeti faktörü (PVF) tahmin mekanizmasını sunmuştur. Önbellek kayıp oranı (cache miss rate), TLB kayıp oranı, dallanma (branch) ve hafıza komutları oranları gibi uygulama özelliklerini kullanarak SVM tabanlı bir model oluşturmuşlardır. Vishnu vd. (2016), makine öğrenmesi modeli için gerekli özellikler oluşturmak için hata enjeksiyonu deneylerinden toplanan anlamsal bilgileri kullanmıştır. Çok bitlik bir hafıza hatası varsa büyük ölçekli uygulamaların bir hatayla sonuçlanıp sonuçlanmayacağını tahmin etmek için uygulama hatası modelleri oluşturulmuştur. Mutlu vd. (2019), yinelemeli yöntemlerin geçici hatası için temel doğruluk tahminini (ground-truth prediction) değerlendirmek amacıyla AdaBoost regresyonu temelli makine öğrenmesi modeli önermiştir.

PRISM (Kalra vd. (2018)), GPU programlarının komut türlerine dayalı olarak hata oranlarını tahmin etmek için istatistiksel modeller sunmaktadır. Önerilen regresyon modelleri, SDC tahmininin iyi performans göstermediği lineer regresyon ve K-en yakın komşu algoritmalarını kullanmaktadır.

GPU hatalarının tahmini için makine öğrenmesi modelleri öneren Nie vd. (2018), geçici hata hassasiyetine odaklanmak yerine, GPU cihazları içeren büyük ölçekli bir sistemde hedef programın yürütülmesi sırasında bir hata oluşup oluşmadığını tahmin etmeyi amaçlamaktadır. Model, uygulamaya özel karakteristikler, sıcaklık/güç tüketimi, düğüm konumu ve hata frekansı gibi hem zamansal hem de uzaysal özellikleri içermektedir.

2.3 Seçimli Hata Toleransı Yöntemleri

Başlıca hata toleransı yöntemi olarak uygulanan çoklama (redundancy), donanım parçalarının çoklanmasına yönelik olarak donanım seviyesinde olabileceği gibi kod parçalarının çoklanması şeklinde yazılım seviyesinde de gerçeklenebilmektedir (Mittal & Vetter (2016)). Yazılım düzeyinde çoklama yaklaşımları doğrudan programlarla çalışmakta, ek komutlar hedef koda eklenmektedir (Dimitrov vd. (2009)). Yedekli kodlar, hedef mimarinin potansiyel olarak paralel yürütme birimlerinde yürütülerek nihai çıktı için hata algılama veya düzeltme sağlanmaktadır. Çoklama yöntemi yüksek hata kapsamına (fault coverage) ulaşırken, bir programın tamamının çoklanmış hali ile çalıştırılması enerji verimsizliği, performans kaybı ve kaynak tüketimi ile sonuçlanmaktadır. Bundan dolayı, hedef program kodunda hataya en açık bölgeleri bulup sadece o parçaları çoklamak, hem yüksek performans hem de kabul edilebilir hata oranı sağlamaktadır.

Wadden vd. (2014a), OpenCL tabanlı GPU uygulamaları için derleyici düzeyinde yedekli bir çoklu iş parçacığı (redundant multithreading) yaklaşımı önermektedir. Yaklaşım, komut tabanlı veya fonksiyon tabanlı çoklamanın neden olduğu ek yükleri ortadan kaldırarak çalışma gruplarını (work-group) çoklamaktadır. Ek olarak, Gupta vd. (2017) yedekli komutlar arasında senkronizasyon yükünü azaltmak için parmak izi ve çapraz şerit işlemlerini kullanan derleyici teknikleri önermekte, hem iş parçacığı tabanlı hem de blok tabanlı çoklama yöntemleri için daha verimli yedekli yürütmeler sunmaktadır.

ArmorAll (Kalra vd. (2020)), GPU'lardaki geçici donanım hatalarıyla başa çıkmak için derleme düzeyinde seçici bir yaklaşım önermektedir. Çalışma; Address Armor, Value Armor ve Hybrid Armor olmak üzere üç farklı derleyici tabanlı yedeklilik yöntemi sunmaktadır. Address Armor yönteminin amacı, hafıza erişim adreslerinin hesaplanmasında kullanılan her komutu çoklayarak hafıza komutları tarafından kullanılan adresleri korumaktır. Öte yandan Value Armor hafızaya yazılacak değerlerin hesaplanmasına katılan her komutu kopyalamaktadır. Hybrid Armor'un amacı da hem adresleri, hem de değerleri korumaktır. Farklı komutları koruyarak seçimli çoklama sağlayan ArmorAll, hem yüksek hata toleransı hem de iyi performansı hedeflemektedir.

Yang vd. (2021a), bir warp içindeki her parçacığın belirli bir zamanda aynı komutu yürütmesine dayanmaktadır. Aynı hata hassasiyeti özelliğine sahip iş parçacıklarını aynı warplarda çalışacak şekilde yeniden düzenleyerek seçici bir çoklama yöntemi sunmaktadır. Hata hassasiyeti düşük iş parçacıkları ve hata hassasiyeti yüksek iş parçacıkları ayrı warplarda çalıştırılarak, seçilen hata hassasiyeti yüksek olan iş parçacıklarının bulunduğu warpların çoklanması sağlanmaktadır. Çalışma, yalnızca hata hassasiyeti yüksek iş parçacıkları içeren warpların çoklanmasının daha verimli olduğunu ortaya koymaktadır.

3 GEREÇ VE YÖNTEM

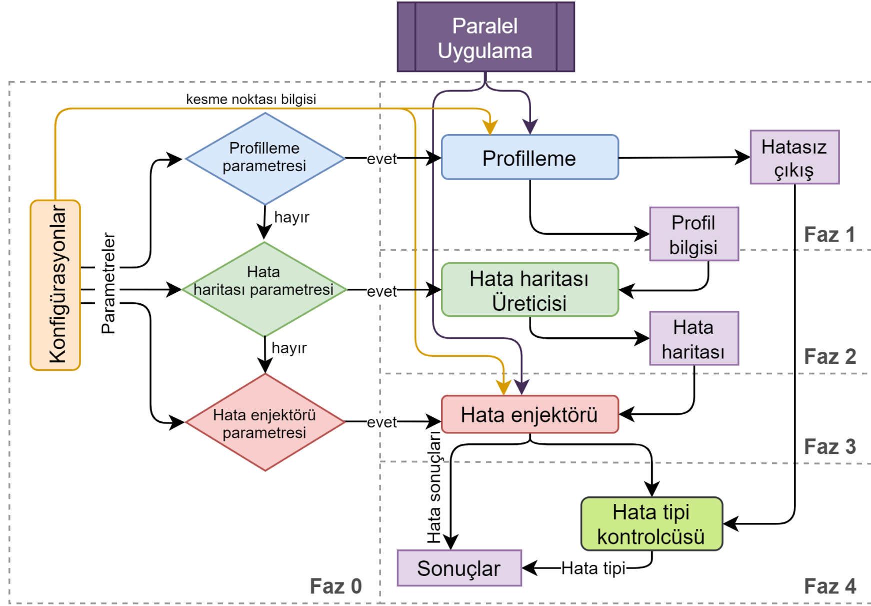
Gereç ve yöntem üç ana başlık altında anlatılmıştır. Projemizin İş Paketi 1 ve İş Paketi 2 kapsamında geliştirilen hata enjeksiyonu aracımızın detayları Bölüm 3.1'de verilmektedir. Sonrasında geliştirilen aracımızın kullanımıyla gerçekleştirdiğimiz, sırasıyla İş Paketi 3 ve İş Paketi 4'e karşılık gelen hata hassasiyeti tahminleme ve seçimli hata toleransı yöntemlerimiz Bölüm 3.2 ve Bölüm 3.3'te açıklanmaktadır.

3.1 GPGPU Uygulamaları İçin Bölgesel Hata Hassasiyet Analizi Aracı Geliştirilmesi

Projenin ilk adımı olarak literatürde var olan hata enjeksiyonu araçları incelenerek geliştirmeyi hedeflediğimiz araç için altyapı oluşturulmuştur. Program kaynak koduyla hata karakteristikleri arasında ilişki gözlemlemeyi amaçladığımızdan, uygulama seviyesine yakın hata enjeksiyonu araçları incelenmiştir. Bu kapsamda cuda-gdb hata ayıklayıcısı temelli ve açık kaynak kodlu GPU-Qin (Fang vd. (2014)) ve CAROL-FI (Santos vd. (2019b)) araçlarının incelenmesiyle başlanmıştır. Nispeten eski bir hata enjeksiyonu aracı olan GPU-Qin aracının güncel GPU mimarilerinde çalışmasının mümkün olmadığı gözlendikten sonra çalışmalarımız CAROL-FI aracı üzerine yoğunlaşmıştır. Birbirini takip eden farklı adımlar içeren ve konfigürasyon arayüzüyle hata enjeksiyon deneylerini mümkün kılan bu araç üzerinde hata enjeksiyon testleri gerçekleştirilmiştir. Öncelikle proje kapsamında geliştirmeyi hedeflediğimiz hata enjeksiyonu aracını CAROL-FI üzerine kurulması planlanmasına rağmen, aracın özellikle küçük boyutlu/kısa süreli uygulamalar için kullanımının mümkün olmadığının görülmesi ve bazı durumlarda çalıştırılmasında problemler gözlemlenmesi üzerine CAROL-FI aracının tasarımı temel alınarak GPGPU uygulamalarının bölgesel hata hassasiyetini belirleyecek kendi özgün hata enjeksiyon aracımızın geliştirilmesine karar verilmiştir.

3.1.1 Bölgesel Hata Hassasiyet Aracı

Projemizde geliştirdiğimiz bölgesel hata enjeksiyonu aracı, rastgele bir zaman diliminde hata enjeksiyonu yerine belli bir kernel fonksiyonunda veya kernel fonksiyonu içindeki belli bir kod satırında hata enjeksiyonu yapma özelliklerine sahiptir. Bu sayede spesifik olarak her satırın veya her bölgenin kendine has güvenilirlikleri ölçülebilmektedir. Araç, Şekil 1'deki gibi işlemektedir ve şu aşamalardan oluşmaktadır:



Şekil 1: Bölgesel hata enjeksiyonu aracı.

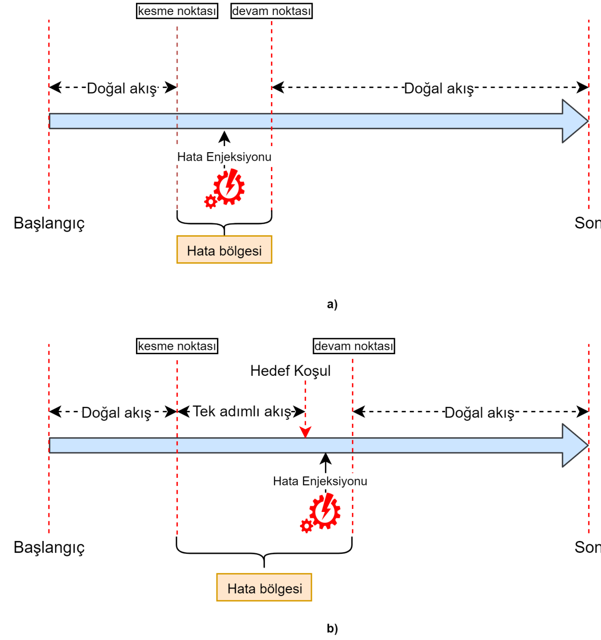
- **Faz 0: Konfigürasyonları belirleme**

  Öncelikle, yapılması planlanan enjeksiyon işlemi için konfigürasyonların doğru belirlenmesinin önemi yüksektir. Araç tasarlanırken, kullanıcının sadece konfigürasyonları belirterek aracı kullanabilmesi hedeflenmiştir. Bu sebeple hata enjeksiyonu esnasında gerekli olacak her bir özellik konfigürasyonlarda bulunmaktadır. Aracın sunduğu başlıca özellikler şunlardır:

  - Uygulama bazında hata enjeksiyonu,
  - Satır bazında veya koşul bazında hata enjeksiyonu,
  - Özel veya bölgesel olarak blok, iş parçacığı, yazmaç veya bit bazında hata enjeksiyonu,
  - 3 fazlı hata enjeksiyonu.

- **Faz 1: Profilleme**

  GPU cihazlarının hafızaları oldukça büyük olduğu için tüm hafızaya yapılan hata enjeksiyonundaki SDC oranları oldukça düşüktür. Bu sebeple profilleme uygulamanın aktif olarak kullandığı blokları ve iş parçacıklarını bularak rastgele olarak yapılan enjeksiyonun daha odaklı olmasını sağlar. Bunu sağlamak için, uygulama hata ayıklayıcısı (cuda-gdb) üzerinden çalıştırılarak konfigürasyonlarda belirtilen kesme noktasındaki (breakpoints) aktif blok ve iş parçacığı sayısı bulunur. Ayrıca, iş parçacığı başına düşen yazmaç sayısı ve bit sayısı kon-



Şekil 2: Hata enjektörünün akış diagramı. a) Belirli bir kesme noktasındaki hata enjeksiyonu. b) Özel bir koşul altında hata enjeksiyonu.

figürasyonlarda manuel olarak belirtilmiştir. Buna paralel olarak SDC karşılaştırılmasında kullanılmak üzere, uygulamanın hatasız çıkışı (golden output) saklanır.

- **Faz 2: Hata haritası üretimi**

  Profilleme işleminden sonra, hata haritası üreticisi ile elde edilen bilgilere uygun bir hata dağılımı çıkarılır. Hataların eşit olarak dağılımını sağlamak için tekdüze (uniform) dağılım kullanılmıştır. Bu aşamayla, enjekte edilecek hataların konumları belirlenmiş olur.

- **Faz 3: Hata enjeksiyonu**

  Hata enjeksiyonu aşamasında ise, konfigürasyonlarda belirtilen satıra hata haritasındaki koordinatlara göre hata enjekte edilir. Uygulama hata ayıklayıcısı üzerinden çalıştırılıp, kesme noktasında durdurulur. Eğer özel bir koşul belirlenmemişse Şekil 2(a)'daki gibi ilerler. Eğer belirlenmişse Şekil 2(b)'deki gibi o koşula ulaşıncaya dek uygulama adım adım devam ettirilir. Sonrasında eğer belirlenmişse blok ve iş parçacığına odaklanılır, belirlenmemişse hata haritasında rastgele üretilene odaklanılır. Hedeflenen yazmacın depoladığı bilgiye erişilir ve hedef bite hata enjeksiyonu yapılır. Hata enjeksiyonundan sonra uygulama doğal akışında devam ettirilir ve enjekte edilmiş uygulamanın çıkışı toplanır.

- **Faz 4: Sonuçların toparlanması**

  Araç son aşamada, kullanıcıya dönüt olarak hata enjeksiyon sonuçlarını ve eğer SDC gözlemlenmişse verideki bozunumun boyutunun hesaplanması için SDC görülen çıkışı verir. Hata tipi kontrol edilirken ilk önce, askıda kalma (Hang) durumu için uygulamanın süresi hatasız çıkışın süresi ile karşılaştırılır. Eğer uygulama konfigürasyonlarda belirtilen süre çarpanını aşarsa uygulama öldürülür ve hata tipi askıda olarak belirlenir. Eğer uygulama askıda değilse, herhangi bir çökme veya hata kodu varlığına bakılır. SDC için ise enjekte edilmiş çıkışlarla hatasız çıkışlar karşılaştırılır ve bozulma durumuna göre SDC olup olmadığına karar verilir. Eğer hiçbiri gözlemlenmemişse, doğru çalışma olarak karar verilir.

Bunlara ek olarak profilleme, hata haritası üreticisi ve hata enjeksiyonu fazları birbirinden ayrı olarak çalışabilir durumdadır. Bu sayede dinamik profilleme ve dinamik hata haritası oluşturulması yerine önceden tanımlanmış bilgilerle hata enjeksiyonu başlatılabilir.

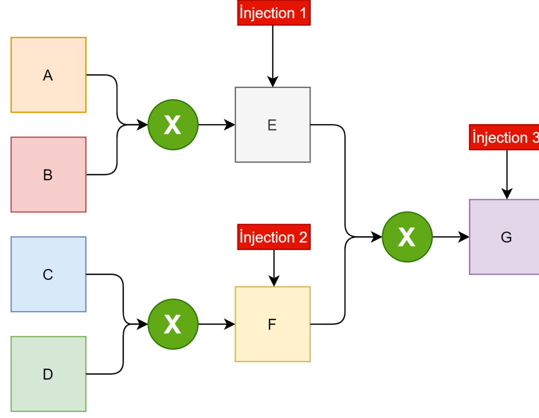
3.1.2 Hata Yayılımı Takibi

Hata enjeksiyonu aracımız, hedef uygulamadaki veri yapılarının hesaplanan değerlerini elde etmek ve GPU hafıza alanlarındaki hata yayılımının değerlendirilmesi amacıyla konfigüre edilebilmektedir. Bu özelliğinin aktive edilmesiyle, kullanıcı tarafından belirlenen kod parçalarında belirtilen değişkenlerin değerleri kaydedilmektedir. Böylece değişkenlerin olması gereken değerleri ve hangi kod bölgesinde, ne miktarda değişikliğe uğradığı (ne kadar bozulduğu) gözlemlenmektedir. Bu özelliği hata ayıklayıcısı (debugger) seviyesinde, istenilen program satırlarında uygulamayı durdurarak (breakpoint ile) ve istenilen değişkenlerin değerlerini kaydederek gerçekledik. Bu şekilde uygulamanın kaynak koduna müdahale etme gerekliliği bulunmamaktadır. Hata yayılımının gözlemlenmesi ile, bir hatanın uygulamanın güvenilirliğine program satırı ve değişkenler seviyesindeki etkisini değerlendirmek mümkün olmaktadır.

Hata yayılımı analizimizi örnek bir uygulamayla şu şekilde açıklayabiliriz: Aşağıda kernel fonksiyonları verilen, Polybench uygulama setinden (Grauer-Gray vd. (2012)) bir lineer cebir uygulaması olan 3MM'de, üç farklı kernel fonksiyonu içinde üç farklı matris çarpma işlemi yapılmaktadır. Her bir iş parçacığı girdi matrislerinin bir satır ve sütunu için bir çarpma ve toplama işleminden sorumludur.

```
__global__ void mm3_kernel1(float *A, float *B, float *E){
1:  int j = blockIdx.x * blockDim.x + threadIdx.x;
2:  int i = blockIdx.y * blockDim.y + threadIdx.y;
3:  if ((i < size) && (j < size)){
4:      int k;
5:      for(k=0; k < size; k++){
6:          E[i*size+j] += A[i*size+k] * B[k*size+j];
7:      }
8:  }
}
__global__ void mm3_kernel2(float *C, float *D, float *F){
1:  int j = blockIdx.x * blockDim.x + threadIdx.x;
2:  int i = blockIdx.y * blockDim.y + threadIdx.y;
3:  if ((i < size) && (j < size)){
4:      int k;
5:      for(k=0; k < size; k++){
6:          F[i*size+j] += C[i*size+k] * D[k*size+j];
7:      }
8:  }
}
__global__ void mm3_kernel3(float *E, float *F, float *G){
1:  int j = blockIdx.x * blockDim.x + threadIdx.x;
2:  int i = blockIdx.y * blockDim.y + threadIdx.y;
3:  if ((i < size) && (j < size)){
4:      int k;
5:      for(k=0; k < size; k++){
6:          G[i*size+j] += E[i*size+k] * F[k*size+j];
7:      }
8:  }
}
```

Bu programın matris çarpımlarını hangi sırada yaptığı Şekil 3'te verilmektedir. İlk iki kernel fonksiyonunda hesaplanan değerler, son kernel fonksiyonunda kullanıldığından uygulamanın sonlarına doğru oluşan hataların sonuç matrisinde daha az sayıda elemanın hatalı hesaplanmasına sebep olacağı düşünülmektedir. E veya F matrislerinin hesaplandığı sırada (Injection 1 veya Injection 2) oluşan bir hata, sonuç matrisinin 1 veya 2 satırını tamamen bozacaktır. Bununla birlikte G matrisinin hesaplandığı sırada (Injection 3), yani uygulamanın sonlarına doğru oluşan bir hata, sonuç matrisinin sadece 1 veya 2 elemanının bozulmasına sebep olacaktır. Hata yayılımı analizimizde, hataların değişkenler üzerindeki etkilerini anlayabilmek amacıyla görselleştirmeler uyguladık. Şekil 4'te bir örneği verildiği gibi sessiz hata bozunumu (Silent Data Corruption-SDC) du-

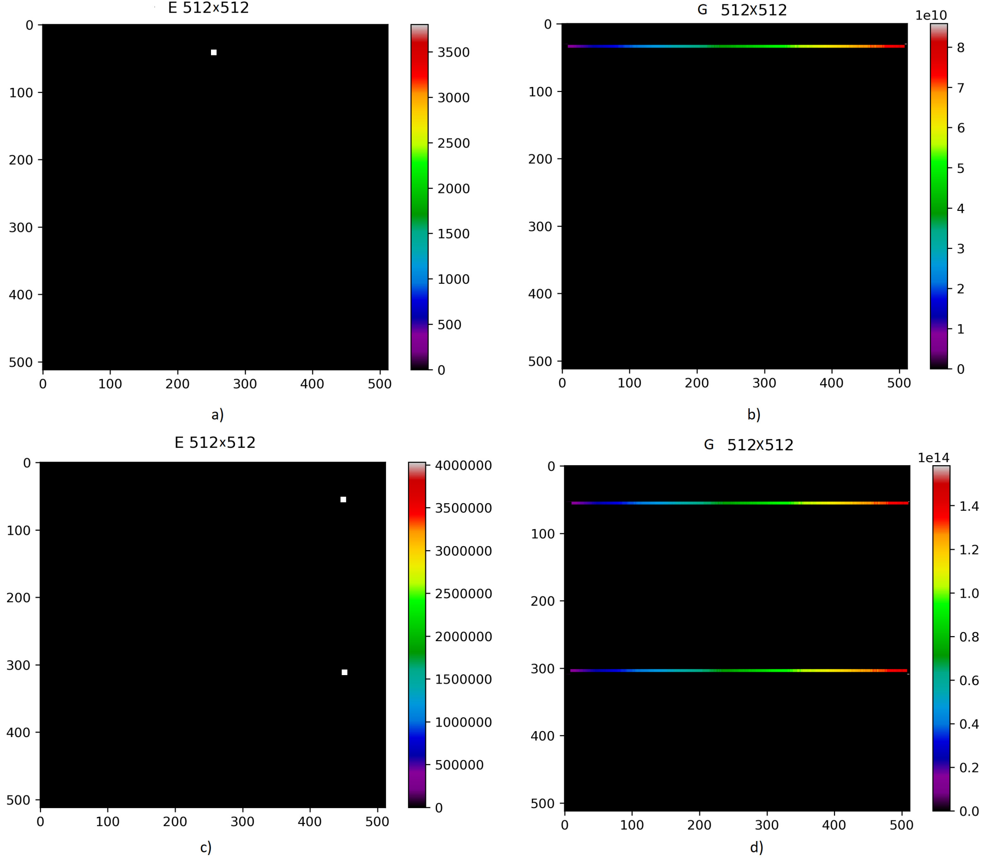


Şekil 3: 3mm uygulamasının akış diyagramı.

rumlarında, uygulamadaki hangi değişkenlerin, hangi elemanlarının olması gerekenden farklı şekilde hesaplandığını değişkenlerin elemanlarını farklı renklerde ifade ederek gösterdik. Siyah renk, belirtilen matris elemanının doğru hesaplandığını (0 hata ile) belirtirken; beyaz renk, elemanın değerinin diğer değişikliklere oranla maksimum farkla yanlış hesaplandığını göstermektedir. Şekil 4, yukarıda verilen 3mm uygulamasının *mm3_kernel1* kernel fonksiyonu çalıştırılırken gerçekleştirilen hata enjeksiyonu neticesinde sonuç matrisinin (G matrisi) yanlış hesaplanmasıyla sonuçlanan bir durum için *mm3_kernel1* fonksiyonunun çıktı matrisi olan E matrisi ve G matrisinin elemanlarının bozulma miktarlarını göstermektedir. Şekil 4.a, bir E elemanının bozulmasını göstermekle birlikte Şekil 4.b bunun G matrisindeki ilgili satırı tamamen bozduğunu göstermekte olup, Şekil 4.c diğer bir hata enjeksiyonu neticesinde iki elemanı bozulan E matrisindeki farklı elemanları ve bunun neticesinde Şekil 4.d, G matrisinin bozulan iki satırını göstermektedir. Bu şekillerden E matrisinin yanlış hesaplanmasına sebep olan hataların, çıktı matrisi olan G matrisine ne şekilde yayılım gösterdiği gözlemlenmektedir. Geliştirdiğimiz araç, hem görsel, hem de nümerik olarak bu yayılımı raporlayabilmektedir.

3.2 GPGPU Uygulamalarının Hata Hassasiyetlerinin Tahminlenmesi

Hata enjeksiyonu, hedef uygulamanın geçici hatalar karşısındaki güvenilirliğini nicelendirmek açısından kullanışlıdır. Fakat deney sayısı çok fazla olacağından dolayı pratik değildir. Bundan dolayı, deneyleri fazlaca tekrar etmek yerine meydana gelebilecek geçici hata oranlarını tahminlemek daha ilgi çekici bir çözümdür. Projemiz kapsamında, genel amaçlı GPU (GPGPU) uygulamaları için makine öğrenmesi temelli hata tahminleme çalışması yapılmıştır. Amacımız, uzun süren hata en-



Şekil 4: Program değişkenlerindeki bozulmaların görselleştirilmesi.

jeksiyon zamanını azaltmak ve uygulamaların hata toleranslarını simülatör ya da profil oluşturucu tarafından sağlanan metriklerin makine öğrenmesi temelli tahminlemesi sonucunda belirlemektir.

Çalışmadaki ana odağımız hata oranlarını donanım metrikleri ve performans metrikleri ile ilişkilendirerek tahmin etmektir. Hataların oluşma sıklığını tahmin edebilmek için regresyon ve sınıflandırma temelli makine öğrenme metotlarından faydalandık. Şekil 5 çalışmamızdaki deney basamaklarının genel yapısını göstermekte olup akışımız aşağıda açıkladığımız dört ana basamaktan oluşur:

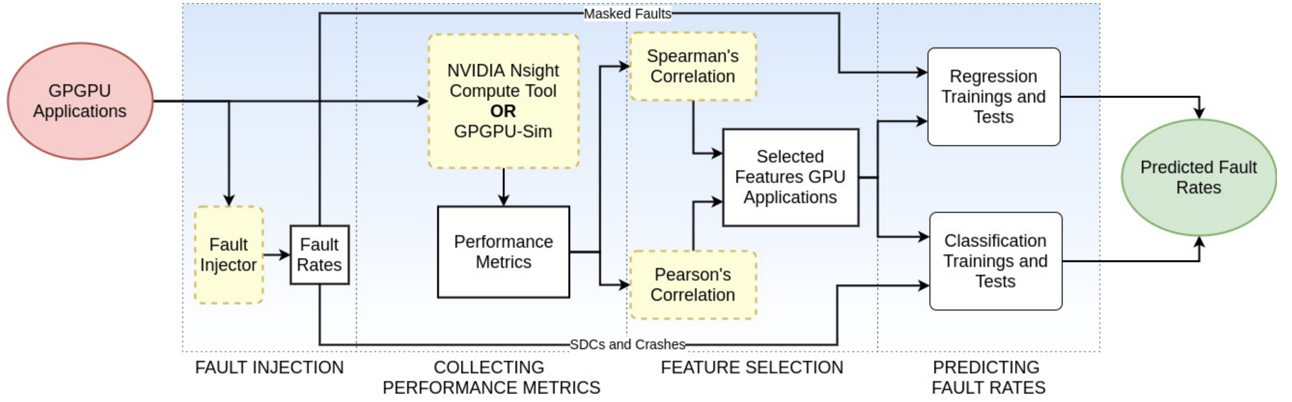
1. **Hata enjeksiyonu:** Hedeflenen GPGPU programlarında meydana gelen hata oranlarını ölçmek için hata enjeksiyon deneyleri çalıştırılarak GPGPU uygulamalarının kernel fonksiyonu özelinde hata hassasiyetleri sessiz hata bozunumu (SDC), çökme (Crash) ve doğru çalışma

(Masked) olarak toplanır.

2. **Performans metriklerinin toplanması:** Hedeflediğimiz GPU uygulamalarına ait metrikler, hem GPGPU-Sim simülatörü (Bakhoda vd. (2009); Khairy vd. (2020)), hem de NVIDIA Nsight Compute tool (Nsi (2021)) profil oluşturucu ile toplanır.

3. **Belirleyici özelliklerin seçimi:** Simülatör ve profil oluşturma aşamalarında çok fazla metrik sunulması ve bu metriklerin de tahminleme performansını olumsuz etkilemesinden dolayı, Spearman ve Pearson korelasyon metotlarından faydalanarak hatalarla en ilişkili metrikler seçilir.

4. **Hata oranı tahminlenmesi:** Hata hassasiyeti oranları makine öğrenmesi temelli regresyon ve sınıflandırma metotları ile tahmin edilir.



Şekil 5: Hata tahminleme yapımızın genel işleyişi.

Hata enjeksiyonu, İş Paketi 2 kapsamında Bölüm 3.1'de anlatıldığından bu bölümde diğer adımlarla ilgili kısımlar anlatılacaktır.

3.2.1 Performans Metriklerinin Toplanması

Tahminleme çalışmamız için GPGPU programlarının karakterlerini belirlemek adına, donanım kullanımı ve performans ile ilgili birçok metrikleri toplanmıştır. Bu metrikler yaygın olarak kullanılan simülatör ve profil oluşturucu yardımı sayesinde gözlemlenmiştir. İki yaklaşım sonucunda toplanan metrikler de değerlendirilmiş olup bunlara ait sonuçlar karşılaştırılmıştır. İlk yaklaşım olarak, çalışmamızda kullanacağımız programlar GPGPU-Sim (Bakhoda vd. (2009); Khairy vd.

Tablo 1: Simülatörden toplanan metrikler.

Performans Metriği	**Açıklaması**
Load komutu	Toplam load komutu sayısı
Store komutu	Toplam store komutu sayısı
Parametre hafıza komutu	Toplam parameter hafızası komutu sayısı
Toplam komut	Kernel fonksiyonu sonucundaki toplam komut
Komut/Döngü	Bir döngüde (cycle) gerçekleşen komut sayısı
Global (DRAM) hafıza okuma	Toplam global hafıza okuması
Global hafıza yazma	Toplam global hafızaya yazma
Warp doluluğu	SM'lerdeki ortalama warp (32 thread) doluluk miktarı
Kontrol akış komutu yoğunluğu	PTX kodunda toplam kontrol akış komutu / toplam komut
Ondalıklı işlem komutu yoğunluğu	PTX kodunda toplam ondalıklı işlem komutu / toplam komut
Tam sayı işlem komutu yoğunluğu	PTX kodunda toplam tam sayı işlem komutu / toplam komut
Mantık komutu yoğunluğu	PTX kodunda toplam mantık komutu / toplam komut
Veri taşıma komutu yoğunluğu	PTX kodunda toplam veri taşıma komutu / toplam komut
Load komutu yoğunluğu	PTX kodunda toplam load komutu / toplam komut
Koşullu komut yoğunluğu	PTX kodunda toplam koşul komutu / toplam komut

Tablo 2: Profil oluşturucudan toplanan metrikler.

Performans Metriği	**Açıklaması**
SOL SM	SM doluluğu
SOL Hafıza	Hesaplama memory borusu doluluğu
SOL L1 Tex Cache	L1 texture hafıza doluluğu
SOL L2 Cache	L2 cache doluluğu
SOL DRAM	GPU DRAM doluluğu
Süre	Toplam süre (milisaniye)
Geçen toplam döngü (cycle)	GPU'nun aktif olduğu toplam cycle
Komut/Döngü	Bir döngüde (cycle) gerçekleşen komut sayısı
SM meşguliyeti	Yüzde olarak SM'lerin aktif ve dolu olduğu oran
Hafıza verimliliği	DRAM'den erişilen byte/saniye (Gbyte/saniye)
L1 texture vuruş (hit) oranı	L1 memory bölümlerindeki başarılı erişim sayısı
L2 vuruş oranı	L2 bölümlerindeki başarılı erişim sayısı
Hafıza meşguliyeti	Cache'ler ve DRAM içindeki içsel aktivitelerin verimliliği
Maksimum bant doluluğu	SM, cache'ler ve DRAM aralarındaki bant doluluğu
Bir scheduler için aktif warp	Bir scheduler (dağıtıcı/düzenleyici) için aktif warp sayısı
Warp cycle ve komut	Warp'in bir instruction'u çalıştırmak için gereken ortalama cycle
Çalıştırılan komut	Çalıştırılan toplam warp komutu
Register/thread	Bir thread başına düşen register sayısı
Erişilen doluluk	Yüzde olarak erişilen ortalama doluluk
Erişilen warp	Bir SM'de erişilen aktif warp oranı

(2020)) simülatöründe simüle edilmiştir. GPGPU-Sim, GPU mimarisinde bulunan donanımsal özellikleri konfigüre ederek (SM miktarı, bant genişliği gibi) GPU uygulamalarını simüle etmek için kullanılabilmektedir. Performans simülasyon modu sayesinde, benchmark GPU uygulamalarımızı simüle etmemizin yanı sıra birçok performans metriği toplamamızı da sağlamaktadır. Diğer yöntem olarak, programlar NVIDIA tarafından sağlanan Nsight Compute tool (Nsi (2021)) aracılığı ile çalıştırılmıştır. Nsight Compute tool, GPU uygulamaları için farklı modlarda profil oluşturmamızı sağlamaktadır. Bu modlardan bazıları SASS, PTX seviyesi, detaylı donanım kullanım ve performans metriklerinin gözlenebileceği modlardır. Simülatörden farklı olarak, profil oluşturucuda uygulamayı çalıştırabilmek için gerçek bir NVIDIA donanımı gereklidir. Veri setinin kompleksleşmesinden kaçınmak ve hata tahminlemesinde kullanılacak verinin niteliğini arttırmak için, sadece benchmark uygulamaları ve GPU donanımı arasındaki ilişkiyi temsil edebilecek metrikler toplanmıştır. Örneğin, toplam yükleme/depolama komutu ya da SM verimliliği gibi daha genel ve belirleyici metrikler alınırken, DRAM satır verimliliği metriği gibi detaylı ve özel metrikler elimine edilmiştir.Tablo 1 ve Tablo 2 sırasıyla simülatör ve profil oluşturucudan toplanan metrikleri göstermektedir. Simülatör ve profil oluşturucu farklı amaçlara hitaben oluşturuldukları için farklı şekillerde sonuçlar üretmektedirler ve bundan dolayı iki araç üzerinden aynı metrikleri toplamak beklenmemelidir. Simülatör çoğunlukla performans metrikleri üretirken, profil oluşturucu genel olarak donanım kullanımı ve performans metriklerini istatiksel olarak oluşturur. Yine de Komut/döngü, erişilen SM doluluğu ve toplam sayıda komut gibi metrikler ortak olarak gözlenebilmektedir. Bu araçlar üzerinde gerçekleşen çalışmalar ve gözlenen çıktılar farklı olduğundan dolayı, metrikler ayrı ayrı toplanmıştır ve iki araç için de ayrı tahminleme modelleri oluşturulmuştur.

3.2.2 Metriklerin Seçilmesi

Önemli metrikleri fark etmek ve çalışmamızdaki potansiyel tahminleme başarısını arttırabilmek için, hata oranları ve topladığımız metrikler arasındaki korelasyon değerlerini inceledik. Korelasyon değerlerini incelerken Spearman ve Pearson metotlarını kullandık. Sperman korelasyon metodu seçilecek özellikler ve hata oranları arasındaki monoton ilişkiyi lineer ya da lineer olmaksızın ortaya çıkarır. Sonuçlar [-1,1] arasında normalize edilirken, -1 ve 1'e yakınlık incelenen iki özellik arasında yüksek korelasyon olduğunu gösterir. Spearman korelasyon katsayısı aşağıdaki formülle hesaplanır:

$$\rho_s = 1 - \frac{\sum_{i=1}^{n} d_i^2}{n^3 - n}$$

d iki sıralama değeri arasındaki fark için kullanılırken, n ise gözlem sayısı için kullanılmıştır.

Pearson korelasyonu ise iki girdi arasındaki lineer ilişkiyi açıklamada kullanılır ve korelasyon sonuçları [-1,1] arasına normalize edilir. -1 veya 1'e yakınlık incelenen iki metriğin korele olduğunu gösterirken, 0'a yakınlık ise o metriklerin birbirleri ile uyumsuz olduğunu gösterir. Pearson korelasyon katsayısı ise aşağıdaki formül ile hesaplanır:

$$\rho_p = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y}$$

E beklenti (expectation), σ standart sapma, μ ortalamayı gösterir.

3.2.3 Hata Tahminleme Modellerinin Oluşturulması

Her GPU kernel fonksiyonu için farklı hata oranı ve farklı metrikler belirlediğimiz için, her bir kernel fonksiyonunu tahminleme modelimiz için 1 sample (veri) olarak kullanılmıştır. Çalışmadaki veriyi GPU programından kernel seviyesine düşürmek, oluşan hata tiplerinin sebeplerini anlamak ve mümkün senaryoların teşhisi açısından daha detaylı bir çalışma ortamı yaratmıştır. Farklı hata senaryoları için regresyon ve sınıflandırma yaklaşımları kullanılmıştır. Regresyon yaklaşımında, Random Forest (RF), Support Vector Machine (SVM) ve Gradient Boosting (GB) algoritmaları kullanılmıştır. RF algoritması, birçok karar şeması yaratarak bu şemalardan en kesin olanı deneme setinin uygulanması sonucunda oluşturur ve seçer. SVM lineer ya da parabolik bir eğri oluşturup, bu eğriyi girilen inputlar sonucundaki outputları sağlayacak şekilde düzenlemeye çalışır. Bu düzenlemede deneme ve gerçek sonuç arasındaki hata sonucunu azaltmaya yönelik ilerlenir. GB aşama aşama ilerleyen kümülatif bir eğri yaratır. Kümülatif düzenleme rastgele türevlenebilir kayıp fonksiyonunun kullanımı ile elde edilir. Her aşamada, regresyon eğrisi kayıbı azaltacak şekilde güncellenir ve güncel hali üzerine testler uygulanır. Toplamda üç farklı regresyon algoritması kullanılmıştır ve her bir algoritma dört farklı şekilde hiper-parametrelerle konfigüre edilerek kullanılmıştır. Konfigüre edilmiş her model için tüm metriklerin kullanıldığı ve seçilmiş metriklerin kullanıldığı ikişer farklı deney yapılmıştır. Yani toplamda bir hata tipi için 24 farklı regresyon metodu uygulanmıştır. Kesinlik (accuracy) sonuçları da aşağıdaki formüle göre hesaplanmıştır:

$$(1 - \frac{|hata_{tahminlenen} - hata_{\text{gerçek gözlem}}|}{hata_{\text{gerçek gözlem}}}) * 100$$

Maskelenmiş hatalar için güvenilirlik tahminleme sonuçlarına ulaşabilmiş olsak da benzer sonuçları regresyon modeli kullanarak SDC ve crash hataları için gözlemleyemedik. SDC hataları [0.012,

0.263] arasındayken crash hataları [0.008, 0.173] arasındadır. Maskelenmiş hatalar SDC ve crash hatalarına göre çok daha büyük değerlerde olduğundan dolayı, tahminleme sonucundaki küçük farklılıklar büyük sapmalara sebep olmazken bu durum SDC ve crash hataları için geçerli değildir. SDC ve crash hata tahminlemesindeki küçük sapmalar kesinlik değerinde büyük etkiye sahiptir. Bundan dolayı, regresyon yaklaşımından farklı olarak, crash ve SDC hatalarının tahminlemesi için sınıflandırma temelli bir yaklaşım kullandık. Sınıflandırma yaklaşımında, SDC ve crash hatalarını düşünerek farklı sınıflar tanımladık ve hataların hangi sınıfa dahil olduklarını tahminlemeye çalıştık. Yani iki sınıflı ve üç sınıflı sınıflandırma modellerinde iki veya üç farklı sınıf tanımlanmış olup, uygulamaların hangi sınıfa dahil oldukları tahminlenmeye çalışılmıştır. Örneğin, SDC oranı [0.01, 0.05] arasında olan değerler SDC hatalarına karşı güvenli olarak sınıflandırılırken, [0.051, 0.02] arasındaki değerler SDC hatası ile sonuçlanabilir sınıf olarak değerlendirilir. Buna göre, GPU kernel fonksiyonlarının geçici hatalara karşı güvenilirliklerini tahminleyebiliriz. Uygulamadaki hassasiyete göre sınıf sayısı arttırılabilir ve tahminleme modelleri güncellenebilir. Biz çalışmamızda SDC ve crash hatalarını ikili ve üçlü sınıflara ayrılarak tahminlenmeye çalıştık. RF ve GB algoritmalarının yanı sıra, iki modelin hibrit olarak çalıştırılması ile elde edilen bir sınıflandırma modelini de çalışmamıza ekledik. Bu model Standart Scaler (SS) ve Stochastic Gradient Descent (SGD) algoritmalarını sırası ile çalıştırarak çalışır. Bu güncellemedeki amacımız, skala edilmiş (normalize edilmiş) özelliklerin sınıflandırma tahminlemesindeki etkisini gözlemektir. SS girilen veri kümesini, ortalama değeri 0 ve standart sapma değeri 1 olacak şekilde dönüştürür. SGD algoritması her bir veri için aşamalı kayıpları tutarak, algoritmadaki iç parametreleri öğrenme oranınca günceller. Ek olarak, L1 ve L2 regülarizasyonlarını SGD algoritmasında kullanabiliriz. Bu regülarizasyonlar sayesinde over-fitting durumu, yani sistemin verilen girdiler için tahminleyeceği sonucu ezberlemesi durumunu bozabiliriz. Sınıflandırma modelimizi değerlendirmek için, kesinlik metriğini kullandık. Kesinlik metriği basitçe doğru sınıflandırmaları sayar. Buna ek olarak da precision, recall ve F-score metriklerini de kullandık. Precision tahminlenen değerlerin gerçekten kaç adedinin doğru tahmin olduğunu gösterirken, Recall ise tahminlenmesi gereken işlemlerden ne kadarının doğru tahminlendiğini gösteren metriktir. Bundan dolayı, precision ve recall değerleri kesinlikle ilgilidir. F-score ise precision ve recall metrikleri arasındaki harmonik ortalamayı gösterir. Bu terimlere ait formüller aşağıdaki gibidir:

$$precision = \frac{tp}{tp + fp} \qquad recall = \frac{tp}{tp + fn}$$

$$Fscore = 2 * \frac{precision * recall}{precision + recall}$$

Burada *tp* doğru pozitif, *fp* yanlış pozitif, *fn* ise false negatif için kullanılmıştır.

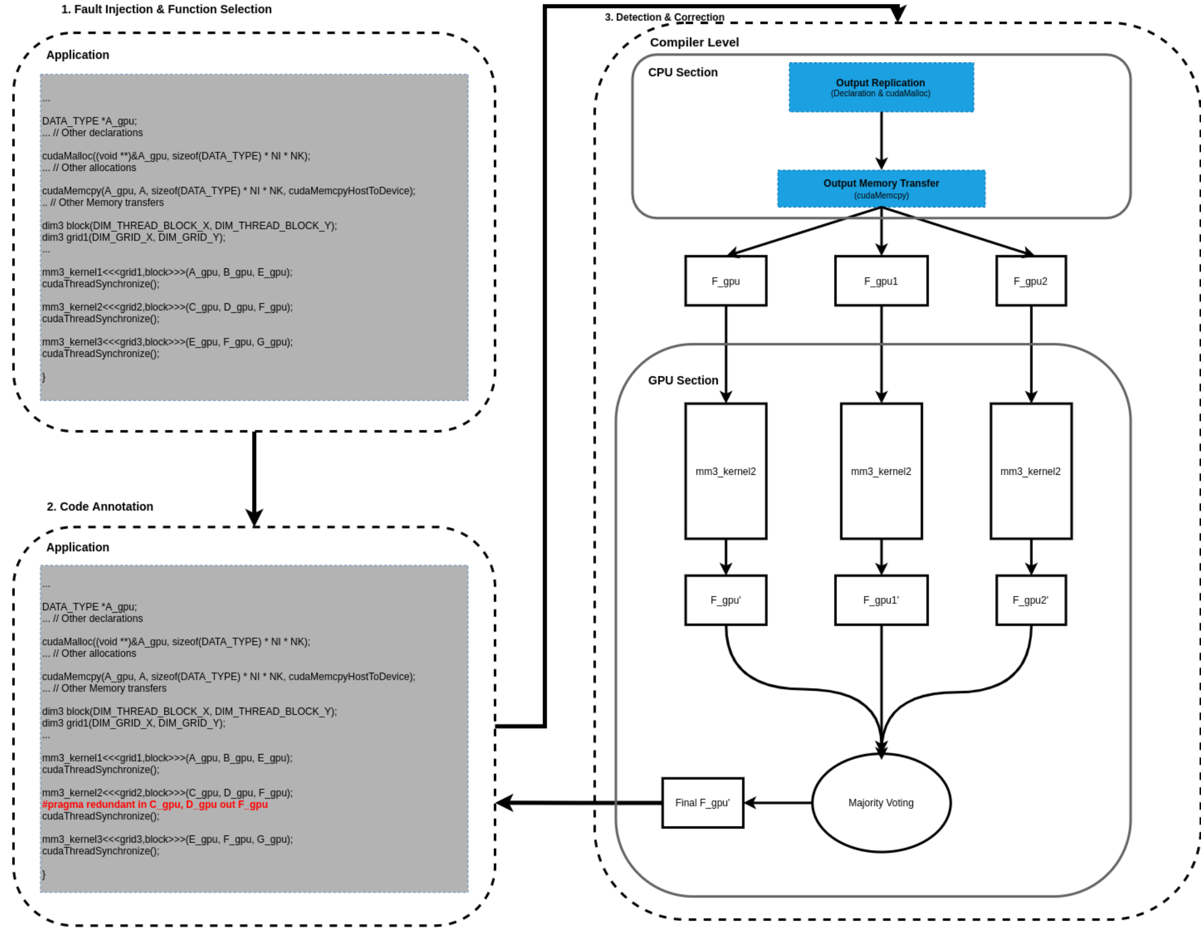
3.3 Seçimli Hata Toleransı Yöntemi Geliştirilmesi

Donanım hatalarının hedef programlardaki etkilerinin önüne geçmek için, kod birden fazla çalıştırılarak yazılım temelli hata toleransı teknikleri uygulanabilmektedir. Çoklama yöntemi yüksek hata kapsamına (fault coverage) ulaşırken, bir programın tamamının çoklanmış hali ile çalıştırılması enerji verimsizliği, performans kaybı ve kaynak tüketimi ile sonuçlanır. Bundan dolayı, hedef program kodunda hataya en açık bölgeleri bulup sadece o parçaları çoklamak, hem yüksek performans hem de kabul edilebilir hata oranı sağlar. Projemizin İş Paketi 3 kapsamında, bölgesel hata enjeksiyonu aracımızla hata toleransı seviyelerini belirlediğimiz GPGPU uygulamalarının tümü yerine hataya en hassas kısımlarının çoklanması mantığına dayalı seçimli hata toleransı yöntemi geliştirilmiştir.

Seçimli hata toleransı yöntemimiz şu şekilde işlemektedir: İş Paketi 2 kapsamında geliştirdiğimiz hata enjeksiyonu aracımızın profil oluşturma aşamasında toplanan bilgilere dayanarak her kernel fonksiyonu için hata yerleştirme noktaları oluşturur. Hedef GPGPU programındaki her bir kernel fonksiyonunun hata güvenlik açığını elde etmek için sessiz veri bozunumu oranlarını değerlendirir. LLVM (Lattner & Adve (2004)) tabanlı derleyici iskeletimiz, hata enjeksiyon analizimizden gelen geri bildirimi kullanan programcı tarafından işaretlenerek belirtilen kernel fonksiyonları için çoklu kod bölümleri de dahil olmak üzere hedef çalıştırılabilir dosyayı oluşturur. Seçici çoklamalı çoklu iş parçacığı yapımızın, Şekil 6'da görüldüğü gibi, üç temel bölümü vardır:

- **Hata enjeksiyonu ve kernel fonksiyonu seçimi:** Hedef GPGPU programlarında kernel fonksiyonlarının hata hassasiyetlerini elde etmek için hata enjeksiyon deneyleri yapıyoruz. Bu şekilde, belirli bir programdaki farklı kernel fonksiyonlarının hata oranlarını inceliyor ve en hataya açık olanı çoklama için aday fonksiyon olarak kararlaştırıyoruz.
- **Kod işaretleme:** Çoklamalı olarak yürütülecek kernel fonksiyonu seçtikten sonra, derleyiciyi çoklamalı yürütme hakkında bilgilendiren ve derleyiciye ek bilgi sağlayan pragma yönergesini ekleriz. Özelleştirdiğimiz derleyicimiz bu yönergeyi destekler ve verilen fonksiyon için gerekli parametrelerle çoklamalı yürütmeyi sağlar.
- **Çoklanmış kod üretimi ve yürütme:** pragma yönergesinden hedef kernel fonksiyonunu aldıktan sonra, derleyici iskeletimiz hem kernel komutlarını hem de ek kopyalar tarafından üretilecek çıktı verilerini çoğaltarak çoklamalı yürütme için kod üretir. Ayrıca, herhangi bir hata durumunda düzeltilmiş sonucu elde etmek için çoğunluk oylaması fonksiyonunu ekler.

İlk bölüm, İş Paketi 2 kapsamında Bölüm 3.1'de anlatıldığından bu bölümde iskelet yapımızı üzerine kurduğumuz LLVM yapısı ve geliştirdiğimiz derleyici desteğiyle ilgili kısımlar anlatılacaktır.

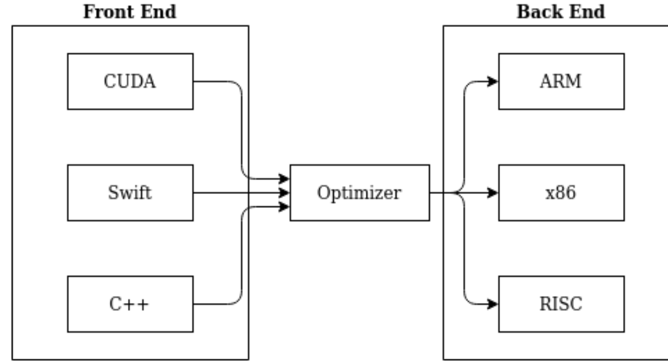


Şekil 6: Çoklamalı çoklu iş parçacığı yapımız.

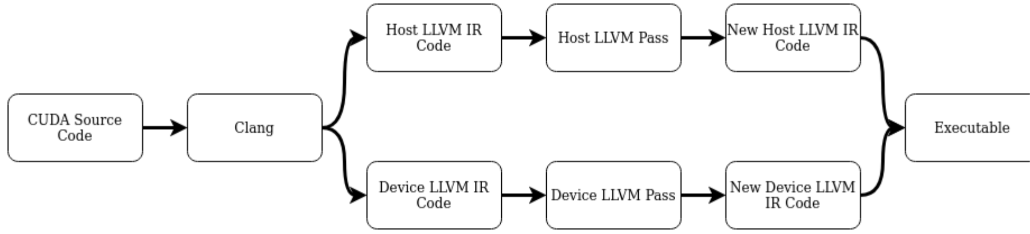
3.3.1 LLVM Derleyici İskeleti

Derleyici-tabanlı çoklamalı yürütme iskelemiz LLVM derleyici iskeleti üzerine inşa edilmiştir. Şekil 7'de görüldüğü gibi, LLVM üç büyük bileşenden oluşur: Ön-uç, Optimize edici ve Arka-uç.

Ön uç bileşeni programlama diline bağlıdır, kaynak kodunu girdi olarak alır ve LLVM IR kodunu oluşturur. Arka uç bileşeni mimariye bağlıdır, LLVM IR kodunu girdi olarak alır ve makine kodunu oluşturur. Optimize Edici bileşeni, LLVM IR kodunu alır ve optimize edilmiş LLVM IR kodunu oluşturur. LLVM iskeleti tabanlı ve Şekil 8'de gösterilen genel derleme akışımız üç bölümden oluşur: 1) LLVM IR kodunun oluşturulması, 2) Hem ana bilgisayar hem de cihaz için yeni LLVM IR kodunun oluşturulması, 3) Oluşturulan LLVM IR kodlarını kullanarak yürütülebilir dosyayı oluşturma.



Şekil 7: Temel LLVM bileşenleri Lattner (2011).



Şekil 8: CUDA kodunun Clang ile derlenmesi.

Derleyici düzeyindeki RMT şemamız, yönergemiz tarafından belirtilen kernel fonsiyonunu ve çıktı verilerini çoğaltır, çoğunluk oylama fonksiyonunu uygular ve programcı tarafından işaretlenen kodun fazladan yürütülmesini sağlar. Uygulamamızda dört ana bileşen vardır: 1) Derleyici yönergesi, 2) Çıktı çoklanması, 3) Kernel fonksiyonunun çoklanması, 4) Çoğunluk oylaması uygulaması. Şekil 8'de verilen derleme aşamasında hem ana makine hem de cihaz kodunda değişiklikler yapıyoruz. Çıktı Çoklama ve Kernel fonksiyonu çağrısı çoklama bileşenlerinin uygulanması ana makine kodunda değişiklikler gerektirse de, Çoğunluk Oylaması için hem ana makine kodunu hem de cihaz kodunu güncelleriz. Kod 1 ve Kod 2, sırasıyla bizim işaretlememiz ve derleyicimiz tarafından oluşturulacak hedef kodun olduğu bir örnek sunar. Sonraki bölümde uygulama detaylarımızı örnek kod parçasında sunarak açıklayacağız.

```
1   void main(){
2   ...
3   DATA_TYPE *A_gpu;
4   ... // Other declarations
5
6   cudaMalloc((void **)&A_gpu, sizeof(DATA_TYPE) * NI * NK);
7   ... // Other allocations
8
9   cudaMemcpy(A_gpu, A, sizeof(DATA_TYPE) * NI * NK,
10                      cudaMemcpyHostToDevice);
11  .. // Other Memory transfers
12
13  dim3 block(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
14  dim3 grid1(DIM_GRID_X, DIM_GRID_Y);
15
16  mm3_kernel1<<<grid1,block>>>(A_gpu, B_gpu, E_gpu);
17  cudaThreadSynchronize();
18
19  mm3_kernel2<<<grid2,block>>>(C_gpu, D_gpu, F_gpu);
20  #pragma redundant in C_gpu, D_gpu out F_gpu
21  cudaThreadSynchronize();
22
23  mm3_kernel3<<<grid3,block>>>(E_gpu, F_gpu, G_gpu);
24  cudaThreadSynchronize();
25
26  }
```

Kod 1: Kodun ilk hali.

```
1   __global__ void majorityVoting2(float *d_A1,
2                                   float *d_A2,
3                                   float *d_A3,
4                                   float *Output,
5                                   int size){
6       unsigned int i = blockDim.x * blockIdx.x
7                          + threadIdx.x;
8       if(i < size){
9         if(d_A1[i] == d_A2[i]){
10          Output[i] = d_A1[i]
11        }else{
12          Output[i] = d_A3[i]
13        }
14      }
15  }
16
17  ...
18  DATA_TYPE *A_gpu;
19  ... // Other declarations
20
21  cudaMalloc((void **)&A_gpu, sizeof(DATA_TYPE) * NI * NK);
22  ... // Other allocations
23
24  cudaMemcpy(A_gpu, A, sizeof(DATA_TYPE) * NI * NK,
25                             cudaMemcpyHostToDevice);
26  .. // Other Memory transfers
27
28  dim3 block(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
29  dim3 grid1(DIM_GRID_X, DIM_GRID_Y);
30
31  mm3_kernel1<<<grid1,block>>>(A_gpu, B_gpu, E_gpu);
32  cudaThreadSynchronize();
33
34
35  mm3_kernel2<<<grid2,block>>>(C_gpu, D_gpu, F_gpu);
36  DATA_TYPE *F_gpu1;
37  cudaMalloc((void **)&F_gpu1,
38                sizeof(DATA_TYPE) * NI * NK);
39  cudaMemcpy(F_gpu1, F, sizeof(DATA_TYPE) * NI * NK,
40                        cudaMemcpyDeviceToDevice)
41
42  mm3_kernel2<<<grid2,block>>>(C_gpu, D_gpu,
43                                  F_gpu1);
44
45
46
47  DATA_TYPE *F_gpu2;
48  cudaMalloc((void **)&F_gpu2,
49                sizeof(DATA_TYPE) * NI * NK);
50  cudaMemcpy(F_gpu2 F, sizeof(DATA_TYPE) * NI * NK,
51                  cudaMemcpyDeviceToDevice)
52  mm3_kernel2<<<grid2,block>>>(C_gpu, D_gpu,
53  F_gpu2);
54  cudaThreadSynchronize();
55
56
57  majorityVoting<<<grid,block>>>(F_gpu, F_gpu1,
58      F_gpu2, F_gpu,   NI * NK);
59
60
61  mm3_kernel3<<<grid3,block>>>(E_gpu, F_gpu, G_gpu);
62  cudaThreadSynchronize();
63
64  }
```

Kod 2: Hedeflenen kod.

3.3.2 Derleyici Direktifi

Programlayıcıdan kernel fonksiyonunun kopyalanmasını sağlamak için derleyici şemamız bir derleyici yönergesi tanımlar. Yönergeyi kullanarak, programcı fonksiyonun yanı sıra giriş ve çıkış değişkenlerine açıklama ekleyebilir. Yönergenin sözdizimi aşağıdaki gibidir:

#pragma redundant in <input> *out* <output>

Kod 3, özelleştirilmiş Clang'ın Kod 4'de verilen hedef kod için işaretlenmiş fonksiyon çağrısını nasıl bir kod oluşturduğunu gösterir. İşaretlenmiş fonksiyon çağrısı, fonksiyon çağrısının hem girdisini hem de çıktısı gösteren Redundancy meta verilerine sahiptir. Ekli meta veriler, optimize edici aşamasında kullanılabilir.

```
void mm3Cuda(int ni,
             ... // Other parameters
             )
{
    DATA_TYPE *A_gpu;
    ... // Other declarations

    cudaMalloc((void **)&A_gpu, sizeof(DATA_TYPE) * NI * NK);
    ... // Other allocations

    cudaMemcpy(A_gpu, A, sizeof(DATA_TYPE) * NI * NK,
                          cudaMemcpyHostToDevice);
    .. // Other Memory transfers

    dim3 block(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
    dim3 grid1(DIM_GRID_X, DIM_GRID_Y);

    mm3_kernel1<<<grid1,block>>>(A_gpu, B_gpu, E_gpu);
    cudaThreadSynchronize();

    mm3_kernel2<<<grid2,block>>>(C_gpu, D_gpu, F_gpu);
    #pragma redundant in C_gpu, D_gpu out F_gpu

    cudaThreadSynchronize();

    mm3_kernel3<<<grid3,block>>>(E_gpu, F_gpu, G_gpu);
    cudaThreadSynchronize();
}
```

Kod 3: İşaretlenmiş CUDA kodu.

```
%call42 = call i32 @cudaConfigureCall(i64 %120, i32 %122,
                                    i64 %126, i32 %128,
                                    i64 0,
                                %struct.CUstream_st* null)
%tobool43 = icmp ne i32 %call42, 0
br i1 %tobool43, label %kcall.end45,
                 label %kcall.configok44

kcall.configok44:
%139 = load i32, i32* %ni.addr, align 4
%140 = load i32, i32* %nj.addr, align 4
%141 = load i32, i32* %nk.addr, align 4
%142 = load i32, i32* %nl.addr, align 4
%143 = load i32, i32* %nm.addr, align 4
%144 = load float*, float** %C_gpu, align 8
%145 = load float*, float** %D_gpu, align 8
%146 = load float*, float** %F_gpu, align 8
call void @_Z11mm3_kernel2iiiiiPfS_S_(i32 %139, i32 %140,
                                    i32 %141, i32 %142,
                                    i32 %143, float* %144,
                                    float* %145, float* %146),
                                    !Redundancy !3
br label %kcall.end45
...
!3 = !{!"Inputs &C_gpu&D_gpu Outputs &F_gpu"}
```

Kod 4: Kodun derlenmiş hali.

3.3.3 Çıktı Çoklaması

Tüm fonksiyon yürütmelerinin, yani bir orijinal ve iki çoklanmış yürütmenin sonuç değerlerini depolamak için, işaretlenmiş GPU kernel fonksiyonunun çıktı değişkenini çokluyoruz. Bu aşama üç adımdan oluşur:

1. Değişken bildirimi: Kernel fonksiyonunun CPU kodunda çoğaltılacak çıktı değişkeninin türünde iki değişken daha tanımlıyoruz.
2. Bellek ayırma: cudaMalloc fonksiyonunun çağrılarını kullanarak GPU'da bellek ayırırız.
3. Başlatma: Hedef fonksiyon yürütmelerinde başlangıç değerlerinin kullanılması nedeniyle çıktı değişkenlerini başlatmamız gerektiğinden, veri değerlerini orijinal çıktı değişkeninden çoklanmış değişkenlere kopyalarız. GPU belleğinde başlangıç değerlere sahip olduğumuz için, ana bilgisayar CPU'dan GPU aygıtına kopyalamadan kaynaklanan ek yükü önlemek için cihazdan cihaza bellek kopyalama işlemini (cudaMemcpyDevicetoDevice parametresi aracılığıyla) kullanırız.

3.3.4 Kernel Fonksiyonunun Çoklanması

Amacımız, hedef kernel fonksiyonunun çoklanmış şekilde yürütmek olduğundan, orijinaline ek olarak iki fonksiyon çağrısı daha ekledik. Daha önce oluşturulan çıktı değişkenlerinin çoklanmış kopyalarını, çoklanmış fonksiyon çağrılarının çıktı parametreleri olarak sağlarız. Izgara boyutu, blok boyutu ve giriş değişkenleri dahil olmak üzere diğer parametreler, orijinal fonksiyon çağrısıyla aynı kalır.

3.3.5 Çoğunluk Oylaması Gerçeklemesi

Kernel fonksiyonlarının çoklanmış kopyalarını çoklanmış çıktı değişkenleriyle çalıştırdıktan sonra, olası hataları tespit etmek ve düzeltmek için sonuçlarını karşılaştırmamız gerekir. Bu nedenle, çoklanmış fonksiyon yürütmeleri tarafından oluşturulan üç çıktıyı karşılaştırarak tek bir çıktı üreten bir çoğunluk oylama fonksiyonu uygularız. Çıktılar zaten GPU'nun global belleğinde olduğundan ve paralel yürütme, karşılaştırma işlemlerini hızlandırabileceğinden, çoğunluk oylama fonksiyonunu bir GPU kernel fonksiyonu olarak uygularız. Derleyici uygulamamızda hem ana makine kodunu hem de cihaz kodunu değiştirmemiz gerekiyor. Cihaz kodu, işlev gövdesinin eklenmesiyle güncellenirken, değiştirilen ana bilgisayar kodu, bağımsız değişken kurulumuyla birlikte çoğunluk oylama fonksiyonu çağrısını içerir. Bir GPU uygulaması birden çok veri türü içerebileceğinden, birden çok çoğunluk işlevi uygulamasına ihtiyacımız var. Çoğunluk oylaması işlevi adına çıktı türünün yerleşik tip kodunu kullanırız ve çıktının veri türüne bağlı olarak uygun olanı çağırırız.

Çoğunluk oylama fonksiyonumuz beş parametreden oluşur. Bunlardan ilk üçü, kernel fonksiyon yürütmelerinin çıktılarıdır. Kalan sonuç çıktısı ve işaretlenmiş çıktının boyutudur. Çoğunluk oylama fonksiyonumuzun iki adımı vardır: İş parçacığının numarasının hesaplaması ve Karşılaştırma. İş parçacığının numarasının hesaplaması, görece basit bir işlemdir. Sonraki adım, Karşılaştırma, en az iki değerin birbirine eşit olduğu varsayımına dayanır. Bu nedenle sadece bir karşılaştırma yeterli olacaktır. Karşılaştırma, birinci ve ikinci çıktı arasındadır. Birbirlerine eşitlerse, çıktıya ilk değeri atayacağız. Değillerse, varsayımımıza dayanarak üçüncü değerin ikisinden birine eşit olacağını söyleyebiliriz. Bu nedenle, daha fazla karşılaştırma yapmadan çıktıya üçüncü değeri atayabiliriz.

4 BULGULAR

Bu bölümde önceki bölümlerde anlatılan yöntemlerimizi uygulayarak gerçekleştirdiğimiz testlerimizin sonuçları ve test sonuçlarıyla ilgili açıklamalar verilmektedir. Öncelikle Bölüm 4.1'de deneylerimizi yürüttüğümüz GPU ortamlarının özellikleri ve GPGPU uygulamalarının özellikleri verilmektedir. Bölüm 4.2'da önceki bölümde yöntemleri açıklanan çalışmalarımıza ait deney sonuçları ve açıklamaları sunulmaktadır.

4.1 Test Ortamı

4.1.1 GPU Ortamları

Projemize ait deneyler, Pascal mimarisine sahip GPU cihazlarında ve GPGPU-Sim simulatör ortamında gerçekleştirilmiştir. Pascal mimarisine sahip GPU cihazlarımız, kurumumuzda var olan 2xXeon Silver 4114 işlemci ve 32 GB hafıza içeren iş istasyonundaki NVIDIA Quadro P4000 GPU cihazı ve NVIDIA Quadro P620'dir. Tahminleme çalışmamız için ayrıca GPGPU-Sim simulatörü versiyon 4.0 kullanılarak gerçek GPU cihazı ve simulatör ortamı arasında bir karşılaştırma çalışması yapılmıştır.

4.1.2 GPGPU Benchmark Uygulamaları

Deneylerimizde kullanmak üzere Polybench benchmark setinden (Grauer-Gray vd. (2012)) 12 CUDA uygulamasıı seçilmiştir. Uygulamaların özelliklerine göre geliştirdiğimiz yapılar için farklı uygulama alt kümeleri seçilerek deneyler gerçekleştirilmiştir. Tablo 3'te deneylerimizi çalıştırdığımız uygulamalar ve tanımları verilmektedir. Farklı testlerimizde farklı uygulama alt kümeleri dahil edilmiştir.

4.2 Test Sonuçları

4.2.1 Bölgesel Hata Hassasiyet Analizi Aracı ile GPGPU Uygulamalarının Hata Hassasiyetlerinin Elde Edilmesi

Geliştirdiğimiz bölgesel hata hassasiyeti aracımızla farklı özelliklere sahip GPGPU uygulamalarının geçici hatalara olan hassasiyetleri ölçülerek sonuçlar analiz edilmiştir. Sessiz Veri Bozunumu (silent

Tablo 3: Polybench benchmark setinden seçtiğimiz uygulamalar.

Uygulama Adı	Açıklaması
2DConvolution	2 boyutlu evrişim operasyonu
3DConvolution	3 boyutlu evrişim operasyonu
3mm	3 matris çarpımı
Atax	Matris devriği ile and the matirs-vektör çarpımının çarpımı
BicG	BiCGSTAB (BiConjugate Gradient STA-Bilized method) gerçeklemesi
Correlation	Pearson's korelasyon katsayılarının hesaplanması
Covariance	Kovaryans hesaplaması
Fdtd2d	Simplified Finite-Difference Time-Domain metodunun gerçeklemesi
Gramschmidt	QR ayrıştırmasının Modified Gram Schmidt ile gerçeklemesi
Gemm	Genel matirs çarpımı hesaplaması
Gesummv	Scalar, vektör ve matris çarpımı
Mvt	Matris vektör çarpımı and devriği

data corruption-SDC) oranı, geçici hata hassasiyet metriği olarak kullanılmakta olup bazı uygulamaların sonuçlarının yanlış olarak değerlendirilmeleri farklı metriklerle ifade edilebilmektedir. Bu yüzden sessiz veri bozunumuyla sonuçlanan hata enjeksiyonu deneylerimizde SDC oranı yerine, veri bozunumunun kritikliği değerlendirilmiştir. Bu kritiklik, farklı uygulamalar için farklı şekillerde değerlendirilmektedir:

- **Çıktı Bozunumu Oranı (Output Corruption Rate):** Matris operasyonları içeren ve çıktı olarak matris üreten uygulamalarda, ham SDC oranları yerine yanlış hesaplanan matris elemanı sayısı hata hassasiyeti metriği olarak kullanlmıştır. İmge işleme uygulamaları gibi bazı uygulamalarda, bir miktar piksel değerinin (matris elemanının) yanlış hesaplanmasına rağmen doğru sonuç ürettiği varsayılabildiği için şu şekilde tanımladığımız Çıktı Bozunumu Oranı hata kritikliği metriği olarak kullanılmıştır:

$$Cikti\ bozunumu\ orani = \frac{yanlis\ hesaplanan\ matris\ elemani\ sayisi}{matris\ boyutu}$$

- **Mutlak Hata (Absolute Error):** Çıktı olarak tek bir değer üreten uygulamalar için bek-

lenen değer ile gözlemlenen değer arasındaki farkı ifade eden Mutlak Hata (Absolute Error) hata kritikliği metriği olarak kullanılmıştır:

$$Mutlak\ hata = |\ beklenen\ deger - gozlemlenen\ deger\ |$$

- **Mutlak Hata Ortalaması (Mean Absolute Error):** Çıktı olarak birden fazla değer üreten uygulamalar için beklenen değer ile gözlemlenen değer arasındaki farkların ortalamasını ifade eden Mutlak Hata Ortalaması (Mean Absolute Error) hata kritikliği metriği olarak kullanılmıştır:

$$Mutlak\ hata\ ortalamasi\ (MAE) = \frac{1}{N}\sum_{1}^{N} |\ beklenen\ deger - gozlemlenen\ deger\ |$$

N hesaplanan değerleri ifade etmektedir.

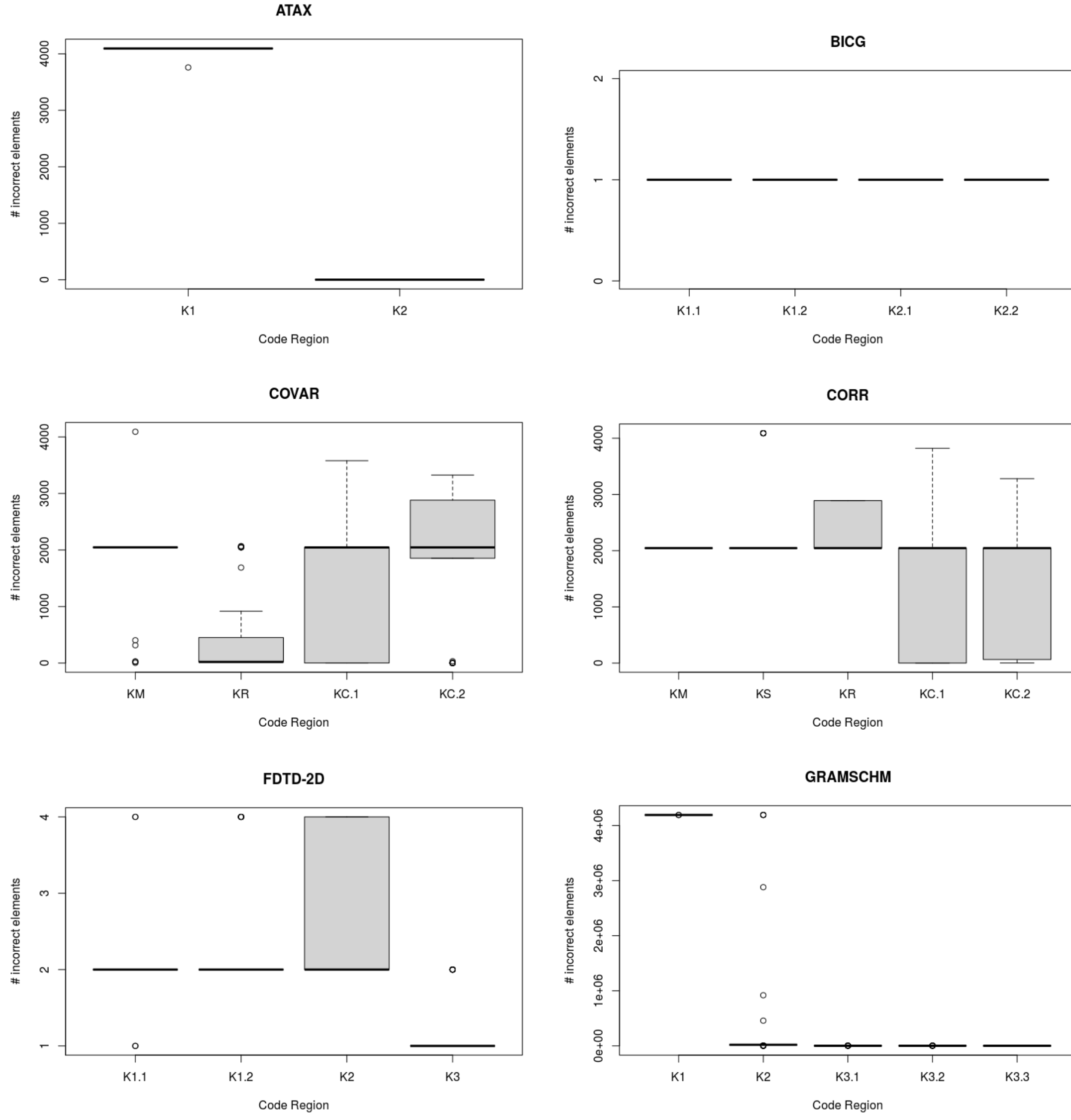
Hedef uygulamaların ürettiği çıktılara göre, matris operasyonları içeren uygulamalar için çıktı bozunumu oranı ve mutlak hata ortalaması, tek bir değer üreten uygulamalar için mutlak hata metriklerini kullanarak hata hassasiyet analizleri yapılmıştır. Şekil 9 sessiz hata bozunumu durumlarındaki yanlış hesaplanan matris elemanı sayılarının boxplot gösterimlerini içermektedir. İlerleyen bölümlerde hata enjeksiyonu deneylerini gerçekleştirdiğimiz GPGPU uygulamaları tek tek incelenerek hata hassasiyetleri ve hata yayılımlarının açıklamaları verilmektedir.

ATAX: Bu uygulama, matris devriği (matrix transpose) ve matris-vektör çarpımının (matrix-vector multiply) çarpımını (A^T times Ax) hesaplamaktadır. Uygulamadaki ilk kernel fonksiyonu matris-vektör çarpımını, ikinci kernel fonksiyonu matris devriğini hesaplamaktadır:

```
__global__ void atax_kernel1(float *A, float *x, float *tmp){
1:   int i = blockIdx.x * blockDim.x + threadIdx.x;
2:   if (i < size){
3:        int j;
4:        for(j=0; j < size; j++){
5:             tmp[i] += A[i * size + j] * x[j];
6:        }
7:   }
}
__global__ void atax_kernel2(float *A, float *y, float *tmp){
1:   int j = blockIdx.x * blockDim.x + threadIdx.x;
2:   if (j < size){
3:        int i;
4:        for(i=0; i < size; i++){
5:             y[j] += A[i * size + j] * tmp[i];
6:        }
7:   }
}
```

Analizimiz kapsamında iki kernel fonksiyonuna (K1 - $atax_kernel1$ ve K2 - $atax_kernel2$) hata enjeksiyonu yapılmıştır. K2 fonksiyonu için, her hata bozunumu durumunda tek bir elemanın değerinin yanlış hesaplanmasıyla sonuçlanmaktadır. K1 fonksiyonu içinse, çoğu durumda tüm çıktı elemanları (4096 adet) yanlış hesaplanmakta, bazı durumlarda daha az sayıdaki elemanın değeri bozulmaktadır. İlk kernel fonksiyonu çalışırken gerçekleşebilecek bir hata, uygulamanın çıktısını daha ciddi bir şekilde etkilemektedir. Ancak ikinci kernel sırasındaki bir hata sadece tek bir elemanı bozabilecektir. $atax_kernel1$ fonksiyonu çalışırken tmp dizisindeki tek bir eleman yanlış hesaplanmaktadır, ancak bu yanlış hesaplama y dizisindeki tüm elemanları bozabilmektedir. Yanlış hesaplamadaki değer büyüklüğüne göre, hata çıktı elemanlarında görünür olabilmektedir. Yani tmp dizisindeki çok küçük bozunma y dizisinin tüm elemanlarının hesaplanmasında görünür olmayabilmektedir (Şekil 9-ATAX' taki daire olarak gösterilen durumlar).

BICG: Bu uygulama, lineer sistemlerin nümerik çözümleri için bir yinelemeli yöntem olan BiCGSTAB (BiConjugate Gradient STABilized method)'ın gerçeklemesidir. Uygulamada birbirinden bağımsız matris-vektör çarpım işlemleri içeren ve çıktı olarak iki farklı matris hesaplayan iki kernel fonksiyonu bulunmaktadır. Her bir kernel fonksiyonu için biri dizi ilklendirme, diğeri çarpma işlemlerinin yapıldığı kısım olmak üzere iki hata enjeksiyon noktası belirlenerek toplamda uygulamanın dört farklı noktasına hata enjeksionu yapılmıştır (K1.1, K1.2, K2.1, K2.2). Sonuç olarak, hesaplamalar birbirinden bağımsız olduğu için ve kernel fonksiyonlarında hiç veri tekrar kullanımı



Şekil 9: Kernel fonksiyonlarının yanlış hesaplanan eleman sayılarına ait boxplot gösterimleri.

olmadığından, tüm SDC durumları sadece bir çıktı matrisi elemanının bozulması ile sonuçlanmıştır.

COVAR: Bu uygulama, iki değişkenin lineer olarak ilişkisini istatistiksel olarak gösteren kovaryans (covariance) değerini hesaplamaktadır. Kovaryans değeri, x ve y değişkenlerinin sapmalarının çarpımı olarak tanımlanmaktadır:

$$\sigma_{x,y} = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{N-1}$$

Görüldüğü üzere, kovaryans hesaplaması üç temel işlem içermektedir: 1) Veri elemanlarının ortalaması (average), 2) Veri elemanlarının ortalamadan sapması, 3) Bu sapmaların çarpımlarının toplamı. Polybench setinde bu işlemler üç farklı kernel fonksiyonu olarak gerçeklenmiştir: *mean_kernel*, *reduce_kernel*, *covar_kernel*.

```
__global__ void mean_kernel(float *mean, float *data){
1:  int j = blockIdx.x * blockDim.x + threadIdx.x + 1;
    ...
2:  for(i = 1; i < (size+1); i++){
3:      mean[j] += data[i * (size+1) + j];
4:  }
    ...
}

__global__ void reduce_kernel(float *mean, float *data){
1:  int j = blockIdx.x * blockDim.x + threadIdx.x + 1;
2:  int i = blockIdx.y * blockDim.y + threadIdx.y + 1;
    ...
3:  data[i * (size+1) + j] -= mean[j];
}

__global__ void covar_kernel(float *symmat, float *data){
1:  int j1 = blockIdx.x * blockDim.x + threadIdx.x + 1;
    ...
2:  for (j2 = j1; j2 < (size+1); j2++){
3:      symmat[j1*(size+1) + j2] = 0.0;
4:      for(i = 1; i < (size+1); i++){
5:          symmat[j1 * (size+1) + j2] += data[i *(size+1) + j1] *
                                   data[i *(size+1) + j2];
6:      }
7:      symmat[j2 * (size+1) + j1] = symmat[j1 * (size+1) + j2];
    }
}
```

Hata enjeksiyonu deneylerimiz için dört farklı kod bölgesi tanımlanmıştır: bir adet *mean_kernel* için (KM), bir adet *reduce_kernel* için (KR), bir adet *covar_kernel* içindeki iç döngü için (KC.1), bir adet *covar_kernel* içindeki dış döngü için (KC.2). Şekil 9-COVAR'da görüldüğü üzere, KM bölgesine ait değerlerin varyansı daha küçükken, diğer bölgelerdeki değerler farklı SDC durumları için daha fazla farklılık göstermektedir. *mean_kernel* kernel fonksiyonunda hesaplanan veri, *reduce_kernel* kernel fonksiyonunda kullanıldığından çıktı verisinin bozunumunu direkt etkilemektedir. Daire ile gösterilen küçük farklılıklar hata enjeksiyonu sebebiyle meydana gelen bozunumun değerinin küçük olduğu ve çıktı değerindeki etkisinin küçük miktarlarda gözlendiği veya gözlenmediği durumları göstermektedir. İndeks değerlerindeki olası bozulmalardan dolayı *reduce_kernel* kernel fonksiyonunda daha fazla çifte eleman bozulmaları gözlemlenmiştir. İndeks değerlerinin *reduce_kernel* içindeki 3. satırda görüleceği üzere i, j ve *size* değişkenlerince belirlenen *data* dizisi indeks değerinin üç farklı registerda meydana gelebilecek hatalardan etkilenme ihtimali de daha yüksektir. *covar_kernel* kernel fonksiyonundaki yanlış hesaplanan eleman sayılarının median değerleri her iki bölge (KC.1 ve KC.2) için benzerdir.

CORR: Bu uygulama, normalize edilmiş kovaryans olan Pearson korelasyon katsayılarını hesaplamaktadır:

$$r_{x,y} = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{N}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{N}(y_i - \bar{y})^2}}$$

Korelasyon hesaplaması dört temel işlemden oluşmaktadır: 1) Veri elemanlarının ortalaması, 2) Veri elemanlarının standart sapması, 3) Veri elemanlarının standart sapmanın ortalamasından sapmasının bölümü, 4) Bu sapmaların toplamı. COVAR uygulamasına benzer olarak bu uygulamada bu dört farklı işlem dört kernel fonksiyonu ile gerçeklenmiştir: *mean_kernel*, *std_kernel*, *reduce_kernel*, *corr_kernel*. Bu uygulama için de benzer olarak hata enjeksiyon deneyleri uygulanmıştır: bir adet *mean_kernel* için (KM), bir adet *std_kernel* için (KS), bir adet *reduce_kernel* için (KR), bir adet *corr_kernel* içindeki iç döngü için (KC.1), bir adet *corr_kernel* içindeki dış dögü için (KC.2). *mean_kernel* ve *std_kernel* fonksiyonları farklı SDC durumları için benzer sonuçlar verirken, *reduce_kernel* ve *corr_kernel* daha farklı sonuçlar vermektedir.

FDTD-2D: Bu uygulama, Simplified Finite-Difference Time-Domain tekniğinin gerçeklemesidir ve üç kernel fonksiyonundan oluşmaktadır:

```
__global__ void fdtd_step1_kernel(float *_fict_, float *ex,
                        float *ey, float *hz, int t){
1:   int j = blockIdx.x * blockDim.x + threadIdx.x;
2:   int i = blockIdx.y * blockDim.y + threadIdx.y;
3:   if ((i < size) && (j < size)){
4:        if (i == 0){
5:             ey[i*size+j] = _fict_[t];
6:        }
7:        else{
8:             ey[i*size+j] = ey[i*size+j] - 0.5f*(hz[i*size+j] - hz[(i-1)*size+j]);
9:        }
10: }
}
__global__ void fdtd_step2_kernel(float *ex, float *ey, float *hz, int t){
1:   int j = blockIdx.x * blockDim.x + threadIdx.x;
2:   int i = blockIdx.y * blockDim.y + threadIdx.y;
3:   if ((i < size) && (j < size) && (j > 0)){
4:        ex[i*(size+1)+j] = ex[i*(size+1)+j] - 0.5f*(hz[i*size+j] - hz[i*size+(j-1)]);
5:   }
}

__global__ void fdtd_step3_kernel(float *ex, float *ey, float *hz, int t){
1:   int j = blockIdx.x * blockDim.x + threadIdx.x;
2:   int i = blockIdx.y * blockDim.y + threadIdx.y;
3:   if ((i < size) && (j < size)){
4:        hz[i*size+j] = hz[i*size+j] - 0.7f*(ex[i*(size+1)+(j+1)]
                      - ex[i*(size+1)+j] + ey[(i+1)*size+j] - ey[i*size+j]);
5:   }
}
```

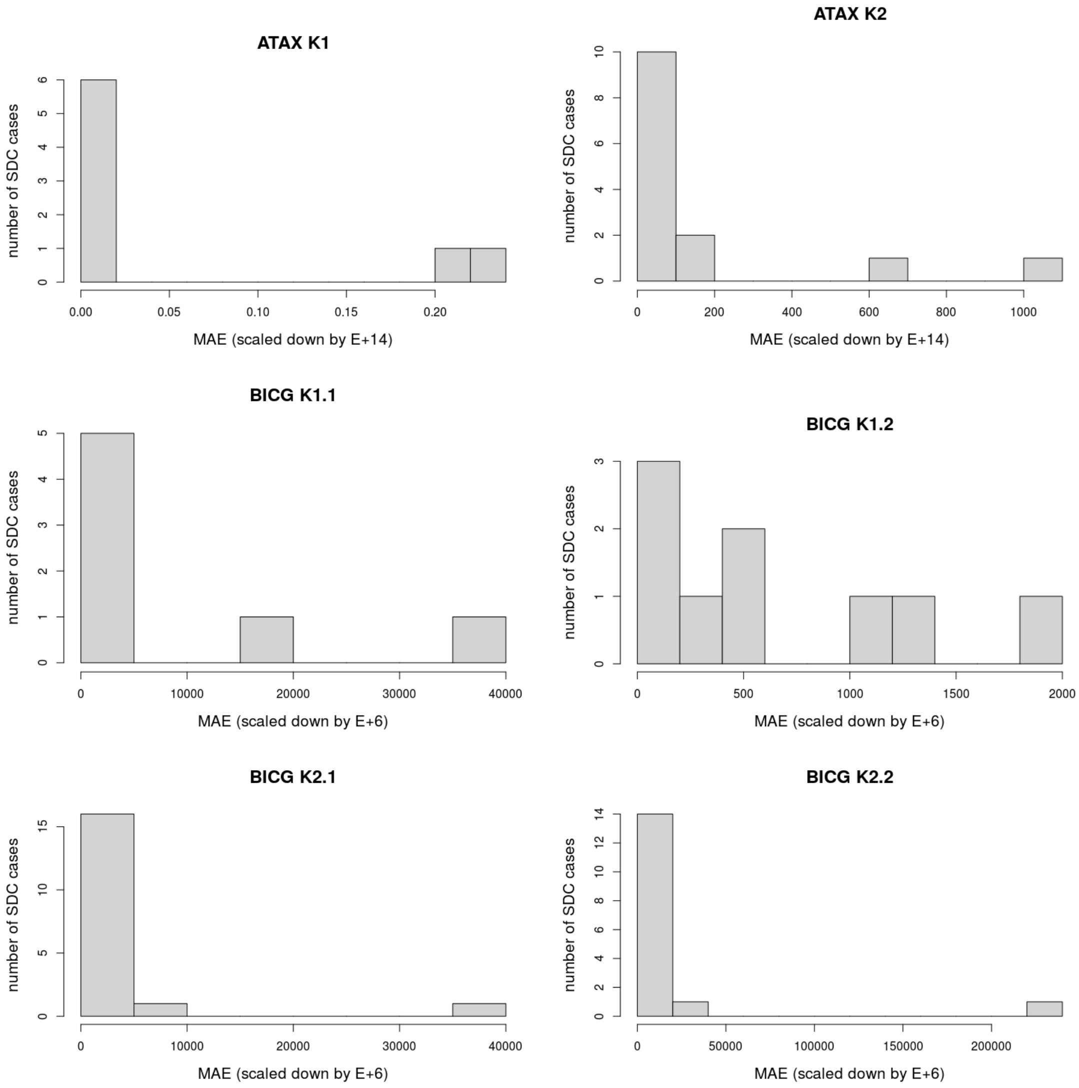
İlk kernel fonksiyonu *ey* değişkenini hesaplarken, ikinci kernel fonksiyonu ondan bağımsız olarak *ex* değişkenini güncellemektedir. Sonuç olarak son kernel fonksiyonu, bu iki değişkeni kullanarak sonuç değişkeni olan *hz* dizisinin değerlerini hesaplamaktadır. *ey* ve *ex* dizi elemanları iki farklı *hz* elemanının hesaplanmasında kullanıldığından üç kod bölgesindeki ($K1.1$, $K1.2$, $K2$) hatalar sonuç dizisinin iki elemanını bozmaktadır. Bununla birlikte son kernel fonksiyonunda (K3) herhangi bir hata yayılımı söz konusu olmadığından sadece tek bir çıktı elemanının bozulması söz konusudur.

GRAMSCHM: Bu uygulama, Modified Gram Schmidt yöntemiyle QR Decomposition hesaplamasının gerçeklemesidir ve üç kernel fonksiyonundan oluşmaktadır:

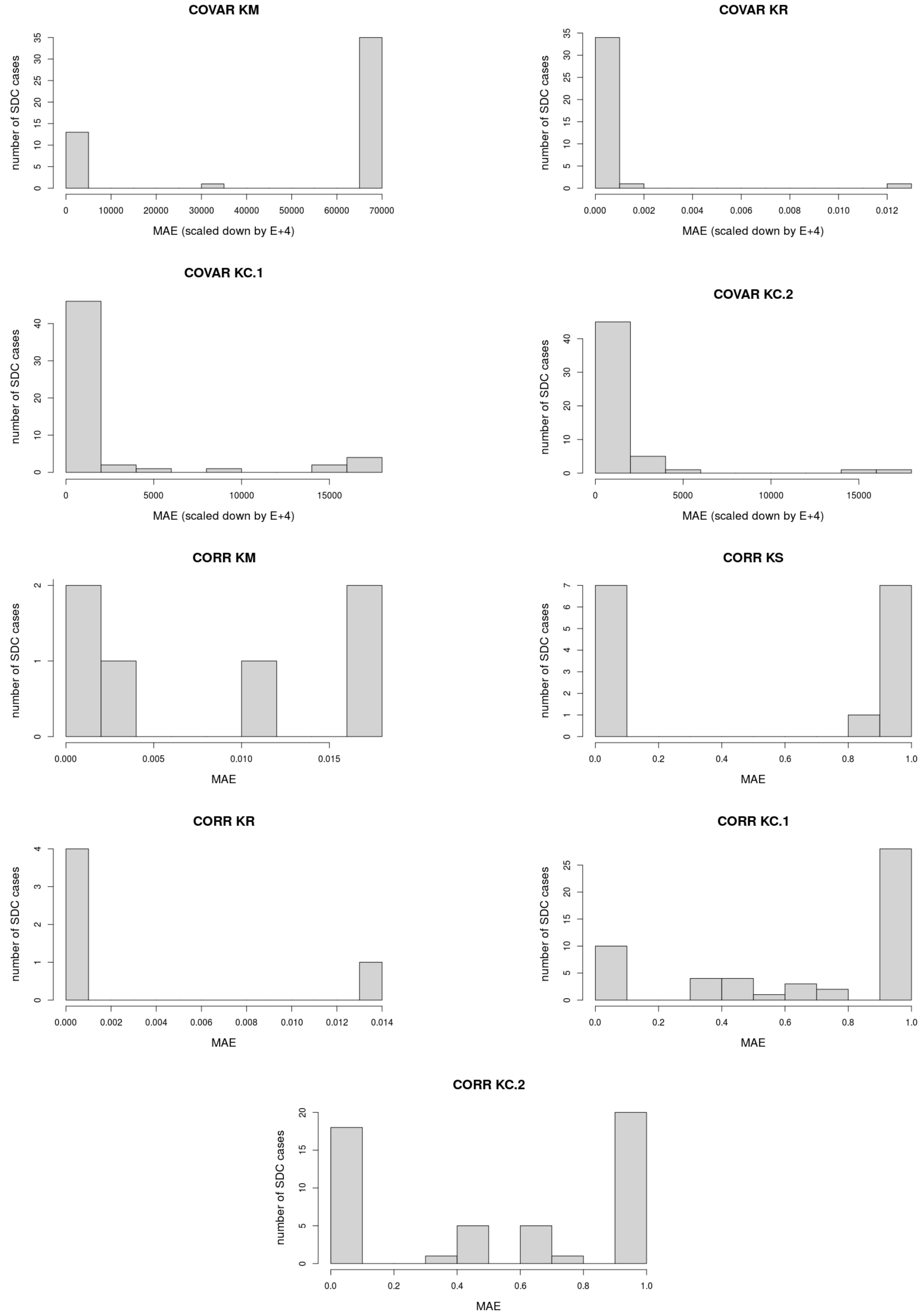
```
__global__ void gramschmidt_kernel1(float *a, float *r, float *q, int k){
1:  int tid = blockIdx.x * blockDim.x + threadIdx.x;
2:  if(tid==0){
3:      float nrm = 0.0;
4:      int i;
5:      for (i = 0; i < size; i++){
6:          nrm += a[i * size + k] * a[i * size + k];
7:      }
8:      r[k * size + k] = sqrt(nrm);
9:  }
}

__global__ void gramschmidt_kernel2(float *a, float *r, float *q, int k){
1:  int i = blockIdx.x * blockDim.x + threadIdx.x;
2:  if (i < size){
3:      q[i * size + k] = a[i * size + k] / r[k * size + k];
4:  }
}

__global__ void gramschmidt_kernel3(float *a, float *r, float *q, int k){
1:  int j = blockIdx.x * blockDim.x + threadIdx.x;
2:  if ((j > k) && (j < size)){
3:      r[k*size + j] = 0.0;
4:      int i;
5:      for (i = 0; i < size; i++){
6:          r[k*size + j] += q[i*size + k] * a[i*size + j];
7:      }
8:      for (i = 0; i < size; i++){
9:          a[i*size + j] -= q[i*size + k] * r[k*size + j];
10:     }
11: }
}
```

GRAMSCHM uygulamasında toplamda beş farklı kod bölgesine hata enjeksiyonu yapılmıştır. İlk kernel fonksiyonu ($gramschmidt_kernel1$ (K1)) tek bir iş parçacığı tarafından çalıştırıldığı için ($tid = 0$) ve tüm çıktı elemanlarının hesaplanmasında kullanılan nrm değerini hesapladığı için bu fonksiyonda oluşabilecek bir hata tüm çıktı elemanlarını bozmaktadır (toplamda 4192256). Diğer kernel fonksiyonlarındaki hesaplamaların çıktı matrisinde bu şekilde bir etkisi olmadığından, etkileri de daha farklıdır.

Çıktı dizisi elemanları olarak birden fazla değer hesaplayan uygulamaların hata durumları, mut-



Şekil 10: Mutlak hata ortalama (Mean Absolute Error (MAE)) histogram değerleri.



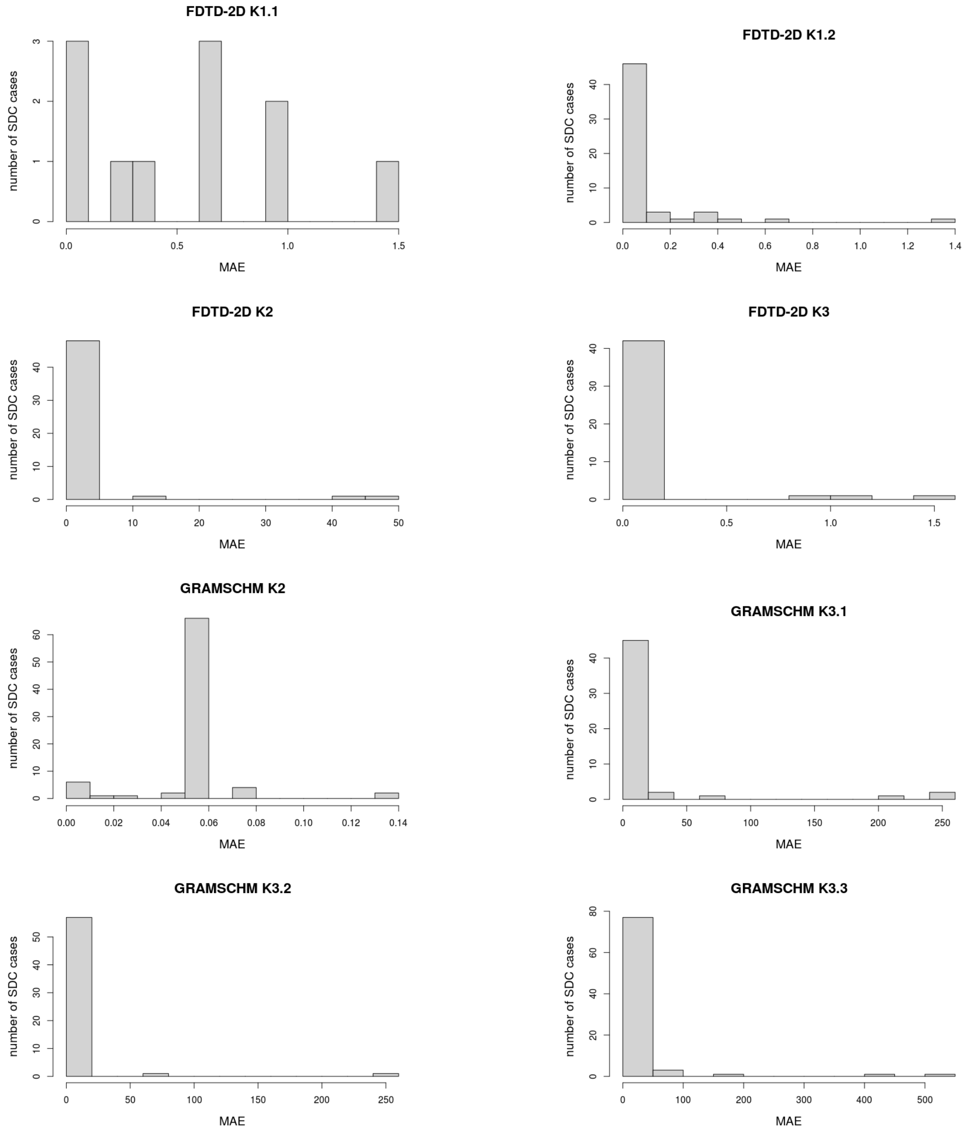
Şekil 11: Mutlak hata ortalama (Mean Absolute Error (MAE)) histogram değerleri.

lak hata ortalaması (MAE) değerleri ölçülerek daha detaylı incelenmiştir. Basitçe, önceden verilen yanlış hesaplanan elemanların ortalaması alınmıştır. Şekil 10, Şekil 11 ve Şekil 12 tüm hata enjeksiyonu yapılan bölgeler için gözlemlenen SDC durumlarına ait mutlak hata ortalaması değerlerini göstermektedir.

Bölgesel hata hassasiyeti ile ilgili gözlemler: Bölgesel hata enjeksiyonu deneylerimiz ve analizlerimiz neticesinde şu gözlem sonuçlarına ulaşılmıştır:

Gözlem 1: GPGPU uygulamalarındaki farklı kod bölgeleri farklı geçici hata hassasiyetlerine sahiptir. Farklı kernel fonksiyonları içindeki farklı bölgeler üzerinde bölgesel hata enjeksiyonları gerçekleştirerek, farklı kod bölgeleri için çeşitli geçici hata etkileri gözlemlenmiştir. Gerçekleştirilen işleme veya hedef kodun kullandığı verilere bağlı olarak, programın çıktısı çeşitli şekillerde etkilenebilmektedir. Sonuç olarak, kara kutu (black-box), kaba taneli (coarse-grained) hata enjeksiyon analizinin GPGPU programlarının hata hassasiyeti hakkında ayrıntılı bilgi vermediğini ve önlem alabilmek için veri bozulması kritiklik değerlendirmesi de dahil olmak üzere ince taneli bölgesel analizler yapmamız gerektiğini görüyoruz. Bu nedenle, daha ince taneli (fine-grained) hata hassasiyeti analizinin hata toleranslı hesaplama için daha fazla ayrıntı içerdiğini düşünüyoruz.

Gözlem 2: Kod bölgeleri, diğer kod bölgelerini (aynı kernel fonksiyonunda veya diğer kernel fonksiyonlarında) veya veri yapılarını, ve ayrıca GPGPU uygulamasının akışına ve veri kullanımına bağlı olarak nihai çıktıyı farklı şekillerde etkilemektedir. Deneysel hata yayılım analizimiz sayesinde, uygulama kodu ve verileri arasında bir hatanın yayılımı gözlemlenebilmektedir. Hedef programın özelliklerine ve hafıza erişim modeline bağlı olarak, uygulama verilerinin farklı bölümlerinde bir hata ortaya çıkabilmektedir. Bu yayılmanın kritikliği ve hızı, erişilen verilere ve programın yürütülmesi sırasında gerçekleştirilen işlemlere de bağlıdır. Örneğin, birkaç çıktı veri elemanının hesaplanması için kullanılacak herhangi bir veri elemanına isabet eden tek bir hata, birkaç çıktı öğesini bozabilmektedir. Ayrıca çok bitli hatalar, daha büyük miktarda program verisini bozarak program çıktısı üzerinde daha ciddi etkilere neden olabilir. Programlar, özellikle büyük miktarda veri ile uğraşan GPGPU programları, çeşitli veri bağımlılık ilişkileri sergilediği için; GPGPU program yürütmesi için hata yayılımını değerlendirmek daha fazla önem kazanmaktadır. Geçici hataların nihai program sonucu üzerindeki etkilerini anlamak çok önemli olmakla birlikte, program yürütme yoluyla hata yayılımını izlemek, hedef programın hata hassasiyeti davranışına ilişkin bilgi vermekte ve programlar arasında yayılma riskini azaltmanın yollarını düşünmemizi sağlamaktadır.



Şekil 12: Mutlak hata ortalama (Mean Absolute Error (MAE)) histogram değerleri.

Gözlem 3: GPGPU programlarındaki paralellik derecesi, yapılan işe veya CUDA iş parçacıkları tarafından kullanılan verilere bağlı olarak geçici hata hassasiyetini farklı şekillerde etkilemektedir. Deneysel analizimizden, mevcut registerları kullanan daha az iş parçacığımız varsa, veri bozulmasının daha olası olduğunu görüyoruz. Esasen, herhangi bir zamanda kullanılan toplam register sayısı, aynı anda yürütülen iş parçacıklarının sayısına bağlıdır. Bir iş parçacığı tarafından kullanılan register sayısı, bir CUDA programında paralelliği sınırlarken, geçici hata hassasiyeti için etkinin de incelenmesi gerekir. Programdaki bir iş parçacığının register kullanımını analiz etmek, programdaki paralellik hakkında bilgi sahibi olmadan, örneğin o iş parçacığına paralel olarak çalışan iş parçacıklarının sayısı hakkında bilgi sahibi olmadan yeterli olmayabilir. Bir iş parçacığı, verilerini tutmak için registerları yoğun olarak kullandığında (ancak mevcut registerların tümünü değil), ancak çok fazla paralel iş parçacığı olmadığında, genel hata hassasiyeti yüksek olmayabilir. Öte yandan, birkaç register kullanımıyla birçok eşzamanlı iş parçacığına sahip olduğumuz bir yürütmede, daha fazla genel register kullanımı nedeniyle daha büyük bir hata hassasiyetimiz olabilir.

Projemiz kapsamında geliştirdiğimiz bölgesel hata enjeksiyonu aracımızın GPGPU uygulamalarının hata hassasiyetlerini ölçerek farklı amaçlar için kullanılması mümkündür.

- **Kod özellikleriyle hata davranışının ilişkilendirilmesi:** Son zamanlarda, GPU programlarında yazılım hatalarını tahmin etmek için makine öğrenimi teknikleri öneren bazı çalışmalar olmuştur (Kalra vd. (2018); Palazzi vd. (2020)). Önerilen tahmin yöntemleri, HPC sistemleri (Jauk vd. (2019)) için hata tahminine dayalı çalışmalara benzer şekilde, ML modellerinde girdi özellikleri olarak program özelliklerini kullanır. Modeli eğitmek için, hafıza adres yönergeleri, aritmetik yönergeler, register kullanımı gibi program özelliklerini toplamakta ve SDC, çökme oranları gibi hata enjeksiyon sonuçlarını tahmin etmektedirler. Ne kadar çok veri toplarlarsa o kadar doğru tahminler elde edebilecekleri için; yürütmeyi daha küçük parçalara (bölgelere) bölmek ve her parça için veri toplamak, hedef ML modeli için veri noktalarının sayısını artırabilir. Bölgesel hata ekleme yöntemimiz, hedef CUDA programındaki herhangi bir kod parçası için hedef registerlara hatalar ekleyerek ince taneli hata enjeksiyon deneylerini kullanmaktadır. Geliştiriciler tarafından, programın tamamı (veya tüm kernel fonksiyonu) için hata enjeksiyonu gerçekleştirmek yerine, hedef program için birden fazla hata enjeksiyon deneyi yapmak ve ML model girdileri için birden çok veri noktası elde etmek için aracımız kullanılabilecektir. Böylece potansiyel olarak, modellerinin tahmin doğruluğu artacaktır. Bölüm 3.2'de projemizde geliştirdiğimiz tahminleme yönteminden bahsedilmiştir.
- **Seçmeli çoklu yürütme:** Yazılım çoklama teknikleri performans kayıpları getirdiğinden,

hataya en hassas parçaların kopyalanmasına dayanan seçici artıklık şemaları, güvenilirlik ve performans açısından en iyi kapsamı sağlayarak daha verimli yürütme kullanmaktadır (Dimitrov vd. (2009); Wadden vd. (2014b); Gupta vd. (2017)). Bölgesel hata enjeksiyonu aracımız, hedef programın en hataya hassas kısımlarını belirlemek için çoklu yürütme teknikleri tarafından kullanılabilir ve bu yöntemlerle gerçekleştirilen seçici çoklama sağlayabilir. Hedef programın farklı bölümlerinde hata enjeksiyon deneyleri yaparak, gözlem altındaki kod bölgelerinin kritikliğine karar verilebilir ve tam replikasyon yerine en kritik kısımlarda çoklama kullanma tercih edilebilir. Bölüm 3.3'te projemizde geliştirdiğimiz seçici çoklama yönteminden bahsedilmiştir.

- **Kısmi veri koruma:** Çoklu yürütmeye benzer şekilde, ECC veya parity gibi veri çoklamaya dayalı hata toleransı teknikleri, ek maliyet ve performans yükü yaratmaktadır. Register veya paylaşılan hafıza gibi özellikle yüksek oranda kullanılan hafıza yapılarının korunması çoğu zaman tercih edilmese de, bazı kritik sistemlerin bu düzeyde korumayı kullanması gerekebilir. Bir GPU sistemindeki tüm hafıza yapıları yerine, bunların bir alt kümesini korumak ve korunan kaynakları (register gibi) daha hassas işlemler için kullanmak, çoklamanın performans yükünü azaltcaktır.

- **Yaklaşık hesaplama yöntemleri:** %100 doğru çıktı gerektirmeyen uygulamalar için büyük performans giderleriyle başa çıkmak için, farklı seviyelerde yaklaşık hesaplama teknikleri uygulanmaktadır (Mittal (2016)). Veri bozunma kritiklik değerlendirmemiz, çıktı verilerinin farklı bölümleri üzerindeki veri bozulmalarını rapor ettiğinden, herhangi bir kod bölgesindeki bir hatadan (veya herhangi bir yanlış hesaplamadan) çıktının ne kadar etkilendiğini anlayabiliriz. Sonuç olarak, hata güvenilirliği açısından kritik olmayan programlar bile, yaklaşıklaştırılacak program bölümleri hakkında kararlar almak için çerçevemizi kullanabilir. Deneysel çalışmamızda gösterdiğimiz gibi, hata yayılım analizimiz, program yürütülürken program verilerinin izlenmesine olanak tanımaktadır. Belirli kod bölgelerinin yürütülmesi sırasında program verilerini yok eden hataları gözlemleyerek, daha az hassas program noktalarında yaklaşıklık yapmaya karar verilebilir. Daha az hassas, program çıktısı üzerinde daha az etkiye karşılık geldiği için, yürütme sırasında hesaplanan yanlış veya kesin olmayan değerlerden çıktının etkilenmeyeceği düşünülerek, bu kod bölgelerindeki hesaplamalar atlanabilir veya daha az hassasiyete sahip veri türleri kullanılabilir.

- **Yazılım geliştiricileri hata hassasiyeti konusunda yönlendirme:** Yazılım geliştiriciler öncelikle doğru kodu yazmayı amaçlarken, performans veya güvenilirlik, sırasıyla yüksek performanslı hesaplama ve güvenilirlik açısından kritik sistemlerde çalışan uygulamalar için önemli

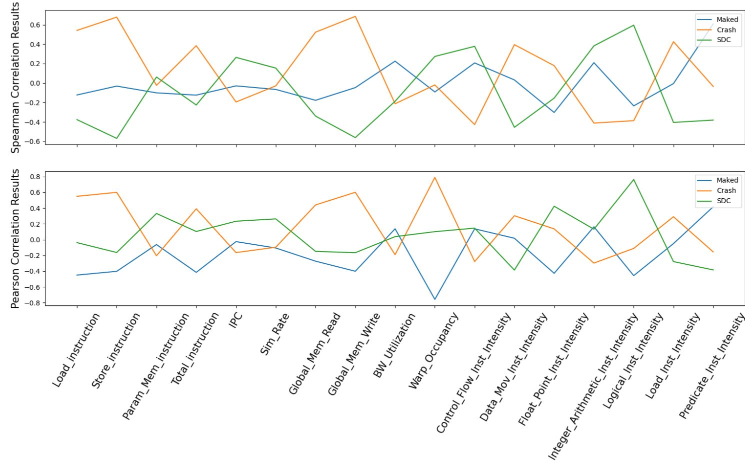
bir özellik olarak görülebilir. Programcılar, temel sistemin mimari ayrıntılarını bilmeyebileceklerinden, onlara çok daha hızlı veya daha güvenilir kod yazmaları için rehberlik edecek yüksek seviyeli talimatlar sağlamak, hedef programların gereksinimlerini karşılamaya iyi bir katkı olabilir. Yüksek performanslı ve verimli program geliştirme kılavuzları (Cook (2013); Kirk & mei W. Hwu (2017)) için önemli çalışmalar mevcut olsa da, literatürde güvenilir yürütme önerileri eksikliği vardır. Özellikle, ayrıntılı kod analizi ve daha güvenilir yürütme için hata hassasiyetine duyarlı öneriler sunan uygulamalar kullanılmamaktadır. Hata ayıklayıcı düzeyinde hata enjeksiyonu gerçekleştiren ve üst düzey program (örneğin, CUDA kernel fonksiyonları) ile temeldeki mimari kaynaklar (register dosyası gibi) arasında bir bağlantı sağlayan hata enjeksiyonu yapımız, programcıların programlarının daha savunmasız kısımlarını öğrenmelerine olanak tanır. Deneysel çalışmamızın sonuçlarından çıkarılan yönergelere göre, programcı daha güvenilir bir program geliştirmek için aracımızı kullanabilir. Örneğin, hedef kodda meydana gelen iş parçacığı sayısı ve paralellik, performansın yanı sıra güvenilirlik de dikkate alınarak ayarlanabilir. Gözlemlerimizde tartıştığımız gibi, daha yüksek doluluk daha yüksek performansla sonuçlanabilir, ancak aynı zamanda programı register dosyasındaki geçici hatalara karşı daha savunmasız hale getirir. Yazılım geliştirici hem performans hem de güvenilirlik hususlarını dikkate alarak, daha hızlı veya daha az savunmasız bir program arasında bir denge analizi yapabilir ve aracımız tarafından sağlanan yönergelere göre kararlar alabilir.

4.2.2 GPGPU Uygulamalarının Hata Hassasiyetlerinin Tahminleme Sonuçları

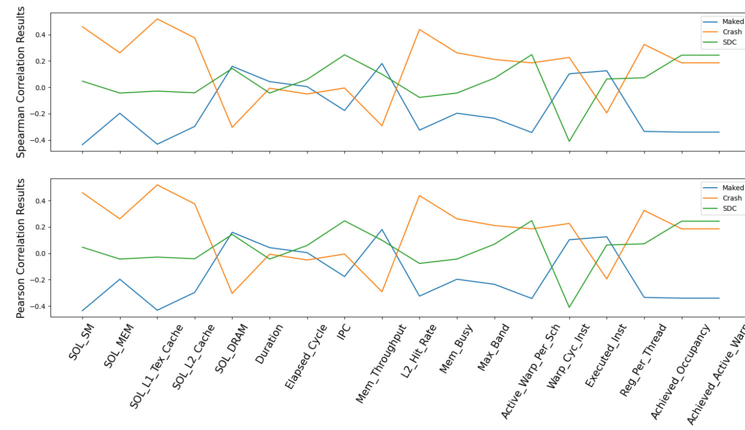
Bölüm 3.2'de bahsedildiği üzere, tahminleme modellerimize ait deneylerde 3 farklı makine öğrenme algoritması kullanılmıştır ve her algoritma 4 farklı şekilde hiper parametreleri değiştirilerek konfigüre edilmiştir. Bu algoritmalar, Sci-kit (Pedregosa vd. (2011)) kütüphanesinde implement edilmiş halleri ile kullanılmıştır. GB algoritmasındaki öğrenme oranlarını 0.001, 0.01, 0.1 ve 0.25 olarak, RF algoritması için rastgelelik parametresini 20, 50, inşa edici sayısını da 100, 1000 ve 10000 olarak kullanılmıştır. Bu durumlarda diğer parametreler default değerler olarak bırakılmıştır. SVM algoritması için, kernel fonksiyonu için 2. ve 3. derece polinom fonksiyonları, sigmoid eğri ve radyal eğri kullanılmıştır. Sınıflandırmada, her algoritma için maksimum ağaçların kollarının maksimum derinliği 2, toplam iterasyon sayısı 1000 olarak belirlenmiştir. Ek olarak, epsilon hata oranını da 0.001 olarak seçilmiş olup paralel işçi sayısı da 20 olarak belirlenmiştir. Bu hiper-parametrelerle GB, RF ve SS+SGD algoritmaları sınıflandırma tahminlemesinde kullanılmıştır.

Simülatör ve profil oluşturucudan donanımsal ve performansa ait metrikleri topladıktan sonra, Spearman ve Pearson korelasyon katsayılarını hesaplayarak hata tipleri ve metrikler arasındaki be-

lirleyici ilişkiye sahip özellikler bulunmuştur. Daha önce detaylı olarak açıkladığımız korelasyon değerleri için, bu bölümde sadece hatalar ve özellikler arasındaki korelasyon değerlerini inceleyeceğiz. Şekil 13 ve Şekil 14, hatalar ve metrikler arasındaki Spearman ve Pearson korelasyon sonuçlarını göstermektedir. Bu korelasyon sonuçlarıyla aynı doğrultuda, hem simülatör hem de profil oluşturucu metriklerinden gereksiz olanlar elenerek daha kesin sonuçlar gözlemlenmiştir.



Şekil 13: Simülatör metrikleri ve hata oranları arasındaki Spearman ve Pearson korelasyon sonuçları.



Şekil 14: Nsight Compute tool metrikleri ve hata oranları arasındaki Spearman ve Pearson korelasyon sonuçları.

Maskelenmiş hataları tahminlemek için regresyon yaklaşımı kullanılmıştır. Tablo 4 regresyon algoritmaları sonucunda gözlenmiş simülatör ve profil oluşturucu metrikleriyle yapılan çalışmaların

kesinlik sonuçlarını göstermektedir. Tüm makine öğrenmesi algoritmaları %90'ın üzerinde kesinlik sağlarken, GB algoritması %96.59 olan en yüksek kesinlik değerine sahiptir. Bir durum dışında (SVM algoritması ve seçilmiş özellikler deneyi), simülatör metrikleriyle Nsight Compute tool metriklerine göre daha kesin sonuçlar elde edilmiştir. Simülatör ve profil oluşturucu metriklerinin kullanıldığı regresyon tahminlemesi sonuçlarının çok farklı olmadığı gözlemlenmiştir.

Tablo 4: Maskelenmiş hatalar için tahminleme sonuçları.

Algoritma	**Simulatör metrikleri**		**Profilleme metrikleri**	
	Tüm metriklerle	**Seçilen metriklerle**	**Tüm metriklerle**	**Seçilen metriklerle**
RF	96.127	96.209	95.209	95.509
GB	96.463	**96.592**	95.671	95.812
SVM	92.111	92.105	91.747	94.814

SDC ve crash hatalarının tahminlenmesi için regresyon modelindeki gibi kesin değerleri tahmin etmek yerine, sınıflandırma modelleri inşa ettik ve hataların dahil olduğu sınıfları tahminlemeye çalıştık. Deneylerimizi iki ve üç sınıflı modeller oluşturarak yaptık. Deneylerde hataların dahil oldukları sınıfları bulmanın yanı sıra, tahminlenen sonuçlar için precision, recall ve F-score değerlerini inceleyerek, tahminin kesinliğini değerlendirdik. Gerçekte hataya yatkın olan (büyük SDC ve crash hata sonuçları) uygulamaları bulmak, bu hata oranlarının düşük olduğu durumları bulmaktan daha önemli olduğu için, precision ve recall değerlerini ilk durum için seçtik.

Tablo 5 ve Tablo 6 sırasıyla iki ve üç sınıflı değerlendirme sonuçlarını vermektedir. Tüm özellikleri kullandığımız deneylerde benzer ya da daha az kesin tahminleme sonuçları gözlediğimiz için, sadece seçilmiş özelliklerin kullanıldığı deney sonuçlarını paylaşırken tüm özelliklerin kullanıldığı deney sonuçlarını eklemedik. İki sınıflı SDC tahminlemesi deneyinde en yüksek sonuç olan %82.6 tahminleme sonucuna ulaştık ve bu sonuçta GB modelini kullandık. GB algoritmasını kullandığımız deneylerden simülatör metrikleri ile yapılan deney crash hataları için de daha yüksek değerle sonuçlanmıştır. GB algoritmasını kullanarak %80 kesinlik oranına ulaştığımız üç sınıflı SDC tahminleme deneyinde profil oluşturucu metrikleri kullanılmıştır ve üç sınıflı tahminleme için en yüksek oran budur. Yakın sonuç birleştirilmiş modelde (SS+SGD) %78.3 olarak gözlenirken, bu değer için precision ve recall değerleri son derece güvenilirdir. Simülatör metrikleri ile yaptığımız deney sonuçlarının profil oluşturucu metrikleri ile yaptığımız deney sonuçlarına göre daha kesin sonuçlar sunduğunu görebiliriz. Bu durumun arkasındaki sebep, profil oluşturucun sunmuş olduğu istatistiksel donanım metriklerinin uygulamaya ait komutlar hakkında açık bilgiler vermemesinden kaynaklanmaktadır. Bu yüzden, deneylerde donanıma ait metrikleri arttırdığımızda hata enjeksiyonu yapılan bölgeden uzaklaştığımız için, genel tahminleme ilişkisi de azalmaktadır.

Fakat, simülatör metrikler uygulamanın karakterizasyonunu yapmamız açısından daha karaktersel bilgiler vermesinin yanı sıra genelleyici donanımsal metrikler de sunmaktadır. Profil oluşturucudaki öncellek başarı/kaçırma oranlarının bağlantılı şekilde deneylerimizle ilgili olduğu düşünülse de, buna bağlı birçok konfigürasyon metriği vardır ki bu metrikler donanım limitlerinden bağımsız da olabilir. Dahası, hata enjeksiyon deneylerinde hedef registerlardır. Bu yüzden, GPU donanım metrikleri yerine uygulamanın registerları nasıl kullandığını anlayabilmek ve bu doğrultuda metrikler toplayabilmek deneyler için daha faydalı olmuştur. İki sınıflı SDC ve crash hatalarının sınıflandırma deneyler sırasıyla maksimum %82.6 ve %87 olarak gözlenmiştir. Bu hata değerleri üç sınıflı sınırlandırma deneylerinde sırasıyla %80 ve %60.9 olarak gözlenmiştir. Bununla birlikte, GB algoritması iki sınıflı sınıflandırma deneylerinde daha yüksek sonuçlara ulaşırken, RF ve birleştirilmiş (SS+SGD) sınıflandırma algoritmaları üç sınıflı sınıflandırma deneylerinde daha başarılı sonuçlar üretmiştir.

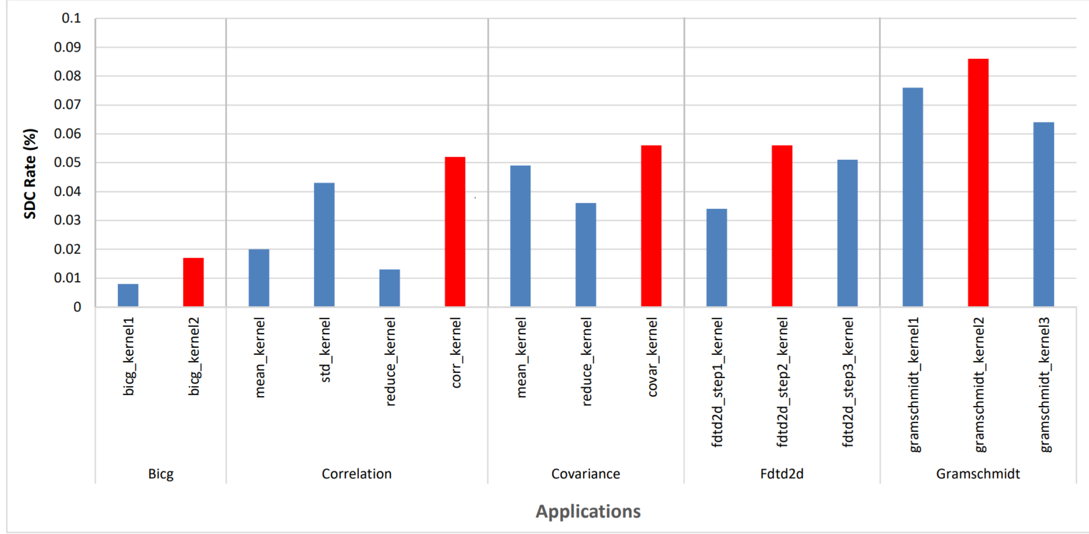
Tablo 5: 2-sınıf sınıflandırma için tahminleme sonuçları.

		Simulatör metrikleri				**Profilleme metrikleri**			
	Algoritma	Precision	Recall	F-Score	Accuracy	Precision	Recall	F-Score	Accuracy
SDC	RF	66.7	50.0	57.1	73.9	50.0	40.0	44.4	60.0
	GB	**75.0**	**75.0**	**75.0**	**82.6**	54.5	60.0	57.1	64.0
	SS+SGD	50.0	50.0	50.0	56.5	100.0	50.0	66.7	60.0
Crash	RF	81.2	86.7	83.9	78.3	77.8	82.4	80.0	72.0
	GB	**87.5**	**93.3**	**90.3**	**87.0**	68.4	76.5	72.2	60.0
	SS+SGD	68.8	73.3	71.0	69.6	75.0	70.6	72.7	68.0

Tablo 6: 3-sınıf sınıflandırma için tahminleme sonuçları.

		Simulatör metrikleri				**Profilleme metrikleri**			
	Algoritma	Precision	Recall	F-Score	Accuracy	Precision	Recall	F-Score	Accuracy
SDC	RF	50.0	25.0	33.3	60.9	x	x	x	68.0
	GB	75.0	75.0	75.0	69.6	**100.0**	**75.0**	**85.8**	**80.0**
	SS+SGD	**100.0**	**100.0**	**100.0**	**78.3**	33.3	25.0	28.7	60.0
Crash	RF	**69.2**	**81.8**	**75.0**	**60.9**	58.3	58.3	58.3	44.0
	GB	72.7	72.7	72.7	52.2	63.6	58.3	60.9	48.0
	SS+SGD	50.0	45.5	47.6	34.8	55.6	41.7	47.6	48.0

Sınıflandırma sonuçlarına göre bu metodu GPGPU programlarındaki hata tahminlemesi çalışmaları için uygundur diyebiliriz. Bu problemi hedeflenen programın hatalara karşı güvenilirliğini sınıflandırmak olarak değerlendirirsek, uygulamanın SDC ve crash hata oranlarının hangi aralıkta



Şekil 15: Kernel fonksiyonları için SDC değerleri.

olacağını tahminleyebiliriz. Bu değerlendirme bize programların hataya yatkınlıklarını anlamamızı sağlar.

Hem sınıflandırma hem de regresyon yöntemleri için, simülatör metrikleri ile yapılan deneylerin profil oluşturucu metrikleri ile yapılan deneylere göre daha kesin olduğu söylenebilir. Yanı sıra, bu kesinlik sonuçlarını da precision ve recall değerleri ile daha güvenilir hale getirdiğimizi söyleyebiliriz.

4.2.3 Seçimli Hata Toleransı Yöntemi ile Elde Edilen Performans ve Güvenilirlik Sonuçları

Seçimli hata toleransı yöntemimizi uygulayabilmek için öncelikle hedef programlarımızdaki hataya en açık kernel fonksiyon(lar)ını elde etmek için hata enjeksiyon deneyleri yapılmıştır. Deneylerimiz Masked, Crash ve SDC oranlarını rapor ederken, yürütmenin hata hassasiyeti ölçüsü olarak yalnızca SDC oranları kullanılmıştır. Şekil 15, hata enjeksiyon deneylerimizden ortaya çıkan her bir kernel fonksiyonu için SDC oranlarını göstermektedir. Ek olarak, Tablo 7, karşılık gelen kernel fonksiyonlarının yürütme sürelerini göstermektedir.

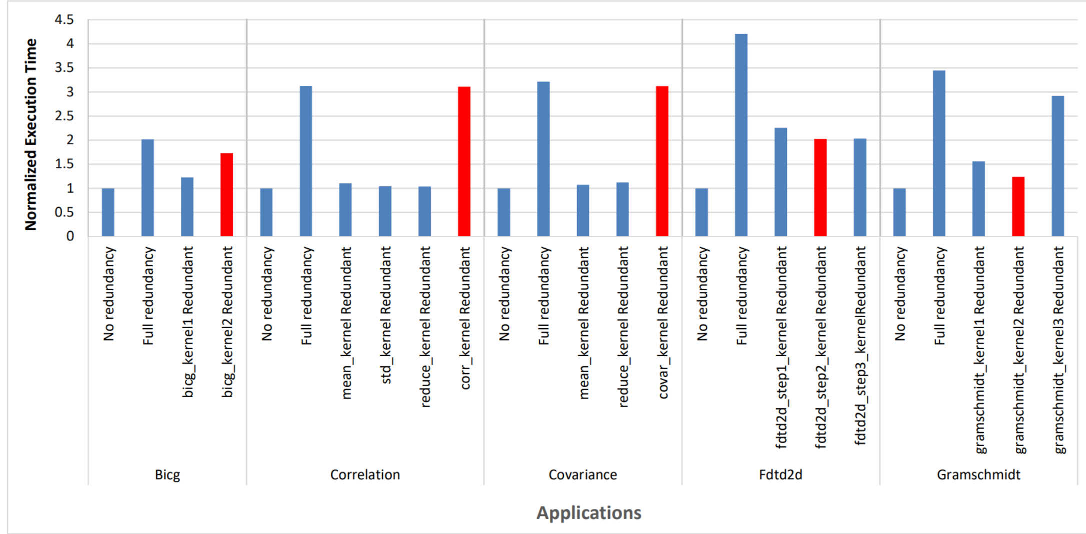
Kernel fonksiyonlarının hata hassasiyetlerini belirledikten sonra, her GPU uygulaması için en yüksek SDC oranına sahip ve Şekil 15'te kırmızı ile işaretlenmiş en hassas olanı seçiyoruz. Ardından, RMT iskeletimizi kullanarak farklı çoklama şemaları ile çoklanmış yürütmeler gerçekleştiriy-

Tablo 7: Kernel fonksiyonları için yürütme süreleri.

Uygulama Adı	Kernel Fonk. Adı	Yürütme Süresi (s)
Correlation	mean_kernel	0.003
	std_kernel	0.003
	reduce_kernel	0.004
	corr_kernel	5.945
Covariance	mean_kernel	0.003
	reduce_kernel	0.001
	covar_kernel	6.174
Bicg	bicg_kernel1	8.056 x 10^{-3}
	bicg_kernel2	22.496 x 10^{-3}
Fdtd2d	fdtd2d_step1_kernel	8.056 x 10^{-3}
	fdtd2d_step2_kernel	2.703 x 10^{-3}
	fdtd2d_step3_kernel	3.043 x 10^{-3}
Gramschmidt	gramschmidt_kernel1	0.922 x 10^{-3}
	gramschmidt_kernel2	0.008 x10^{-3}
	gramschmidt_kernel3	4.344 x 10^{-3}

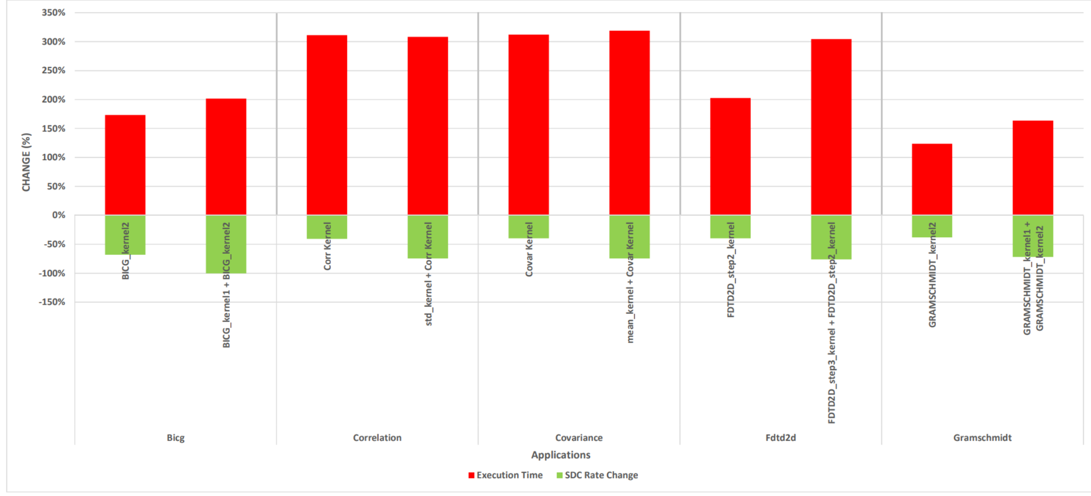
oruz. Spesifik olarak hedef programlarımızı şu senaryolarla yürütüyoruz: 1) Programı olduğu gibi çalıştırma (*No Redundancy*), 2) Programdaki tüm kernel fonksiyonlarını çoklanmış olarak, yani üç kere çalıştırma (*Full Redundancy*), 3) Yalnızca bir kernel fonksiyonu seçip onu çoklanmış olarak çalıştırdığımız bir kernel fonksiyonu çoklama.

Şekil 16, farklı çoklama şemalarına sahip hedef uygulamalarımızın normalleştirilmiş yürütme sürelerini göstermektedir. En yüksek SDC oranına sahip kernel fonksiyonunun şekilde her program için kopyalandığı yürütme senaryosunu işaretliyoruz, örneğin, *Bicg* için *bicg_kernel2*, *Correlation* için *corr_kernel*. En yüksek SDC oranına sahip fonksiyonu *bicg_kernel2* olan *Bicg*'e bakarsak, *bicg_kernel2*'nin çoklandığı yürütmenin, tam çoklanmış versiyonuna göre önemli ölçüde daha kısa sürdüğünü görebiliriz. Tam çoklanmış ile karşılaştırıldığında, seçici şemamız performans kazancı sağlamaktadır. Öte yandan, *Correlation* ve *Covariance* aynı şekilde davranmaz. En yüksek SDC oranına sahip *Correlation*'un *corr_kernel*'i aynı zamanda diğer kernel fonksiyonlara baskın olacak şekilde en yüksek yürütme süresine sahiptir. Bu nedenle, Tam çoklanmış ile yalnızca *corr_kernel* çoklanmış arasındaki farkın ihmal edilebilir olduğunu söyleyebiliriz. *corr_kernel* en yüksek SDC



Şekil 16: Çoklanmış uygulamaların normalleştirilmiş yürütme süreleri.

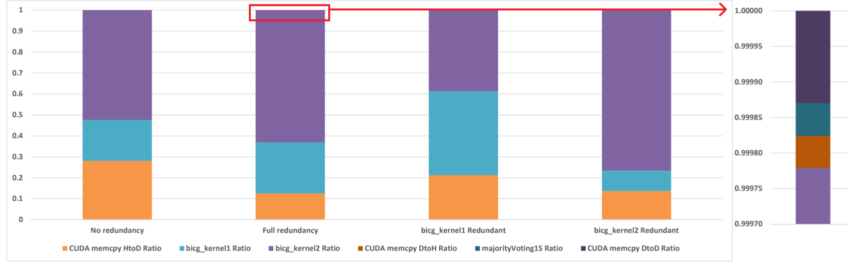
oranına sahip olsa da, *std_ kernel*'in SDC oranı çok farklı değildir (Şekil 15). *std_ kernel*'in yürütme süresi, *corr_ kernel*'in yürütme süresine kıyasla çok küçük olduğundan (Tablo 7) önemli performans düşüşü olmadan SDC oranını azaltmak için iki fonksiyonu da çoklamak faydalı olacaktır. Benzer şekilde, *Covariance*'da, diğer kernel fonksiyonların yürütme süreleri ihmal edilebilir olduğundan, *covar_ kernel* tüm uygulamaya baskındır. Öte yandan, *mean_ kernel*, Şekil 15'de gözlemlendiği gibi, *covar_ kernel*'e yakın SDC oranına sahiptir. Bu işlevlerin her ikisini de çoklarsak, ihmal edilebilir yürütme süresi farkıyla potansiyel olarak daha düşük hata hassasiyeti elde ederiz. Tam çoklanmışlık ile yalnızca *covar_ kernel* çoklanmış arasındaki zaman farkının çok küçük olduğunu unutmamalıyız. Bu nedenle en iyi çoklanmışlık seçeneğinin Tam çoklanmışlık olduğunu söyleyebiliriz. *Correlation* ve *Covariance*'daki eğilimleri gözlemleyerek, bir kernel fonksiyonunun yürütme süresi açısından diğerlerine baskın olduğu programlar için, seçici şemamızın yardımcı olmadığını ve tam çoklama uygulamanın daha yardımcı olabileceğini söyleyebiliriz. Diğer uygulamalardan farklı olarak, yürütme süresi ile SDC oranı arasındaki oran *Fdtd2d* ve *Gramschmidt*'te terstir. Örneğin, *fdtd2d_ step2_ kernel* en yüksek SDC oranına sahipken, yürütme süresi diğer kernel fonksiyonu arasında en kısa olanıdır. Şekil 16'da gösterilen tüm *Fdtd2d* çoklamalı seçenekleri arasında, *fdtd2d_ step2_ kernel* çoklanmış seçenek en düşük yürütme süresine sahiptir. Kernel fonksiyonu en yüksek SDC oranına sahip olduğundan, nispeten daha az yürütme süresinden fedakarlık ederek makul miktarda SDC oranında azalma elde ederiz. Öte yandan, *fdtd2d_ step3_ kernel*'in SDC oranı *fdtd2d_ step2_ kernel*'e yakındır ve yürütme süreleri arasındaki fark büyük değildir. Dolayısıyla ikisini de çoklamamızın faydalı olacağını söyleyebiliriz. *Gramschmidt* benzer özelliklere sahiptir. Örneğin, *gramschmidt_ kernel2* en yüksek SDC



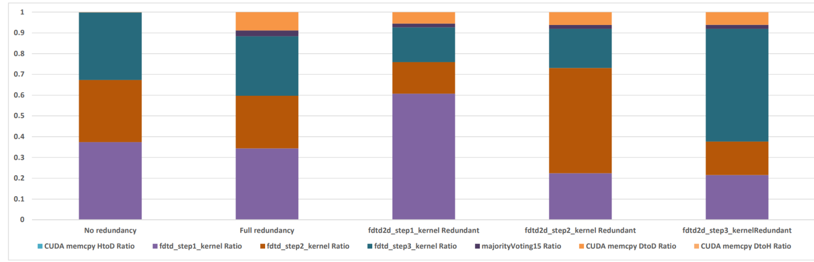
Şekil 17: Çoklanmış yürütmelerin SDC oranlerı ve yürütme sürelerindeki yüzdelik değişim.

oranına sahiptir, ancak uygulamanın diğer kernel fonksiyonları arasında en düşük yürütme süresine sahiptir. Bu nedenle, yalnızca bu fonksiyonu veya iki fonksiyonu, yani *gramschmidt_kernel1* ve *gramschmidt_kernel2*'yi çoklamak, hem performans hem de güvenilirlik açısından faydalı olabilir.

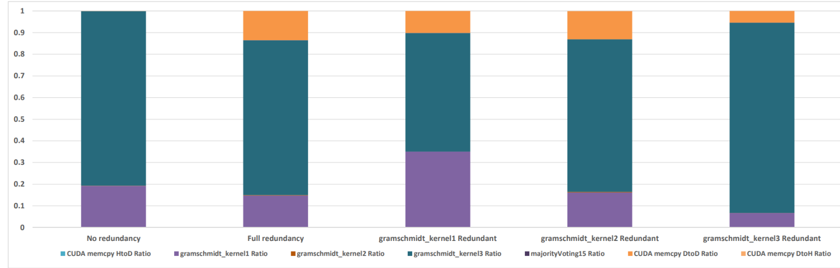
Çoklamalı yürütme şemalarının performansını daha önce tartışırken, performans ve güvenilirlik kazanımlarını da birlikte analiz etmek istiyoruz. Farklı çoklama şemaları arasındaki performans-güvenilirlik dengesini değerlendirmek için, hem SDC oranlarındaki değişikliği (yüzde olarak) hata hassasiyeti ölçüsü olarak hem de yürütme sürelerini Şekil 17'de gösteriyoruz. Üçlü çoklamaya dayalı yedeklilik yöntemimizin tam koruma sağladığını ve çoklamalı olarak çalıştırılan kodumuzun hataları maskeleyerek hatalı durumlara neden olmayacağını varsayıyoruz. SDC oranlarını buna göre hesaplıyoruz. Yalnızca en savunmasız kernel fonksiyonlarının çoklanması ve en hataya hassas iki kernel fonksiyonunun çoklanması dahil olmak üzere her uygulama için iki çoklanmalı yürütme senaryosu kullanıyoruz. Bicg için, esasen, iki kernel fonksiyonundan oluştuğu için Tam çoklama ve en savunmasız fonksiyonun çoklandığı şemalarını sunuyoruz. İki kernel fonksiyonunu çokladığımız, Tam çoklama gerçekleştirirsek, SDC oranı sıfıra düşer (%100 azalma). Ancak, *bicg_kernel2* replikasyonundan elde edilen güvenilirlik kazancı sınırlıdır (yaklaşık %65). Bu nedenle SDC'lere karşı herhangi bir süre sınırlamamız ve/veya toleransımız yoksa tam çoklama uygulamanın daha faydalı olacağını söyleyebiliriz. Aksi takdirde, yalnızca *bicg_kernel2* çoklama kullanılması faydalı olacaktır. Hem *Correlation* hem de *Covariance* için, daha önce tartışıldığı gibi, iki fonksiyonlu çoklama durumu için hata hassasiyeti kazancı büyürken yürütme süresi farklı değildir. Şekil 17'de, yalnızca *fdtd2d_step2_kernel* çoklanmış seçenek ile *fdtd2d_step2_kernel* ve *fdtd2d_step3_kernel* seçenek-



(a) Bicg



(b) Fdtd2d



(c) Gramschmidt

Şekil 18: Her fonksiyonun yürütülme profili.

leri arasında büyük bir yüzde farkı olmasına rağmen, kernel fonksiyonlarının yürütme süreleri kısa olduğu için mutlak fark önemli değildir. Ancak, *Fdtd2d*'nin çok büyük miktarda veriyle yürütüldüğü ve yürütmenin daha uzun süreler gerektirdiği durumlarda, yalnızca en savunmasız fonksiyonun çoklanmasından elde edilen performans kazancı önemli hale gelir. *Gramschmidt* için, hata hassasiyeti ile alternatif çoklama şemalarının performansı arasındaki dengeyi açıkça görebiliriz. Yalnızca en savunmasız fonksiyonun, yani *gramschmidt_kernel2* çoklanması, hemen hemen aynı (yaklaşık %30) performans ve hata hassasiyeti kazanımlarını sağlar. Bu nedenle, yürütme süresi ve güvenilirlik dahil olmak üzere sistemin gereksinimlerini göz önünde bulundurmak ve buna göre yedeklilik düzeyi hakkında bir karar vermek gerekir. Şekil 18, artıklık senaryoları için her bir fonksiyonun yürütme süresi profilini göstermektedir. Spesifik olarak, kernel fonksiyonun yürütmeleri, çoğunluk oylaması, girdinin CPU'dan GPU'ya (CUDA memcpy HtoD) kopyalanması, çıktının GPU'dan CPU'ya (CUDA memcpy DtoH) kopyalanması ve GPU'dan GPU'ya (CUDA memcpy DtoD) çoklanmış çıktı kopyalama işlemleri dahil olmak üzere hafıza kopyalama işlemlerinin profilini çıkarıyoruz. Zamanın büyük bir kısmının kernel fonksiyon yürütmeleri sırasında harcandığını görebiliriz. Çoklanmış çıktı kopyalama işlemleri de *Fdtd2d* (Şekil 18b) ve *Gramschmidt*'te (Şekil 18c) önemli ölçüde zaman alırken, bu işlemlerin yüzdesi, diğer programlar için küçüktür. Küçük yüzdelerin değerleri *Bicg* için daha ayrıntılı bir görünümde gösterilmektedir (Şekil 18a).

Çoklama senaryolarımıza ek hafıza işlemleri ve çoğunluk oylaması fonksiyonu ile çoklanmış kernel fonksiyonlarını dahil etmemize rağmen, yürütme süresindeki artışın ana nedeni, çoklanan fonksiyon yürütmeleridir. Çoğunluk oylama fonksiyonu ve hafıza işlemleri, kernel yürütmelerine kıyasla önemli ölçüde zaman almamaktadır.

5 SONUÇ VE TARTIŞMA

FTGPGPU projesinde, GPGPU uygulamaları için bölgesel geçici hata hassasiyeti ve hata yayılımı analizi gerçekleştirilmiştir. GPU mimarilerinde çalışan programların hem farklı kernel fonksiyonlarında, hem de aynı kernel fonksiyonundaki farklı kod bölgelerindeki hata hassasiyetlerini değerlendirmek için bir hata enjeksiyon aracı tasarlanmış ve geliştirilmiştir.

GPGPU benchmark setlerindeki programlar için ayrıntılı hata enjeksiyonu deneyleri yapılarak, GPGPU programlarının farklı kod bölgeleri için farklı geçici hata hassasiyetleri ve hata yayılımları gösterdikleri gözlemlenmiştir. Uygulamalardaki kod bölgeleri, diğer kod bölgelerini (aynı kernel fonksiyonunda veya diğer kernel fonksiyonlarında) veya veri yapılarını, ve ayrıca GPGPU uygulamasının akışına ve veri kullanımına bağlı olarak nihai çıktıyı farklı şekillerde etkilemektedir. GPGPU programlarındaki paralellik derecesi, yapılan işe veya CUDA iş parçacıkları tarafından kullanılan verilere bağlı olarak geçici hata hassasiyetini farklı şekillerde etkilemektedir.

Projemiz kapsamında geliştirdiğimi hata hassasiyeti analizi aracı, güvenilirliği hedefleyen hem profesyonel hem de akademik sistemlere yardımcı olacaktır. Güvenilirlik için etki alanına özgü metrikleri dikkate alarak ve çeşitli düzeylerde (kernel fonksiyonları arası ve kernel fonksiyonu içi gibi) hata enjeksiyonları gerçekleştirerek, yürütmelerinde hem performans hem de güvenilirlik gerektiren güvenilirlik açısından kritik sistemlerde daha verimli hata toleransı teknikleri kullanılabilecektir. Seçimli hata toleransı yöntemimizde bu çalışmaların bir örneği gerçekleştirilmiştir. Hedef programların yalnızca en savunmasız kısımlarının seçici olarak çoklanması ile performans ve güvenilirlik değiş tokuş analizi yapılmıştır.

GPGPU programlarının sahip oldukları mimari özellikler, donanımı nasıl kullanarak performans gösterdikleri gibi karakteristikleri ile bu uygulamaların hata hassasiyetleri arasında ilişki kurmak, uzun süreli ve maliyetli hata enjeksiyonu deneyleri çalıştırmadan programın özellik metriklerini ölçerek hata hassasiyetlerini belirlemeyi mümkün kılmaktadır. Gerçekleştirdiğimiz tahminleme yapısıyla sessiz veri bozunumu, çökme ve doğru çalışma senaryolarının tahminlenmesi mümkün olmuştur. Hata hassasiyetini belirlemek istenilen kodun, tüm bir GPGPU programı yerine daha küçük kod parçacıklarının olması hedef programların bölgesel hata hassasiyetlerinin tahminlenmesini mümkün kılmaktadır. Geliştirdiğimiz sınıflandırma ve regresyon tabanlı tahminleme modellerimiz, hata hassasiyeti değerlerinin yüksek başarı değerleriyle tespit edilmesini mümkün kılmaktadır.

Bölgesel hata hassasiyeti analizinin gerçeklediğimiz kullanım senaryolarının yanı sıra farklı şekillerde ve yöntemlerle uygulanması mümkündür. Projemizde geliştirdiğimiz ve programların belli bölgelerinde ortaya çıkan hataların program çıktısına etkisini gösteren aracımız, belirli program böl-

gelerindeki hataların program çıktılarını çok fazla etkilemediğini ortaya koymaktadır. Aracımızın belirlediği bu özelliği göz önünde bulundurarak programların hangi bölgelerinde yaklaşık hesaplama uygulanabileceğinin tespit edilmesi mümkündür. Ayrıca program geliştiricilerin hata hassasiyeti yüksek kod parçalarının özelliklerini değerlendirerek mümkün olduğu durumlarda olası hassasiyetleri azaltacak şekilde kodlarını değiştirmeleri söz konusu olacaktır. Yazılım geliştiricilerin hata hassasiyeti konusunda yönlendirilmeleri mümkün olacaktır.

TÜBİTAK
PROJE ÖZET BİLGİ FORMU

Proje Yürütücüsü:	Dr. Öğr. Üyesi IŞIL ÖZ
Proje No:	119E011
Proje Başlığı:	FTGPGPU - Genel Amaçlı Grafik İşlemci Birimi Uygulamaları İçin Donanım Hatası Toleransı Analizi
Proje Türü:	3501 - Kariyer
Proje Süresi:	24
Araştırmacılar:	
Danışmanlar:	
Projenin Yürütüldüğü Kuruluş ve Adresi:	İZMİR YÜKSEK TEKNOLOJİ ENSTİTÜSÜ
Projenin Başlangıç ve Bitiş Tarihleri:	15/08/2019 - 15/02/2022
Onaylanan Bütçe:	327004.68
Harcanan Bütçe:	221245.58
Öz:	Genel amaçlı hesaplamalar için grafik işlemci birimlerinin (GPGPU) kullanımı, donanım hatalarının kritikliğini arttırmakta, programların geçici hata hassasiyetini değerlendirmek ve uygun hata toleransı tekniklerini kullanmak daha önemli hale gelmektedir. Hataya en hassas program bölgelerinin korunması yoluyla, hem performansı, hem de güvenilirliği hedefleyen sistemler için ayrıntılı bölgesel hata hassasiyeti analizi çok önemlidir. Bu projede, GPGPU uygulamalarının geçici donanım hatası hassasiyetinin ölçülmesi, analiz edilmesi ve bu analizlerin sonuçlarının program özellikleri ile ilişkilendirilmesi, seçimli hata toleransı yöntemi geliştirilmesi yoluyla kullanılması amaçlanmıştır. Projenin ilk katkısı, GPGPU uygulamlarının geçici hata hassasiyetlerinin bölgesel olarak belirlenmesi için yazılım ile donanım ilişkisini sağlayacak şekilde assembly seviyesinde hata ayıklayıcı tabanlı bir hata enjeksiyonu ve hata yayılımı analizi aracı geliştirilmesidir. Bu araç kullanılarak farklı yapıdaki, farklı özelliklere sahip GPGPU programlarının belirlenen kod bölgelerine hata enjeksiyonu sağlayan deneyler yapılmış, kod bölgelerinin hata hassasiyetleri ve oluşan hatanın program süresince farklı veri yapılarına yayılımı incelenmiştir. Projenin ikinci katkısı, GPGPU program kod parçalarının özellikleri ile bu kodlar çalışırken meydana gelebilecek hatalara hassasiyetleri arasındaki ilişkinin incelenmesidir. GPGPU programlarındaki kod parçacıklarının performans ve mimari özellikleri profilleme ve simulasyon yöntemleriyle elde edilmiş, ilk adımda geliştirilen hata enjeksiyonu aracıyla belirlenen kod parçalarına hata enjekte ederek uygulanan deney sonuçlarında sessiz veri bozunumu, çökme ve doğru çalışma durumları belirlenmiştir. Program özellikleri-hata hassasiyeti ikilisi arasındaki ilişki incelenerek program özellikleri verilen bir GPGPU uygulamasının hata hassasiyet değerleri makine öğrenmesi yöntemleriyle tahmin edilmiştir. Geliştirilen tahminleme modelleriyle sessiz veri bozunumu için %82, çökme durumları için %87, doğru çalışma durumları için %96 doğruluk oranlarıyla tahminleme başarısı sağlanmıştır. Projenin üçüncü katkısı, hataya daha hassas kod bölgelerinin çoklanmasına dayalı seçimli hata toleransı yöntemi geliştirilmesidir. Program geliştirici veya kullanıcı tarafından kaynak kodda işaretlenen kod bölgelerinin çoklanması şeklinde gerçeklenen derleyici seviyesinde geliştirilen hata toleransı yapısı, belirtilen kernel fonksiyonlarının çoklanmasını artıklı kernel fonksiyonu olarak veya tek kernel fonksiyonu altında artıklı iş parçacığı olarak veya CUDA stream tekniği ile mümkün kılmaktadır. Böylece uygulamanın paralellik ve veri kullanımı özelliklerine göre farklı çoklama yürütme durumları seçilebilmekte, kaba taneli (coarse-grained) bir yapıda çıktı kontrolü ile performanslı bir şekilde çoklama sağlanmaktadır.

TÜBİTAK

Abstract:	As the use of graphics processing units for general-purpose calculations (GPGPU) increases the criticality of the hardware errors, it becomes more important to evaluate the transient error vulnerability of the programs and to perform appropriate fault tolerance techniques. Detailed regional soft error vulnerability analysis is essential for systems targeting both performance and reliability, by protecting the most vulnerable program regions. In this project, we aim to measure and analyze soft error vulnerability of GPGPU programs, and based on the analysis results, we correlate error characteristics with program features and develop a selective fault tolerance method. The first contribution of the project is the development of an assembly-level debugger-based fault injection and error propagation analysis tool that enables regional soft error vulnerability analysis by associating software code regions and hardware components. By utilizing the tool, we carry out fault injection experiments by targeting the determined code regions of GPGPU programs with different structures and different features. We evaluate soft error vulnerability of the target code regions and error propagation through the data structures during the faulty program execution. The second contribution of the project is the analysis of the relationship between the GPGPU program features and their vulnerability to soft errors. We obtain the performance and architectural features of the code snippets in GPGPU programs, and perform fault injection experiments by utilizing our fault injection tool to collect silent data corruption, crash and correct execution rates. By examining the relationship between program features and error vulnerability rates, we predict the error vulnerability values of a GPU application by machine learning methods. Our prediction models achieve prediction accuracy rates of 96.6%, 82.6%, and 87% for masked fault rates, SDCs, and crashes, respectively. The third contribution of the project is the development of a selective fault tolerance method based on the redundancy of more vulnerable code regions. Our compiler-level fault tolerance framework performs redundant multithreading for the code regions marked in the source code by the program developer or the user, and enables the redundant execution of the specified kernel functions as a redundant kernel function or as a redundant thread under a single kernel function or with the CUDA stream technique. Thus, the target execution can be configured with different redundant execution schemes according to the parallelism and data usage characteristics of the application, and the redundancy is maintained in a high-performance manner with coarse-grained output control.
Anahtar Kelimeler:	Geçici donanım hatası güvenilirliği, GPU mimarileri, GPGPU uygulamaları
Fikri Ürün Bildirim Formu Sunuldu Mu?:	Evet
Projeden Yapılan Yayınlar:	1- Regional soft error vulnerability and error propagation analysis for GPGPU applications (Makale - null), 2- Analyzing the Effect of Performance Optimizations on Soft Error Reliability for Machine Learning Applications: A Case Study (Bildiri - Sözlü Sunum), 3- GPGPU Uygulamaları için Bölgesel HataHassasiyet Analizi (Bildiri - Sözlü Sunum), 4- GPGPU Uygulamaları için Bölgesel HataHassasiyet Analizi (Bildiri - Sözlü Sunum), 5- GPGPU Uygulamaları için Bölgesel HataHassasiyet Analizi (Bildiri - Sözlü Sunum),

TÜBİTAK