T.C.
KIRIKKALE ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
YÜKSEK LİSANS TEZİ

CUDA İLE MR GÖRÜNTÜLERİNİN GELİŞTİRİLEN DOSYA YAPISI İLE SIKIŞTIRILARAK SAKLANMASI

Abdüssamed ERCİYAS

NİSAN 2018

Bilgisayar Mühendisliği Anabilim Dalında Abdüssamed ERCİYAS tarafından hazırlanan CUDA İLE MR GÖRÜNTÜLERİNİN GELİŞTİRİLEN DOSYA YAPISI İLE SIKIŞTIRILARAK SAKLANMASI adlı Yüksek Lisans Tezinin Anabilim Dalı standartlarına uygun olduğunu onaylarım.

Prof. Dr. Hasan ERBAY
Anabilim Dalı Başkanı

Bu tezi okuduğumu ve tezin Yüksek Lisans Tezi olarak bütün gereklilikleri yerine getirdiğini onaylarım.

Dr. Öğr. Üyesi Atilla ERGÜZEN
Danışman

Jüri Üyeleri

Başkan : Doç. Dr. Necaattin BARIŞÇI ____________________

Üye (Danışman) : Dr. Öğr. Üyesi Atilla ERGÜZEN____________________

Üye : Dr. Öğr. Üyesi Erdal ERDAL ____________________

19/04/2018

Bu tez ile Kırıkkale Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu Yüksek Lisans derecesini onaylamıştır.

Prof. Dr. Mustafa YİĞİTOĞLU
Fen Bilimleri Enstitüsü Müdürü

ÖZET

CUDA İLE MR GÖRÜNTÜLERİNİN GELİŞTİRİLEN DOSYA YAPISI İLE SIKIŞTIRILARAK SAKLANMASI

ERCİYAS, Abdüssamed
Kırıkkale Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı, Yüksek Lisans Tezi
Danışman: Dr. Öğr. Üyesi Atilla ERGÜZEN
NİSAN 2018, 74 sayfa

Medikal MR görüntülerinin hastanelerde çokça kullanılmasından dolayı bu görüntülerin saklanması için büyük depolama alanlarına ihtiyaç duyulmaktadır. Ayrıca bu görüntüler farklı zaman dilimlerinde teşhis amaçlı sık görüntülenmektedir. Bu nedenle büyük bir bant genişliğine ihtiyaç duyulmaktadır. Bu problemi çözmek için medikal görüntüleri sistemi aksatmadan hızlı bir şekilde sıkıştırıp saklamak gerekecektir. Medikal MR görüntüleri üzerinde yapılan incelemelerde görüntü içinde kullanılmayan bölgelerin (NON-ROI) geniş yer kapladığı ve görüntü içerisindeki bu gereksiz alan temizlendiğinde görüntü boyutunun önemli oranda düşürülebileceği görülmüştür. CUDA ile geliştirilen bu yöntemde: Medikal MR görüntüleri içerisindeki ROI (Region of Interest) bölgesi bir 3x3 lük Kirsch filtre matrisi ile CUDA çekirdeklerine gönderilerek tespit edilir, NON-ROI bölgesi görüntüden çıkarılır. Elde edilen görüntü dikdörtgen yapıya sahiptir. Ancak ROI bölgesi bu görüntü içerisinde NON-ROI bölgesi pikselleri bulunmaktadır. Bunun için yeni görüntü saklama dosya yapısı geliştirilerek başarılı bir şekilde uygulanmıştır.

Bu işlemler sırasıyla önce CPU üzerinde seri uygulama ile, sonra da GPU üzerindeki paralel uygulama ile çalıştırılır. İşlemler sonucunda GPU üzerinde çalıştırılan uygulamanın, CPU üzerindeki uygulamadan 68 kat daha hızlı sonuç ürettiği görülmüştür. Ayrıca geliştirilen dosya yapısı ile orijinal görüntü boyutundan %92

oranında; orijinal görüntünün sıkıştırılmış boyutundan ise %40 oranında boyut kazancı sağlanmış ve oldukça başarılı bir sıkıştırma oranı yakalanmıştır.

Anahtar Kelimeler: CUDA, Görüntü Sıkıştırma, GPU Programlama, Paralel Programlama, Medikal Görüntü İşleme

ABSTRACT

COMPRESSED STORAGE OF MRI BY CUDA WITH THE DEVELOPED FILE STRUCTURE

ERCİYAS, Abdüssamed
Kırıkkale University
Graduate School of Natural and Applied Sciences
Department of Computer Engineering, M. Sc. Thesis
Supervisor: Asst. Prof. Atilla ERGÜZEN
APRIL 2018, 74 pages

Due to the large use of medical MR images in hospitals, large storage areas are needed to store these images. In addition, these images are often displayed for diagnostic purposes at different times. For this reason, a large bandwidth is needed. To solve this problem, medical images will need to be compressed and stored quickly without disruption. It has been seen that in the studies made on medical MR images, the non-used regions (NON-ROI) occupy a large space and the image size can be reduced significantly when the unnecessary area in the image is cleaned. In this method developed with CUDA: Region of Interest (ROI) in the medical MR images is detected by sending a 3x3 Kirsch filter matrix to the CUDA cores, and the NON-ROI region is extracted from the image. The resulting image has a rectangular structure. However, the ROI region has NON-ROI region pixels in this image. For this a new image storage file structure has been developed and implemented successfully.

These operations are executed first by serial application on CPU and then by parallel application on GPU. As a result, it was seen that the application running on the GPU produced 68 times faster results than the application on the CPU. In addition, with the new compressed file structure, 40% of the original size of the compressed size of the original image is saved and a very successful compression ratio is achieved.

Keywords: CUDA, Image Compression, GPU Programming, Parallel Programming, Medical Image Processing

TEŞEKKÜR

Tezimin hazırlanmasında her zaman destek olan, yardımlarını esirgemeyen danışmanın Sayın Dr. Öğr. Üyesi Atilla ERGÜZEN'e, görüntü işleme konusunda bilgilerini ve deneyimlerini paylaşan Dr. Öğr. Üyesi Erdal ERDAL hocama teşekkürlerimi sunarım.

Çalışmalarımda her zaman yanımda olan sevgili eşim Sema ERCİYAS'a içten teşekkürlerimi sunarım.

İÇİNDEKİLER DİZİNİ

Sayfa

ÖZET i
ABSTRACT iii
TEŞEKKÜR v
İÇİNDEKİLER DİZİNİ vi
ŞEKİLLER DİZİNİ ix
ÇİZELGELER DİZİNİ xi
KISALTMALAR DİZİNİ xii
1. GİRİŞ 1
2. MATERYAL VE YÖNTEM 7
2.1. CUDA 7
2.2. OpenCL 8
2.3. CUDA ile OpenCL Kullanımları ve Karşılaştırılması 8
2.4. OpenGL 10
2.5. DirectX 11
2.6. OpenGL ile DirectX Karşılaştırması 12
2.7. GPU ile CPU Kıyaslaması 12
2.8. CUDA GPU Mimarileri 14
2.8.1. Tesla Mimarisi 15
2.8.2. Fermi Mimarisi 17
2.8.3. Kepler GPU Mimarisi 18
2.8.4. Maxwell Mimarisi 19
2.9. CUDA Programlama Mimarisi 20
2.9.1. Çekirdek (Kernel) Nedir? 20
2.9.2. Izgara (Grid) Nedir? 20
2.9.3. Blok (Block) Nedir? 21
2.9.4. İş Parçacığı (Thread) Nedir? 21
2.10. CUDA Bellek Mimarisi 23
2.10.1. Yerel Bellek (Local Memory) 23

2.10.2. Paylaşımlı Bellek (Shared Memory) ... 23
2.10.3. Genel Bellek (Global Memory) ... 23
2.10.4. Sabit Bellek (Constant Memory) ... 24
2.10.5. Doku Bellek (Texture Memory) ... 24
2.11. CUDA Erişim Fonksiyonları ... 24
2.11.1.global ... 24
2.11.2. device ... 25
2.11.3. host ... 25
2.12. CUDA Bellek Operasyonları ... 26
2.12.1. Hafıza Ayırma Operasyonları ... 26
2.12.2. Veri Kopyalama İşlemi ... 26
2.13. CUDA Çalışma Prensibi ... 26
2.14. CUDA Programlama API'leri ... 32
2.14.1. CUBLAS ... 32
2.14.2. CUFFT ... 33
2.15. CUDA Programlama Nasıl Yapılır? ... 33
2.15.1.ManagedCuda ... 35
2.15.2.CudaFy ... 36
2.16. Medikal Görüntü İşleme ... 38
2.16.1. MR Görüntüsü ... 39
2.17. Medikal Görüntülerin Arşivlenmesi ... 40
2.18. DICOM Standardı ... 40
2.19. DICOM Dosya Yapısı ... 41
2.20. Görüntü Bölütleme ... 42
2.21. Görüntü Üzerinde Kenar Belirleme ... 43
2.21.1. Kenar Belirleme Yöntemleri ... 43
2.21.1.1.Türev Almaya Dayalı Kenar Belirleme Yöntemleri ... 43
2.21.1.1.1. 1.Türeve Dayalı Kenar Belirleme – Gradiyent Yöntemi ... 44
2.21.1.1.1.1. Sobel Kenar Dedektörü ... 46
2.21.1.1.1.2. Prewitt Kenar Dedektörü ... 47
2.21.1.1.1.3. Kirsch Kenar Dedektörü ... 48
2.21.1.2. Aktif Kontur Yöntemi ... 49
2.21.1.3. Eşikleme Yöntemi ... 49

2.21.1.3.1. Global Eşikleme 50
2.21.1.3.2. Adaptif Eşikleme 50
2.21.1.3.3. Çoklu Eşikleme 51
3. ARAŞTIRMA BULGULARI 52
3.1. ROI Bölgesinin Tespit Edilmesi 53
3.2. En Uygun Filtre Matrisinin Seçilmesi 54
3.3. ROI Bölgesinin Görüntüden Çıkarılması 55
3.4. ROI Bölgesini Önerilen Dosya Yapısı İle Saklama 56
3.5. Performans Testleri 57
4. TARTIŞMA VE SONUÇ 62
KAYNAKÇA 64
EKLER 69

ŞEKİLLER DİZİNİ

ŞEKİL Sayfa

1.1. Yıllara göre CPU-GPU FPLOPS karşılaştırması [2] 2
1.2. Yıllara göre CPU-GPU bellek bant genişliği karşılaştırılması [2] 2
2.1. CUDA GPU Mimarisi [16] 14
2.2. Fermi ve Kepler mimarileri karşılaştırması [16] 20
2.3. Izgara, Blok ve İş Parçacığı yapısı [21] 22
2.4. Izgara örneği [21] 22
2.5. C dili ile yazılmış Ardışık/Sequental Kod ve CUDA paralel kod [14] 25
2.6. CUDA GPU Belleğinde Yer Tahsis Edilme İşlemi 27
2.7. Bellek tahsisinden sonra verilerin GPU belleğine kopyalanması 28
2.8. Çekirdeği oluşturacak ızgara yapısının tanımlanması 28
2.9. Çekirdek fonksiyonunun çağırılması 29
2.10. vectorAddKernel fonksiyonu 30
2.11. GPU da hesaplanmış verinin ana belleğe kopyalanması 31
2.12. Host ve GPU belleğinin temizlenmesi ve GPU'nun resetlenmesi 31
2.13. Visual Studio üzerinde yeni bir CUDA Runtime projesi açmak 34
2.14. NVIDIA Nsight HUD Launcher ile Debug Mod ayarları yapmak 34
2.15. CUDA kod örnekleri 35
2.16. Beyin MR görüntüsü 39
2.17. DICOM dosya yapısı 42
2.18. Türev operatörleri ile kenar belirleme: (a) Koyu arka plan üzerindeki beyaz bant görüntüsü; (b) Açık arka plan üzerindeki siyah bant görüntüsü. Bu görüntülere ilişkin 1-B çizimler ve bu çizimlerin 1. ve 2.türevleri [32] 45
2.19. Kenar belirleme yöntemi blok şeması [32] 45
2.20. Aktif Kontur Metodu iterasyonlar sonucu ROI tespit edilme aşamaları..........49
3.1. Beyin MR Görüntüleri 52
3.2. Beyin MR görüntülerine farklı filtrelerin uygulanması (a)orijinal görüntü (b)Sobel Filtre Matrisi (c)Prewitt Filtre Matrisi (d)Kirsch Filtre Matrisi uygulandıktan sonraki durumlar 54

3.3. (a)Orijinal görüntü (b)ROI bölgesi tespit edilmiş ve çerçevesi daraltılmış görüntü 55
3.4. ROI Bölgesini Dizi içerisine geliştirilen dosya yapısı ile saklama (a)7x7 lik görüntü örneği (b) Dosya yapısı oluşturma işlemi 56

ÇİZELGELER DİZİNİ

ÇİZELGE Sayfa

2.1. CPU ve GPU geçmişten günümüze minimum ve maksimum değerleri 13
2.2. Maskelenecek görüntü pikselleri.. 46
2.3. Sobe1 kenar dedektörü (a)yatay filtre(b)dikey filtre.. 46
2.4. Prewitt kenar dedektörü (a)yatay filtre (b)dikey filtre 47
2.5. Kirsch kenar dedektörü (a)yatay filtre (b)dikey filtre 48
3.1. MR görüntülerinin orijinal, ROI ve geliştirilen dosya yapısı boyutları……… 59
3.2. Orijinal, ROI ve geliştirilen dosya yapısındaki verilerin sıkıştırılmış boyutları . 60
3.3. CPU üzerinde çalışan seri kod ile CUDA ile kodlanmış GPU kodunun çalışma zamanı karşılaştırması 61

KISALTMALAR DİZİNİ

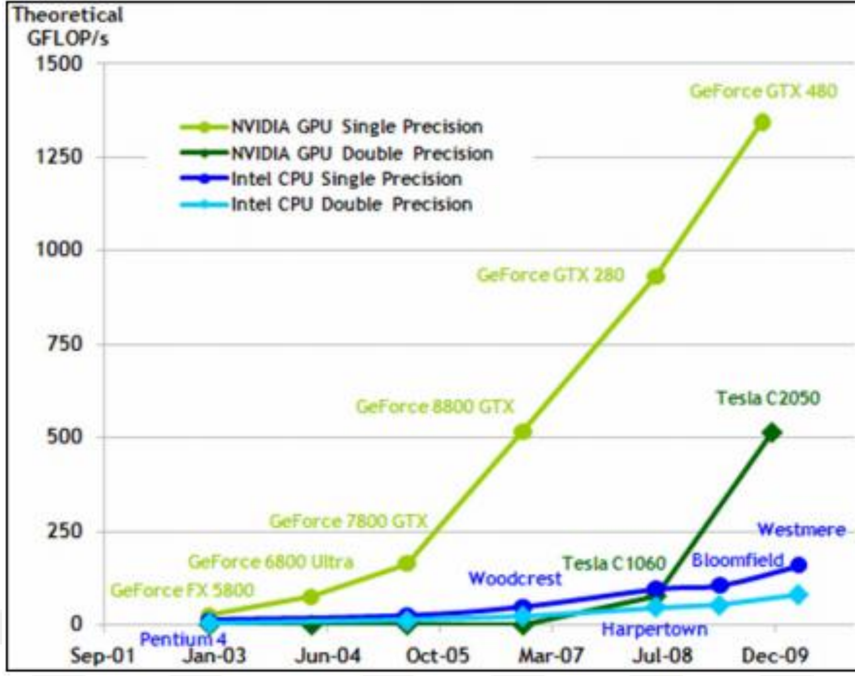
B	Boyutlu
CUFFT	Compute Unified Fast Fourier Transform
CUBLAS	Compute Unified Basic Linear Algebra Subprograms
D	Dimension
DRAM	Dynamic Random Access Memory
FPS	Frame Per Second
GB	Gigabyte
L1	Level-1
L2	Level-2
JPEG	Joint Photographics Experts Group
KB	Kilobyte
MS	Milisaniye
NVCC	NVIDIA C Compiler
PNG	Portable Network Graphics
API	Application Programmig Interface

1. GİRİŞ

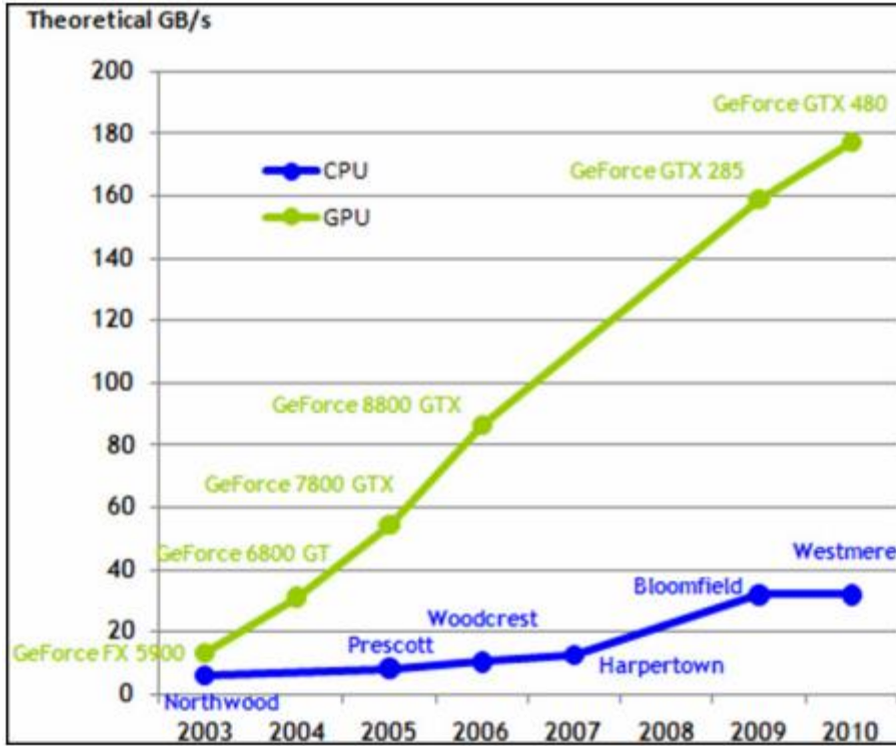
Görüntü işleme günlük yaşamda birçok alanda kullanılmakta ve insan hayatını kolaylaştırmaktadır. Görüntü işleme ile görüntü üzerindeki en ufak bir ayrıntıyı seçebilme, görüntü ile nesneleri eşleştirebilme, nesneleri sayma, hatalı üretimleri tespit edebilme gibi insanın bir anda yapamayacağı işlemleri çok hızlı bir şekilde yapabilmekteyiz.

Görüntü üzerinde çalışabilmek ve hızlı hesaplama yapabilmek için CPU genel olarak yetersiz kalmaktadır. Bu noktada yapısını görüntü işleme üzerine gerçekleştirmiş olan GPU teknolojisi ortaya çıkmıştır.

1980'li yıllarda IBM ve Intel hakimiyetinde olan GPU gerçekleştirme çalışmaları, 1990'lı yıllarda sadece grafik kartı geliştirmeye yönelmiş olan S3 Graphics, NVIDIA ve ATI gibi firmaların kontrolünde ilerlemiştir. Bu dönemde 2B donanımsal hızlandırma yaygınlaşmıştır. 1992 yılında OpenGL ve 1995 yılında DirectX (Direct3D) grafik programlama kütüphanesinin de kullanıma sunulmasıyla GPU dünyasında yeni bir dönem başlamıştır. 2006 yılında NVIDIA, CUDA GPU programlama platformunu kullanıma sunmuştur. 2008 yılında da Apple, OpenCL GPU programlama platformu sunmuştur. CUDA ile sadece NVIDIA grafik kartlarda geliştirme yapılabilirken OpenCL ile tüm GPU sistemleri ile geliştirmeye olanak sağlanmıştır. Bu gelişmelerle 2000'li yıllardan sonra, GPU hesaplama gücünü her yıl artırarak geliştirmiştir [1]. Şekil 1.1'de yıllara göre GFLOPS (Giga Floating Point Operations Per Second/Saniyedeki Giga Kayan Nokta İşlemleri) bazında, Şekil 1.2'de yıllara göre bellek bant genişliği bazında CPU-GPU kıyaslaması gösterilmiştir.



Şekil 1.1. Yıllara göre CPU-GPU GFLOP/s karşılaştırması [2]



Şekil 1.2. Yıllara göre CPU-GPU bellek bant genişliği karşılaştırılması [2]

GPU’ların artan hesaplama kabiliyetleri GPU çalışmalarının görüntü işleme, lineer cebir, istatistik ve 3B modelleme gibi alanlarda uygulanmasının tetikleyicisi olmuştur [1].

Victor Podlozhnyuk, CUDA ile basit konvolüsyon ve FFT (Fast Fourier Transform/Hızlı Fourier Dönüşümü) temelli konvolüsyon işlemlerini CUFFT kütüphanesinden de faydalanarak uygulamış ve analizlerini sunmuştur. Buna göre paylaşımlı bellek kullanılarak geliştirilen uygulama doku bellek kullanılarak geliştirilen uygulamaya göre 2 kat daha hızlı sonuç vermiştir [3, 4].

Michael Garland ve diğerleri yaptıkları çalışmada CUDA kullanımı üzerine değerlendirmelerini paylaşmışlardır. Medikal görüntü işleme çalışmalarında CUDA ile ulaşılan performans değerlerini, aynı çalışmaların CPU uygulamasıyla kıyaslamışlardır. GPU üzerinde geliştirilen uygulamanın CPU üzerinde geliştirilen uygulamadan 16 kat daha hızlı olduğunu tespit etmişlerdir [5].

Zhiyi Yang ve diğerleri tarafından yapılan çalışmada, yaygın şekilde kullanılan kenar bulma, histogram alma gibi görüntü işleme adımlarının GPU uygulaması yapılmış ve CPU uygulamasıyla çalışma zamanı bakımından kıyaslanmıştır. GPU belleğine ana bellekten verinin aktarılması ve verinin tekrar ana belleğe alınması süreleri hesaba katılmadan CPU uygulamasına göre 40 kat performans artışı sağlanmıştır [6].

Jing Huang ve diğerleri tarafından Vektör Uyumlu Eşleme (VCM) algoritması, CPU ve GPU üzerinde ayrı ayrı uygulanarak performansları karşılaştırılmıştır. VCM algoritmasının GPU üzerindeki uygulaması en yeni teknolojiye sahip CPU’daki uygulamaya göre 40 kat daha hızlı çalışmıştır [7].

Raúl Cabido ve diğerleri tarafından yapılan çalışmada, GPU üzerinde parçacık süzme algoritması kullanarak nesne takibi uygulaması yapılmıştır. Daha önceki çalışmalarda 20 fps civarında sonuçlar alınırken, CUDA teknolojisinin kullanıldığı bu çalışmada, 6 durum değişkeni ve 256 parçacıklı bir parçacık süzgeci uygulamasında 320x240 çözünürlüklü video için 700 fps değerine ulaşılmıştır [8].

Ergüzen A. ve Erdal E. medikal görüntülerde ROI (İlgi Alanı) ve OCR (Otomatik Karakter Tanıma) temelli entegre bir arşivleme yöntemi geliştirmişlerdir. Bu yöntemde görüntü ROI ve NON-ROI (ROI Olmayan) bölgelerine ayrılmıştır. ROI bölgesi kayıpsız sıkıştırma algoritması JPEG-LS ile sıkıştırılmış, NON-ROI bölgesinde ise OCR ve Huffman algoritmaları kullanılmıştır. Bu yöntem ile görüntünün NON-ROI bölgesi için %92.12 ile %97.84 arasında bir sıkıştırma oranı elde edilmiştir. Ayrıca geliştirilen bu yöntemde görüntünün NON-ROI kısmı için literatürdeki en iyi yaklaşıma göre %83.30 oranında bir başarı sağlanmıştır [9].

Lei Pan ve diğerleri CUDA kullanarak medikal görüntüler üzerinde Bölge Büyütme ve Yılan algoritmaları uygulayarak CUDA'nın avantajlarını ve nasıl tasarlandığını açıklamışlardır. Bu çalışmada CUDA ile GPU uygulamaları geliştirirken 2 önemli faktöre dikkat edilmesi gerektiğini vurgulamışlardır. Bu faktörlerden ilki görüntü pikselleri ile doğru orantılı iş parçacığı yapısı oluşturulması gerektiği, ikincisi ise ızgara içerisindeki blok yapılanmasından çok blok içerisindeki iş parçacığı yapılanmasından faydalanılması gerektiğidir [10].

Anders Eklund ve diğerleri GPU destekli geçmiş ve günümüzdeki görüntü işleme işlemlerini, en çok kullanılan medikal görüntüleme (görüntü kaydı, görüntü bölütleme, gürültü ayıklama) çalışmalarını incelemişler ve bir GPU'nun her ne kadar CPU'dan hızlı olsa bile global bellek azlığından dolayı bu hız farkının azalmasının GPU için bir dezavantaj oluşturduğunu, bu problemin global bellek bant genişliğinin artırılması ile çözüldüğünü ve doku belleğin gelecekte 4B enterpolasyonu desteklemesinin yararlı olacağını vurgulamışlardır [11].

John D. Owens ve diğerleri CUDA ile 3B akciğer tomografi görüntüsü üzerinde deforme görüntü kayıt algoritması uygulamışlardır. Veri kümesi boyutu ne olursa olsun CPU koduna göre 55 kat hız sağlanmış ve bu konuda mevcut uygulamalar arasında en iyi sonucu elde etmişlerdir [12].

Muhammed A. Shehab ve diğerleri yaptıkları çalışmada görüntü bölütlemede GPU programlamayı Bulanık C-Means (FCM) ve yeni versiyonu olan Tip-2 Bulanık C-Means (T2FCM) algoritmaları ile uygulamışlardır. Bu çalışmada, GPU uygulaması

çalışma zamanı CPU uygulaması çalışma zamanına göre göre FCM için %80, T2FCM için ise %74 daha azalarak başarılı bir sonuç elde edilmiştir [13].

Harish P. ve Narayanan P. J., Nvidia 8800GTX GPU kullanarak 1.5 saniyede 10 milyon köşeli graftaki tekil en kısa yolları hesaplamışlardır. Bazı durumlarda optimal sıralı algoritmanın GPU mimarisinde en hızlı olmadığını belirtmişler, yine de GPU'ların, yüksek performanslı işlemciler olarak büyük bir potansiyele sahip olduklarını vurgulamışlardır [14].

Manavski S.A. ve Valle G., Protein ve DNA verilerinin benzerliklerini araştıran Smith-Waterman algoritmasının CUDA ile hızlandırılması çalışmasını yapmışlardır. Yapılan uygulama ile daha önceki denemelerden 2 ila 30 kat daha hızlı sonuç almışlardır [15].

Bell N. ve Garland M., SpMV (Sparse Matrix Vector Multiplication/Seyrek Matris Vektör Çarpımı) işlemleri için yaşanan performans sorunlarını çözmek için CUDA ile bir method geliştirmişlerdir. Yapısal ızgara temelli matrislerde GPU üzerinde 36 GFlop/s ve çift hassasiyette 15 GFlop/s performans elde etmişlerdir. Yapısal olmayan sonlu eleman matrisleri için tek hassasiyette 15 GFlop/s ve çift hassasiyette 10 GFlop/s performans elde etmişlerdir. Bu sonuçların 4 çekirdekli bir Intel işlemcide üretilen sonuçların 10 katından daha fazla olduğu görülmüştür [16].

Lin. C.Y. ve diğerleri Shell Sort (Kabuk Sıralama) algoritmasını CUDA üzerinde uyarlayıp geliştirerek, GPU üzerindeki QuickSort (Hızlı Sıralama) algoritması ile karşılaştırmışlardır. 32 milyon veri Kabuk Sıralama ve Hızlı Sıralama yaptırılarak performans değerleri ölçülmüştür. Kabuk Sıralamanın Hızlı Sıralamadan 2 kat daha hızlı olduğu tespit edilmiş ve Kabuk Sıralamanın çok çekirdekli mimariler için uygun olduğunu belirtmişlerdir [17].

Görüntü işleme kullanan en önemli alanlardan biri de sağlık sektörüdür. Bazı hastalıkların tespit edilebilmesi için tıbbi görüntüleme aygıtları kullanılarak hastanın vücudunun belli bölgerinin görüntülenmesi ihtiyacı ortaya çıkmıştır. Sağlık sektöründe bu alan Radyoloji bölümüdür. Bu alanda oluşturulan görüntüleme

şekillerinden bazıları MR, Tomografi, Ultrason, Röntgen, Ekokardiyografidir. Sağlık sektöründe zamanın öneminden dolayı GPU teknolojileri de hastanelerde kullanılması gereken bir teknolojidir. Bu çalışmada MR görüntülerinin daha verimli saklanabilmesi için GPU teknolojileri kullanarak bir uygulama geliştirilmiştir. CUDA ile geliştirilen bu metodu aynı zamanda CPU üzerinde geliştirilen seri uygulama ile karşılaştırarak performansın yeterli olup olmadığına değinilmiştir.

Tez çalışmasının ikinci bölümüde CUDA ile yapılmış olan uygulamanın kullandığı teknolojiler, teknikler ve özellikleri üzerinde durulacaktır. Bu bölümde CUDA mimarisi, CUDA donanımsal ve yazılımsal yapısı, CUDA ile kodlamanın nasıl yapılacağı, görüntü bölütleme, kenar bulma yöntemleri, medikal görüntüler, sayısal arşivleme ve gerekliliği üzerinde durulmuştur.

Çalışmanın üçüncü bölümünde CUDA uygulaması ile medikal MR görüntüsü içindeki ROI bölgesinin nasıl tespit edildiği, NON-ROI bölgesinin nasıl görüntüden çıkarıldığı, yeni dosya yapısının nasıl oluşturulduğu ve nasıl sıkıştırıldığı ve performans testleri üzerinde durulmuştur. Yine bu bölümde CUDA ile yapılmış paralel uygulama ile CPU üzerinde yazılmış seri kod çalışma zamanları karşılaştırılmıştır. Ayrıca orijinal MR görüntü verisi ile uygulama tarafından oluşturulan sıkıştırılmış verilerin boyutları karşılaştırılmıştır.

Çalışmanın dördüncü ve son bölümünde uygulamada elde edilen performans verileri değerlendirilerek çalışmanın verimliliği ve gelecekte yapılacak çalışmalar hakkında bilgi verilmiştir.

2. MATERYAL VE YÖNTEM

GPU üzerinde GPGPU (General Purpose Computing/Genel amaçlı Hesaplama) işlemleri yapabilmek ve yapılan işlemleri görüntü çıktısı olarak alabilmek için GPU komut setlerine ihtiyaç duyulur. Bu programlamaya heterojen programlama adı verilir. GPU programlama için CUDA ve OpenCL teknolojileri kullanılırken, GPU'da işlenen verileri görüntü çıktısı olarak alabilmek için DirectX ve OpenGL teknolojileri kullanılır.

2.1. CUDA

CUDA (Compute Unified Device Architecture/Birleştirilmiş Aygıt Mimarisi Hesaplama), 2006 yılında NVIDIA'nın GPU (Graphics Processing Unit/Grafik İşlem Birimi) kapasitesini kullanarak hesaplama performansında önemli derecede artışlara olanak veren paralel hesaplama mimarisidir. Yazılım geliştiriciler, bilim adamları ve araştırmacılar CUDA çekirdekli GPU'lar ile görüntü ve video işlem, hesaplamaya dayalı biyoloji ve kimya, akışkan dinamiği, bilgisayarlı tomografi, sismik analiz, ışın izleme gibi geniş bir kullanım alanı bulabilmektedir [18].

Görüntü ile ilgili işlemler çok zaman alabilmektedir. CPU (Central Proccessing Unit/Merkezi İşlem Birimi) işletim sistemine ait işlemleri ve diğer yazılımsal işlemleri yaptığından dolayı görüntü ile ilgili işlemleri yapmakta zorlanacak hatta yetişemeyecektir. Bu sırada bilgisayar hiçbir işlem yapamayacak hatta kilitlenmeler söz konusu olacaktır. CUDA ile görüntü işleme işlemleri yapıldığında CPU üzerindeki işlem GPU kaynaklarına aktarılacak böylece CPU yükü hafifletilmiş olur.

CUDA sadece GPU üzerinde işlem yapabilmektedir. CPU ve sistem kaynaklarına doğrudan erişemez. Sadece CPU ve ana bellekten gelen verileri işleyerek tekrar ana bellek ve CPU ya gönderir.

CUDA; Linux, Windows ve MAC OSX işletim sistemleri üzerinde çalışabilir ve C, C++, Phyton, Fortran ve C# dillerinde geliştirme desteği bulunur.

2.2. OpenCL

OpenCL (Open Computing Language), Apple tarafından 2008 yılında geliştirilmiş, hem CPU hem de NVIDIA ve ATI gibi grafik kartı üreticilerin GPU'ları üzerinde işlemler yapılabilmesini sağlayan platformdan bağımsız, açık bir standarttır. OpenCL standartları aygıtlar arasında taşınabilir, üretici ve aygıt bağımsız geliştirme olanağı sağlamak amacıyla geliştirilmiştir. OpenCL standartları, Khronos adlı bir konsorsiyum tarafından ortaya konulmuştur. Apple, Qualcomm, Intel, NVIDIA, AMD, Samsung bu konsorsiyumun üye şirketleridir.

OpenCL'deki temel amaç çok çekirdekli yapıya sahip sistemlerdeki kaynakları hızlı ve etkin kullanmaktır. Çekirdek sayısı artığında paralel işlem sayısı da artacağından bu işlemleri dinamik olarak yönetmek gerekecektir. OpenCL bu konuda kendini sürekli geliştirerek bu problemin üstesinden gelebilmiştir.

OpenCL program geliştiricilerine C dili (OpenCL C) ile geliştirme imkanı sunar. Klasik C diline göre farklılıkları vardır. Fonksiyon işaretçileri, özyineleme, değişken uzunluklu diziler kaldırılmış, hafıza yönetimi için global, local, constant ve private sınıfları eklenmiştir.

.

2.3. CUDA ile OpenCL Kullanımları ve Karşılaştırılması

Görüntü işleme kullanan büyük firmalar CUDA ve OpenCL kullanarak uygulamalarının performanslarını artırmışlardır. Bunlara örnek [19] olarak:

1. **Adobe Photoshop CC**
    a. CUDA: 3B ve Çoklu GPU desteği
    b. OpenCL: Spesifik özellik yok
2. **Adobe After Effects CC**

a. CUDA: Mercury Grafik Motoru içinde 30 efekt
b. OpenCL: Spesifik özellik yok

3. Autodesk Maya

a. CUDA: Artırılmış model karmaşıklığı ve Geniş sahne özelliği
b. OpenCL: Fizik simulasyonu

4. Avid Medai Composer

a. CUDA: Daha hızlı video efektleri
b. OpenCL: Kullanılmamış

5. Final Cut Pro X

a. CUDA: Kullanılmamış
b. OpenCL: Daha hızlı genel oynatma ve üçüncü şahıs efekti

6. Red Red-X

a. CUDA: 2 GPU desteği ve Hızlandırılmış Debayering
b. OpenCL: 1 GPU desteği

7. Sony Vegas Pro

a. CUDA: Daha hızlı video efekleri ve çözümleme
b. OpenCL: Spesifik özellik yok

8. The Foundry HIERO

a. CUDA: Daha iyi etkileşim

9. The Foundry NUKE & NUKEX

a. CUDA: Daha hızlı efektler

10. The Foundry Mari

a. CUDA: Artan model karmaşıklığı

Yukarıdaki programların CUDA ve OpenCL platfromları ile geliştirdiği özellikler incelendiğinde CUDA ile geliştirilen özelliklerin daha etkili olduğu görülmüştür. Bu özellikler geliştirilen platformun donanıma yakın olmasıyla doğru orantılıdır. Örenğin; CUDA ve OpenCL kullanan programlardan Adobe CC ile NVIDIA ekran kartı bulanan bilgisayarlar ile yapılan derleme işlemlerinde [20], işlem süreleri CUDA için sırayla 0:56, 0:36, 1:02 iken OpenCL için 3:42,0.38,7:23 olarak bulunmuştur. Bu da platformun donanıma ne kadar yakın olduğunun göstergesidir.

CUDA’nın OpenCL’ye göre şu avantajları vardır: Firmanın kendi geliştirdiği grafik kartı ve GPU için program geliştiricilere bir platform oluşturması, GPU ve grafik kartı yapısının daha iyi anlaşılmasını ve yönetilmesini sağlar. Grafik kartı ve GPU yapısının diğer grafik kartlardan ayrıldığı noktalarda bu ek özellikleri kullanabilmeyi kolaylaştırır. Ayrıca geliştirme yaparken her GPU’nun özelliği tam bilindiğinden çıkabilecek olası hataları yönetmek daha kolaydır. Grafik kartı, GPU veya platformdan kaynaklanacak sorunlar hakkında doğrudan tek muhataptan bilgi alınması da CUDA’nın avantajlarındandır.

OpenCL’nin CUDA’ya göre avantajları şu şekildedir: CUDA tek bir firma tarafından geliştirilmiş ve açık bir platform değildir ve sadece NVIDIA ekran kartlarında kullanılabilir. OpenCL ise açık bir platformdur, tüm CPU ve GPU sistemlerinde kullanılabilir. Geliştiricisi bir konsorsiyum olduğundan daha esnek bir yapıya sahiptir. OpenCL bu sayede taşınabilir ve dinamik bir yapıdadır.

Her iki platformun avantaj ve dezavantajları göz önünde bulundurulduğunda şöyle bir kanıya varılabilir: Eğer bir CUDA destekli NVIDIA ekran kartınız varsa, CUDA ile geliştirme yapmanız daha uygundur veya CUDA ile geliştirilen uygulamalar tercih edebilirsiniz; eğer yoksa OpenCL ile geliştirme yapmanız daha uygundur veya OpenCL ile geliştirilen uygulamaları tercih edebilirsiniz.

2.4. OpenGL

OpenGL (Open Graphics Library/Açık Grafik Kütüphanesi), grafik kartı desteğiyle 2B ve 3B grafikleri ekrana çizebilen bir API’dir. 1992 yılında piyasaya çıkmış ve günümüzde yaygın olarak kullanılmaktadır. Windows, Mac, Linux iştetim sistemleri ve Playstation gibi oyun konsollarına desteği vardır. Khronos konsorsiyumu tarafından destek verilen OpenGL, açık bir platformdur ve taşınabilirdir. C, C++, C#, Ada, Fortran, Python, Perl ve Java gibi birçok dili destekler.

OpenGL; Sanal gerçeklik, simulasyon ve video oyunlarında sıkça kullanılır. Bunlara şu örnekler verilebilir:

1. **Fotograf ve Video**

   Adobe After Effects, Adobe Photoshop, Adobe Premiere Pro.

2. **Modelleme ve CAD**

   3D Studio Max, Autodesk AutoCAD, Autodesk Maya, Blender, Google SketchUp, SAP2000.

3. **Görselleştirme ve Diğer**

   Enhanced Machine Controller (EMC2), Google Earth, InVesalius, Mari, PyMOL, QuteMol, Really Slick Screensavers, SpaceEngine, Stellarium, Universe Sandbox, Vectorworks.

4. **Oyun**

   Dota 2, Tomb Raider, Counter Strike, Half Life, Wolfenstein, Far Cry, Max Payne, Need For Speed, Serious Sam, Minecraft, F1.

OpenGL platformdan bağımsız taşınabilir olduğundan pencere oluşturma, klavye ve fare kontrolü gibi işlemleri doğrudan OpenGL ile yapılamaz. Bu problemlerin çözümü için GLUT (OpenGL Utility Toolkit/OpenGL Yardımcı Araç Kiti) geliştirilmiştir.

2.5. DirectX

DirectX Microsoft tarafından 1995 yılında geliştirilen, grafik kartı desteğiyle 2B ve 3B grafikleri ekrana çizebilen bir API'dir. İlk sürümlerinde sadece grafik desteği sunarken günümüzde joytsick, network, klavye fare girişlerinin kontrolünü de sağlar.

DirectX'in bileşenleri aşağıda verilmiştir:

DirectInput: Fare, klavye, joystick gibi aygıtların giriş, çıkış ve verilerini kontrol eder.

DirectSound: İki ve üç boyutlu sesler için kullanılır.

DirectMusic: Etkileşimli ses bileşenidir.

Direct3D: Ekrana 2 veya 3 boyutlu görüntülerin çizilmesini sağlar.

DirectPlay: Online oyun bağlantılarını sağlar. TCP/IP, Bluetooth ve modem bağlantılarını kontrol eder.

DirectDraw: Grafik kartı özelliklerine hızlı erişmeyi sağlar. Tam ekran pencere ve gömülü ekranların çalışmasını sağlar. 2B işlemleri destekler.

DirectShow: API sayesinde bilgisayarda ve internette yer alan ortam dosyalarını bulma ve oynatma imkanı sağlar. ASF, AVI, DV, MPEG, MP3, WMA, WMV, WAV gibi dosya tiplerini destekler.

DirectX kullanan bazı oyunlara örnek olarak; Assassin's Creed, BattleField, Far Cry, Grand Thieft Audio, Max Payne, Medal of Honor, MotoGp, NBA, Resident Evil, Hitman, FIFA, Age of Empires, Call of Duty, Football Manager, Grand Thieft Audio, Crysis, Mafia, Need For Speed, Sniper Elite, Tekken verilebilir.

2.6. OpenGL ile DirectX Karşılaştırması

OpenGL ve DirectX her ikisi de grafiksel çizim işlemlerini desteklese popülerlik ve kullanım olanaklarında farklılıklar vardır. Microsoft'un oyun piyasasına hakim olması sebebiyle DirectX oyun piyasasında daha popüler iken OpenGL daha çok çizim uygulamalarında yaygındır. Program geliştiricilerin kullanımlarına göre de DirectX, OpenGL'ye göre daha yaygındır. Çünkü kullanması daha kolay, dokümantasyonu ve desteği fazladır.

2.7. GPU ile CPU Kıyaslaması

Bir CPU çekirdeği, tek bir yönerge akışını bir defada hızlı bir şekilde yürütmek için tasarlanmıştır. GPU'lar ise birçok paralel yönerge akışını hızlı bir şekilde yürütmek için tasarlanmışlardır [21].

CPU, ara belleği, genel bellek erişim gecikmesini azaltarak performans arttırmak için kullanır. GPU ise ara belleği (ya da paylaşılan belleği) bant genişliğini arttırmak için kullanır [21].

CPU'da bellek gecikmesi, geniş arabellekler ve tahmin mekanizmaları ile yönetilir. Bunlar tasarım üzerinde büyük yer kaplarlar ve çok güç tüketirler. GPU ise gecikmeyi binlerce iş parçacığını bir anda destekleyerek yönetir. Herhangi bir iş parçacığı bellekten yük bekliyorsa, GPU gecikmeye sebep olmadan başka bir işe geçiş yapar [21].

CPU'lar çekirdek başına bir veya iki iş parçacığı yürütme desteği sunarlar. CUDA destekli GPU'lar ise akış işlemcisi (SM) başına 1024'e kadar iş parçacığını destekleyebilir. CPU'nun uygulamalar arasında bir kez geçiş yapması yüzlerce saat döngüsü gerektirirken GPU'nun geçiş yaparken herhangi bir kaybı yoktur [21].

CPU'lar vektörel işlemler için SIMD (Single Instruction Multiple Data – Tek Yönergeyle Birden Çok Veri) birimleri kullanırlar. GPU'lar ise ölçekli yürütme için SIMT (Single Instruction Multiple Thread – Tek Yönergeyle Birden Çok İş Parçacığı) kullanırlar. SIMT, programcının verileri vektörler halinde düzenlemesini gerektirmez, iş parçacıkları için rastgele dağılıma olanak tanır [21].

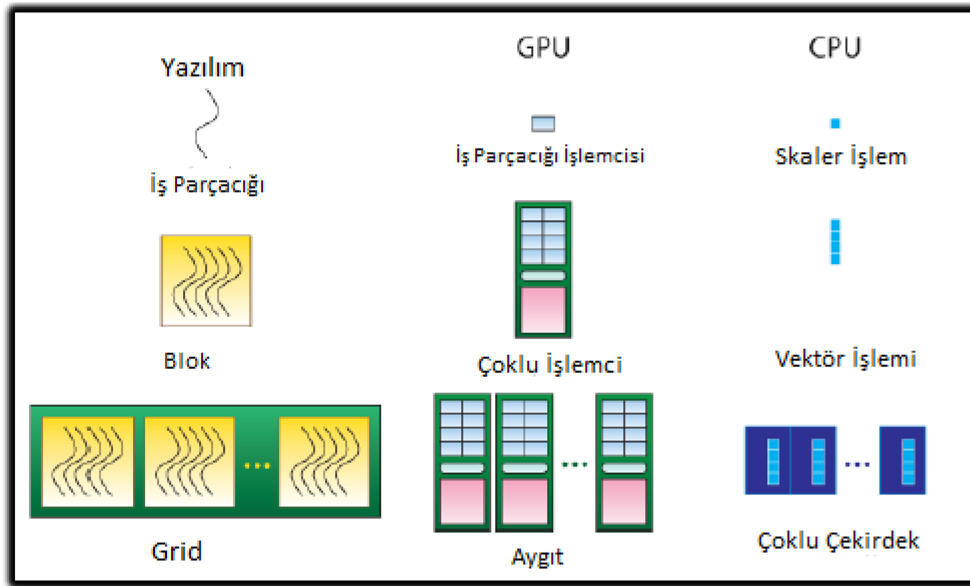
Çizelge 2.1. CPU ve GPU günümüzde bulunan minimum ve maksimum değerleri

	CPU	**GPU**
Bellek Kapasitesi	6 - 64 GB	768 MB - 6GB
Bellek Bant Genişliği	24 – 32 GB/s	100 – 200 GB/s
L2 Cache	8 – 15 MB	512 – 768 KB
L1 Cache	256 – 512 KB	16 – 48 KB

Çizelge 2.1 de CPU ve GPU nun günümüzde sahip olduğu bellek kapasitesi, bellek bant genişliği, L1 ve L2 Cache boyutları gösterilmektedir. Bu çizelge de gösteriyor ki GPU fazla bellek gerektirmeden büyük bant genişliği ile işlemleri çözebilmektedir.

2.8. CUDA GPU Mimarileri

GPU hesaplama kapasitesi birincil versiyon numarası ve ikincil versiyon numarası ile ifade edilmektedir. Birincil versiyon numaraları aynı olan ürünler aynı fiziksel çekirdek mimarisindedirler. Birincil versiyon numarası 1 olan aygıtlar Tesla mimarisine, 2 olan aygıtlar Fermi mimarisine ve 3 olan aygıtlar Kepler mimarisine ve 5 olanlar da Maxwell mimarisine sahiptirler. İkincil versiyon numarası ise bu mimariler geliştirildikçe değişir [22].



Şekil 2.1.CUDA GPU Mimarisi [22]

CUDA GPU mimarisi donanımsal anlamda incelendiğinde her bir iş parçacığının Streaming Processor (Akış İşlemcisi)/SP üzerinde çalıştığı görülmektedir (Şekil 2.1). 8 adet akış işlemcisinin oluşturduğu yapıya Streaming Multiprocessor (Çoklu İşlemci)/SM adı verilmektedir.

SM'ler ise ekran kartı donanımını oluşturur. Örneğin, NVIDIA GT200 grafik kartı 30 adet SM'ye sahiptir. Her bir SM, 8 bloğa kadar görevlendirilebilir şekilde tasarlanmıştır. GT200 işlemcide 30 SM ile 240 bloğa kadar eş zamanlı

görevlendirme olabilir. ızgaraların çoğu 240 bloktan fazlasını içermektedir. GT200'de her bir SM 1024 iş parçacığına kadar görevlendirilebilecek şekilde tasarlanmıştır [22].

CUDA'da uygulamalar GPU'nun özelliklerine göre geliştirilmelidir. Bunun için birincil ve ikincil versiyon numaraları ile birlikte teknik özellikleri Fermi ve Kepler mimarisi Şekil 2.2'de gösterilmektedir.

2.8.1. Tesla Mimarisi

2007 yılında geliştirilen Tesla mimarisinde Tesla kartları için öncelikle doku / işlemci kümelerinde yapılan net bir hiyerarşi vardır. Bu kümeler ölçeklenebilir, tek bir tane olabilir, sekiz veya daha fazla olabilir. Amaç kolayca genişletilebilen bir dizi kümeye sahip olmaktır [23].

Tesla mimarisine sahip grafik kartların genel özellikleri [24] şöyledir:

1. ECC koruma ve en yüksek performans için GPU hızlandırma özellikleri bulunur.
2. Çift DMA (Direct Memory Access) motoru ile çift yönlü hızlı iletişim sunar.
3. GPUDirect ile GPU'lar, network ve depolama arasında hızlı iletişim sunar.
4. Kurumsal anlamla iyi bir şekilde CUDA ve işletim sistemi yönetimi sağlar.
5. ISV sertifikası ile sorunsuz kurulum işlemleri yapılır.
6. Sistem yönetimi araçları ile GPU'ları kolay bir şekilde izleme imkanı sunar.

Tesla mimarisi kullanım alanları [25] aşağıda belirtilmiştir:

Endüstri: 3ality Intellicam, 3DAliens Glu3d, Aon Benfield Pathwise, Abaqus/Standard, Abinit, ABSoft Neat Video, Accelereyes Arrayfire, Acceleware AxRecon, Acceleware RTM, Accuweather Cinemative HD, Accuweather Storyteller, ACEMD, ACES III, ADF, Adobe After Effects CC, Adobe Photoshop CC, Adobe Premiere Pro CC, Adobe SpeedGrade CC, Agilent

Technologies ADS, Agilent Technologies EMPro, Altair AcuSolve, Altair FEKO, Amira, ANSYS Fluent.

Biyoinformatik: Arioc, BarraCUDA, CUDASW++, CUSHAW, GATK, G-BLASTN, GPU-Blast, MUMmerGPU, NVBIO, NVBowtie, PEANUT, REACTA, SeqNFind, SOAP3, SOAP3-dp, UGene, WideLM.

Hesaplamaya Dayalı Finans: MathWorks MATLAB, Aon Benfield Pathwise, Accelereyes Arrayfire, Altimesh's Hybridizer C#, Hanweck Associates, MiAccLib, MISYS Global Risk, Murex, Numerical Algorithms Group, QuantAlea's Alea.cuBase F#, RMS, SciComp, Numerical Algorithms Group, QuantAlea's Alea.cuBase F#, Streambase, SunGard Adaptiv Analytics, Synerscope's Synerscope Data Visualization, Tanay ZX Lib (Fuzzy Logic), Xcelerit.

Derin Öğrenme: Caffe, Theano, Torch7, Trakomatic OSense, OTrack, Cuda-convnet2.

Bilgisayar Destekli Tasarım: AutoCAD, AutoCAD Designt Suite, Autodesk 3ds Max, Bunkspeed Suite, Dassault Systemes CATIA - Live Rendering, Inventor, NVIDIA Iray, PTC Creo Parametric, Siemens PLM NX and Teamcenter, Revit, RTT DeltaGen.

Veri, Analitik ve Veritabanları: GPU-Quicksort, HiPLAR, ParStream, Jedox Palo, WideLM, ParStream, GPU-LIBSVM, Wolfram Mathematica, MathWorks MATLAB.

Medikal Görüntülüme: Acceleware AxRecon, Digisens.

Fizik: Chemora, Chroma, ENZO, GTC, GTS, MILC, PIConGPU, QUDA, RAMSES.

Savunma ve Zeka: Eternix Balze Tarra, Exelis ENVI, GAIA, GeoWeb3d Desktop, Incognia GIS, InterGraph Motion Video Analyst, MrGeo, Intuvision Panoptes, OpCoast SNEAK, Savant.

2.8.2. Fermi Mimarisi

2010 yılında NVIDIA mühendisleri yeni bir GPU mimarisi tasarlamak için yola çıkmıştır. Bu mimari, bir GPU'nun yapı taşlarını yani birbirlerine nasıl bağlı olduklarını ve nasıl çalıştıklarını tanımlamıştır [26].

İtalyan nükleer fizikçi Enrico Fermi’den adını alan Fermi, tam geometri işlemeyi GPU'ya dahil ederek mozaik yer değiştirme eşleme olarak adlandırılan önemli bir DirectX 11 tekniğini kullanıma sokmuştur [26].

Fermi mimarisine sahip grafik kartların genel özellikleri [27] şöyledir:

1. Quadro ölçeklenebilir geometri motoru ile CAD, DCC ve bilimsel uygulamalarda performansı artırarak işlemlerin karmaşık modellerle ve görüntülerle ilişkili bir şekilde akmasını sağlar.
2. Bir milyardan fazla renk içeren dinamik aralığı sayesinde canlı görüntü kalitesi sağlayan zengin bir 30-bit renk motoru bulunur. 128 kata kadar sahne kenar yumuşatma özelliği ile görsel yapaylıkları azaltır.
3. OpenGL 4.0 ve DirectX11 desteği sağlar, performansı etkilemeden sinema kalitesinde ortam oluşturur.
4. Büyük 4B veri setlerinin gerçek zamanlı işlenmesini destekler.
5. Hata Düzeltme Kodları (ECC) ilk defa kullanılmıştır.
6. GigaThread motoru sayesinde eski mimarilere göre 10 kata kadar daha hızlı içerik değiştirme, senkron çekirdek yürütme işlemlerini yapabilir.
7. 6 GB’a kadar DDR5 bellek desteği bulunur.
8. Parallel DataCache özelliği ile üst seviyede verimliliği bulunan, paylaşımlı bellek ile bir araya gelen bir ön bellek yapısını destekleyerek gerçek zamanlı ışın izleme ve doku filtreleme gibi işlemlerin hızlı bir şekilde yapılmasını sağlar.

Fermi mimarisi kullanım alanları [28] aşağıda belirtilmiştir:

Endüstri Alanında: Autodesk, Dasault Ststeme Catia, Dasault Systeme Solidworks, PTC Creo Parametric 3.0, Siemens PLM.

Medya ve Eğlence Alanında: Adobe, Autodesk, Autodesk Flame, Avid Media Composer, 2d3 Sensing Boujou, Accelereyes Jacket, Accelrys Discovery Studio, Bentley MicroStation, Erdas Imagine, Esri ArcGIS for Desktop 10.2, Wolfram Mathematica, Tripos SYBYL, Presagis Creator, Presagis Terra Vista, Presagis Stage, Presagis Vega Prime, Presagis Sensor Prime, Presagis Lyra.

Enerji Alanında: Aveva PDMS, Baker Hughes JOA Jewel Suite, Headwawe Suite, Ffa Geoteric, Ffa SEA3D Pro, FFa SVI Pro, InterGraph SmartPlant 3D, Halliburton GeoProbe, Halliburton Seisworks, Halliburton DecisionSpace, Halliburton 3D Wellbore, Halliburton Geological, LMKR Geographics, Paradigm Geophysics SKUA, , Paradigm Geophysics VoxelGeo, , Paradigm Geophysics Attibutes, Emerson Irap RMS, Schlumberger GeoFrame, Schlumberger Petrel, Schlumberger Petrel Seismic Interpr., Schlumberger Petromod, Schlumberger Techlog, Schlumberger Petrel Reservoir Modeling TerraSpark Insight Earth, IHS Kingdom suite.

2.8.3. Kepler GPU Mimarisi

NVIDIA’nın 2012'de çıkardığı Kepler mimarisinde
Mantık kontrolü yerine işlem çekirdeklerine daha büyük oranda alan ayrılmasına olanak veren bu yeni SM tasarımı ile daha fazla işlem performansı ve verimlilik sunmaktadır [29].

Çok sayıda CPU çekirdeğinin tek bir Kepler GPU’yu eşzamanlı olarak kullanmasına olanak vererek CPU atıl zamanını düşürür, programlanabilirlik ve verimliliği büyük ölçüde geliştirmektedir [29].

2.8.4. Maxwell Mimarisi

2014 yılında yayınlanan Maxwell mimarisi ile güçlendirilen GPU'lar, yeni VXGI (Voxel Global Illumination/Voksel Global Aydınlatma) teknolojisini kullanarak ilk kez dolaylı ışığı dinamik şekilde derleyebilen GPU'lardır. Işık oyun ortamında daha gerçekçi bir şekilde etkileşim içinde olduğundan sahneler gerçek hayata daha yakın ve inanılır olmaktadır [30].

Grafik açıdan yoğun oyunlar oynamak, ayarları yükseltmek ve düşük ayarlarda yüksek resim karesi hızlarının keyfini çıkarmak arasında seçim yapmayı gerektirmektedir. Örneğin; GeForce GTX 900 serisi GPU'lar, muhteşem çözünürlük ve yüksek FPS (Frame Per Second/Saniyedeki Çerçeve Sayısı) ile grafik işlemlerine olanak vermek üzere önceki nesil grafik kartlarının performansını iki katına çıkaran MFAA (Multi Frame Anti Aliasing/Birden Çok Çerçeveli Kenar Yumuşatma) teknolojisini destekler [30].

Kullanıcılara 1080p monitörde 4K deneyimi sunmak üzere, görüntüyü ölçeklemek için gelişmiş bir filtre kullanan DSR (Dynamic Super Resolution/Dinamik Süper Çözünürlük) teknolojisini kullanırlar. Görüntü akış performansından ödün vermeden otomatik olarak optimize edilir [30].

Şekil 2.2. Fermi ve Kepler mimarileri karşılaştırması

2.9. CUDA Programlama Mimarisi

2.9.1. Çekirdek (Kernel) Nedir?

CUDA'da bir kodun GPU üzerinde çalışacak olan kısmına "Çekirdek" denir. GPU, veri kümesinin her elemanı için bir çekirdek kopyası oluşturulur. Bu çekirdek kopyaları "İş Parçacığı" olarak adlandırılır [31].

2.9.2. Izgara (Grid) Nedir?

Izgara, blokların bir araya gelerek oluşturdukları yapılardır. Her bir çekirdek çağrısı bir ızgara oluşturur. Izgaralar 1B, 2B veya 3B olabilirler. Bu boyutlar "gridDim.x", "gridDim.y" ve "gridDim.z" şeklinde ifade edilir. "gridDim" terimi ızgaradaki blok sayısını ifade eder [31].

2.9.3. Blok (Block) Nedir?

Blok yapısı, paralel olarak çalışma yeteneğine sahip iş parçacıklarından oluşur.1B, 2B veya 3B olabilirler. Izgaralar içerisinde dizilerek gruplanırlar. Her bir bloğun, ızgara içerisinde bu 3 boyutta kendine ait tekil bir indisi vardır. Bu indisler “blockIdx.x”, “blockIdx.y” ve “blockIdx.z” şeklindedir. İçerisinde bulunan iş parçacıkları satır ve sütun sayısına göre boyutlanırlar ve bu boyutlar “blockDim.x”, “blockDim.y” , “blockDim.z” olarak ifade edilmektedir. “blockDim” terimi bloktaki iş parçacığı anlamına gelir [31].

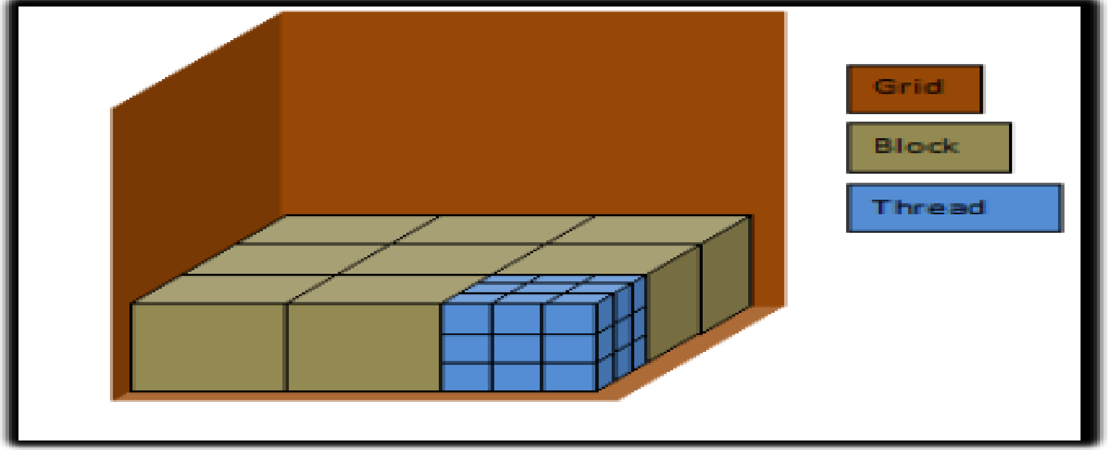
Blok sayısı GPU hesaplama kabiliyetine göre değişiklik gösterir. Örneğin; hesaplama kabiliyeti 2.0 olan bir GPU’da bir ızgara içerisinde 3B en fazla 65535*65535*65535 adet blok bulunur.

2.9.4. İş Parçacığı (Thread) Nedir?

İş Parçacığı, CUDA mimarisindeki en küçük birimdir. bloklar içerisinde 1B, 2B veya 3B olabilirler. Kendi aralarında paralel olarak aynı kod parçasını çalıştırırlar. İş parçacıkları bloklar içerisinde dizilerek gruplanırlar. Farklı bloklardaki iş parçacıkları beraber çalışamazlar [31].

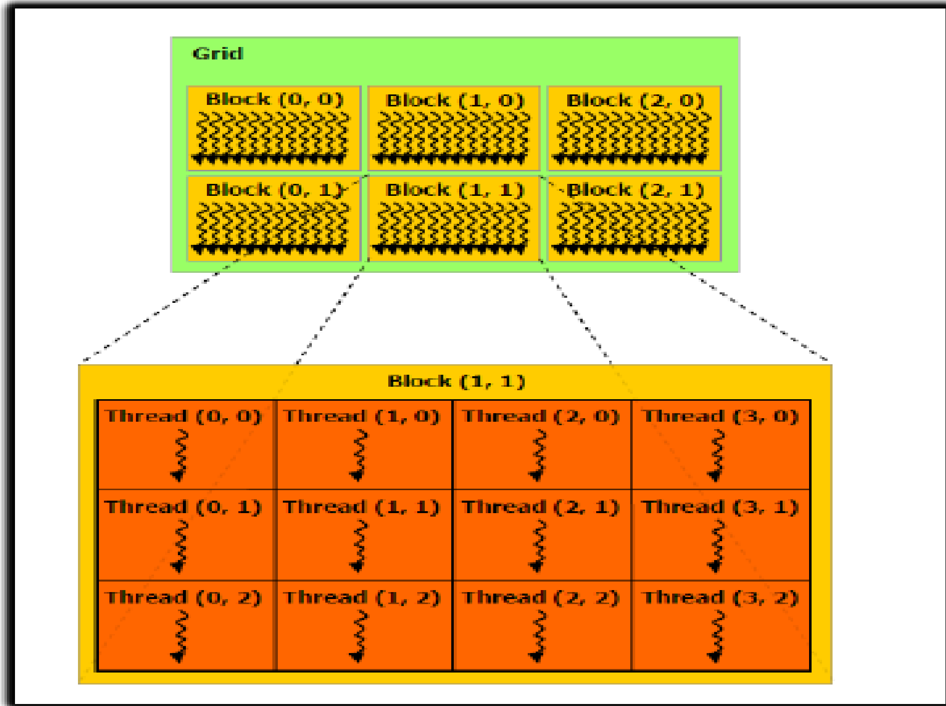
Her bir iş parçacığının bir Program Sayacı (PC), kaydedicisi (Register) ve durum kaydedicisi (State Register) vardır. Her iş parçacığının blok içerisinde kendine ait bir tekil indisi vardır. Bu indisler; “threadIdx.x”, “threadIdx.y” ve “threadIdx.z” şeklinde ifade edilir [31].

Bir blok içerisinde bulunacak iş parçacığı sayısı sınırlıdır ve hesaplama kabiliyetine göre değişkenlik gösterir. Örneğin; hesaplama kabiliyeti 2.0 olan bir GPU için bir blok içerisinde en fazla 3B 1024*1024*64 iş parçacığı olabilir.



Şekil 2.3. Izgara, Blok ve İş Parçacığı yapısı [31]

Özetle Şekil 2.3’te de görüldüğü gibi ızgara 3 boyutlu bloklardan, Her bir blok da 3 boyutlu iş parçacıklarından oluşur.



Şekil 2.4. Izgara örneği [31]

Şekil 2.4 incelendiğinde ızgara 2x3'lük bloktan ve her bir blok 3x4'lük iş parçacıklarından oluşmaktadır. Buna göre **gridDim.x**=3 (gridin blok sütun sayısı), **gridDim.y**=2 (gridin blok satır sayısı), **blockDim.x**= 4 (bloğun iş parçacığı sütun sayısı), **blockDim.y**= 3 (bloğun iş parçacığı satır sayısı kadar) olur.

2.10. CUDA Bellek Mimarisi

2.10.1. Yerel Bellek (Local Memory)

Yerel bellek yapısı, iş parçacığına özel bir yapıdır. Buradaki veriler, sahip olunan her bir iş parçacığının çalışması bitene kadar saklanır. Diğer iş parçacıkları bu verilere ulaşılamazlar. Yerel bellek üzerine kendi iş parçacığı tarafından üzerine veri yazılabilir ve yerel bellek üzerinden veri okunabilir. Bu belleğe program tarafından erişilemez [32].

2.10.2. Paylaşımlı Bellek (Shared Memory)

Sadece aynı blokta bulunan iş parçacıkları bu belleğe veri yazıp okuyabilirler. Aynı blok içerisindeki iş parçacıkları tarafından kullanılabilir. Çok hızlı çalışır. Ancak iş parçacığı senkronizasyonu gerektirdiğinden dolayı kullanımı risklidir.

2.10.3. Genel Bellek (Global Memory)

Çalışan bütün iş parçacıkları ve program tarafından erişilebilen, üzerinden veri okunabilen ve üzerine veri yazılabilen bir bellek türüdür. Bu belleğin bant genişliği düşüktür [32].

2.10.4. Sabit Bellek (Constant Memory)

Read Only (sadece okuma yapılabilen) bellek modelidir. Çekirdek çalıştırıldığında değişmeyecek veriler için kullanılır. 64 KB'lık büyüklüğe sahiptir. NVIDIA'nın değişmez belleği tasarlamasındaki amaç bellek bant genişliğinin Global Belleğe göre daha az olmasındandır. Uygulamada veriler sadece okunacaksa bu bellek modeli kullanılmalıdır. Değişmez bellekten dinamik olarak yer alınmasına izin verilmemektedir. Değişmez belleğe yalnızca CPU tarafından veri yazılabilir [32].

2.10.5. Doku Bellek (Texture Memory)

Doku belleği de değişmez bellek gibi Read Only yapıya sahiptir. Bu yapı performansı artırır ve bellek trafiğini azaltır.Bu yapı bazı durumlarda DRAM'de daha az istek trafiği ile daha etkin bantgenişliği imkanı sunar. Doku önbellekleri grafik uygulamalarını derlemek için tasarlanmıştır [32].

2.11. CUDA Erişim Fonksiyonları

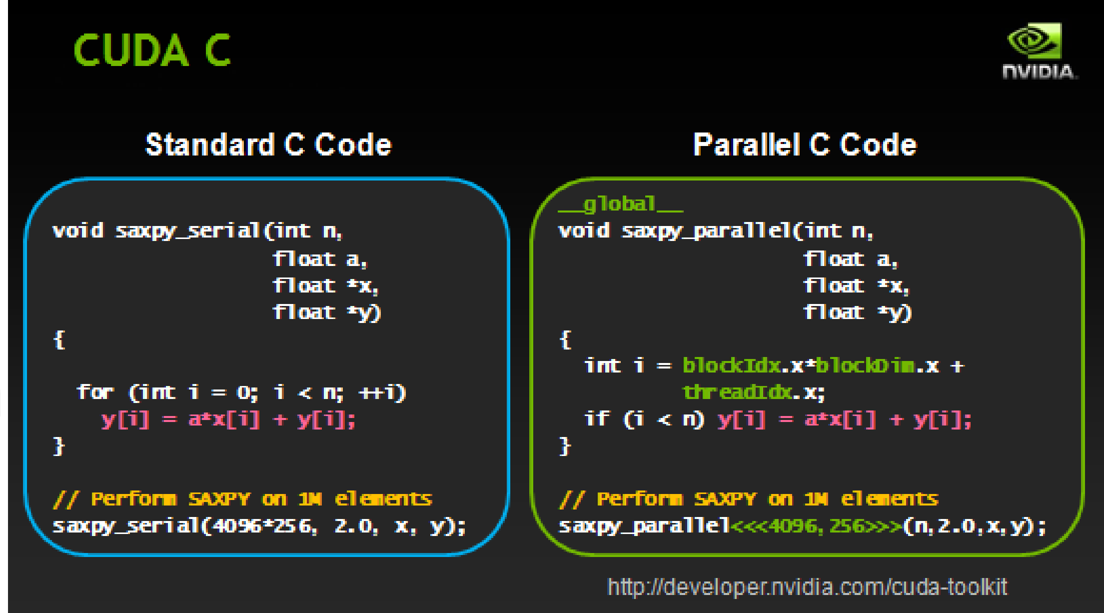
2.11.1. global

Çekirdek kodu, **__global__** ile nitelendirilir. CPU üzerindeki kod ile çekirdek çağrısı yapılırken çekirdek fonksiyon adı yazılır ve "<<<" ve ">>>"parantezleri yazılır. Bu parantezler arasına yazılan ifadeler sırasıyla grid'teki blok sayısı ve bloktaki iş parçacığı sayısı bilgilerini gösterir. Örnek bir çekirdek çağrısı:

__global__void Fonks_Adi<<<BlokSayisi,IsParcacigiSayisi>>>(parametreler);
şeklindedir.

"__global__" olarak yazılmış fonksiyonunun dönüş tipi void olmalıdır. Çünkü geri dönecek veri de parametrelerden biri olarak yazılır. Bu fonksiyon sadece CPU

tarafından çağrılabilir. Başka bir çekirdek tarafından da çağrılamaz. “__global__” metod GPU üzerinde çalışır.



Şekil 2.5. C dili ile yazılmış Ardışık/Sequental Kod ve CUDA paralel kod [18]

2.11.2. device

Sadece GPU üzerinde çalışır. CPU tarafından bu fonksiyona erişilemez. Yani çekirdek içerisinden bir GPU fonksiyonu çağrısı yapılabilen fonksiyonlardır.

2.11.3. host

Sadece CPU tarafından kullanılır. Fonksiyona hiçbir niteleyici yazılmasa bile bu, “__host__” niteleyicisi olarak kabul edilir.

2.12. CUDA Bellek Operasyonları

2.12.1. Hafıza Ayırma Operasyonları

cudaMalloc(void ** dizi, size_t boyut)fonksiyonuC dilindeki Malloc ile eşdeğerdir, ancak hafiza GPU üzerinde ayrılır [33].

cudaMemset(void * dizi, intdeger, size_t boyut) fonksiyonu istenilen değer, istenilen boyut kadar bellek alanına yerleştirilebilir [33].

cudaFree(void* dizi) fonksiyonu ise C dilindeki Free ile eşdeğerdir, ancak burada GPU üzerinden ayrılan hafıza serbest bırakılır [33].

2.12.2. Veri Kopyalama İşlemi

cudaMemcpy(void *hedef, void *kaynak, size_t boyut, enumBellek_Kopya_Yonu)fonksiyonu ile yapılır.

Burada "hedef" kopyalanacak verinin hedefi, "kaynak" ile kopyalanacak verinin kaynağı, "boyut" ile kopyalanacak verinin boyutu ve "Bellek_Kopya_Yonu" ile veri kopyalamanın yönü belirtilir. "enum" veri tipi ile ifade edilen değişkenler daha önceden belirlenmiş kopyalama yönleridir. Buna göre veri alışverişinin yönü CPU'dan GPU'ya, GPU'dan CPU'ya ya da GPU'nun kendi içerisinde olabilir. CPU'dan CPU'ya bir yön bulunmamaktadır çünkü bu durumda uygulama GPU uygulaması değil klasik bir CPU uygulaması olur [33].

2.13. CUDA Çalışma Prensibi

GPU üzerinde bir hesaplama işlemi doğrudan yapılamaz. Bir çekirdek fonksiyonu çağırılmadan önce ilk yapılacak olan işlem çekirdeğe gönderilecek olan veriler için

GPU belleğinde yer tahsis edilmesi gerekir. Bu işlem Host yani CPU tarafından yapılır.

```
int numElements = 50000;
size_t size = numElements * sizeof(float);

float *h_A = (float *)malloc(size);
float *h_B = (float *)malloc(size);
float *h_C = (float *)malloc(size);

for (int i = 0; i < numElements; ++i)
{
    h_A[i] = rand()/(float)RAND_MAX;
    h_B[i] = rand()/(float)RAND_MAX;
}

float *d_A, *d_B, *d_C;
cudaMalloc((void **)&d_A, size);
cudaMalloc((void **)&d_B, size);
cudaMalloc((void **)&d_C, size);
```

Şekil 2.6. CUDA GPU Belleğinde Yer Tahsis Edilme İşlemi

Şekil 2.6'da 50000 boyutta h_A ve h_B dizilerine rastgele değerler atanmıştır. Bu dizi elemanlarının aynı indis değerine sahip elemanları GPU üzerinde toplama işlemine tabi tutulup sonuçlar h_C dizisine aktarılmak istenirse ilk yapılacak olan işlem GPU belleğinde bu dizilere eş boyutta ayrı ayrı yer tahsis edilmesi gerekir. Şekil 2.6'da d_A, d_B ve d_C dizileri GPU üzerinde h_A, h_B, ve h_C dizilerine eş boyutta tahsis edilmiştir.

Bellek tahsisi yapıldıktan sonra işlem yapılacak verilerin GPU üzerinde çalışabilmesi için GPU belleğine kopyalanması gerekmektedir. Bu işlem de Host tarafından yapılır.

```
float *d_A, *d_B, *d_C;
cudaMalloc((void **)&d_A, size);
cudaMalloc((void **)&d_B, size);
cudaMalloc((void **)&d_C, size);


cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Şekil 2.7. Bellek tahsisinden sonra verilerin GPU belleğine kopyalanması

Şekil 2.7'de GPU belleği üzerinde ayrılan d_A dizisine h_A dizisi verileri, d_B dizisine de h_B dizisi verileri kopyalanır. Burada "size" aktarılacak boyutu, "cudaMemcpyHostToDevice" ise aktarım yönünün Host'tan device'a yani ana bellekten GPU belleğine olduğunu belirtir.

Verilerin kopyalanması işlemi bittikten sonra çekirdek çağrılmadan önce çekirdeği çalıştıracak ızgara yapısı belirlenir.

```
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);


int threadsPerBlock = 256;
int blocksPerGrid =(numElements) / threadsPerBlock;
```

Şekil 2.8.Çekirdeği oluşturacak ızgara yapısının tanımlanması

Şekil 2.8'de ızgara yapısı oluşturulmuştur. Buna göre her bir blokta 256 iş parçacığı, ızgara içerisinde ise 50.000/256 (195,3125) adet blok bulunur. Burada dikkat edilmesi gereken nokta blok ve iş parçacığı sayılarının çarpımı kadar iş parçacığı çalışacak olmasıdır. Bu yüzden ızgara oluşturulurken özellikle çok indisli dizilerde işlem yapılacaksa dizi indisi kadar iş parçacığı oluşturulması gerekir.

Dikkat edilmesi gereken diğer bir nokta; GPU ızgara içerisinde blokları sıraya alır ve her blok içerisindeki iş parçacıklarını ayrı ayrı işletir. Bu işlemi yaparken de iş parçacıkları hesaplama kabiliyetine göre belli sayıda aynı anda işletebilir. Örneğin; hesaplama kabiliyeti 2.0 ve üstü GPU'lar için bir ızgara içinde aynı anda çalışabilecek iş parçacığı sayısı 32 dir. Yani GPU iş parçacıkları 32 şer alabilecek Warp denen yapıya sahiptir. Eğer blok içerisindeki iş parçacığı sayısı Warp sayısının tam katları şeklinde olmazsa Warp tam kapasite dolmayacak ve çekirdek çalışma zamanı uzayacaktır. Bu da zaman kayıplarına sebep olacaktır.

```
int threadsPerBlock = 256;
int blocksPerGrid =(numElements) / threadsPerBlock;

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
```

Şekil 2.9.Çekirdek fonksiyonunun çağırılması

Şekil 2.9'da vectorAddKernel fonksiyonu çağırılmıştır. Çekirdek için tanımlanan ızgara yapısı "<<<" ve ">>>" işaretleri arasına yazılır. GPU belleğine tahsis edilen diziler ve diğer parametreler ise "(" ve ")" işaretleri arasına yazılır.

Çekirdek fonksiyonu ise döngüsüz şekilde dizayn edilmelidir. Çünkü GPU her bir iş parçacığı için çekirdeğin bir kopyasını oluşturup işlem yapacaktır. Burada dikkat edilmesi gereken husus indeks değerinin nasıl alınacağı ve işlemin bir önceki işlemle bir bağlantısının olamayacağıdır. Yani GPU üzerinde birbirine bağlantılı şekilde hesaplama işlemi yapılamaz, her bir iş parçacığı kendi işlemini diğer iş parçacıklarından bağımsız olarak yapar.

```
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}
```

Şekil 2.10.vectorAddKernel fonksiyonu

Şekil 2.10 incelendiğinde bir döngüsel yapının olmadığı görülmüştür. vectorAddKernel fonksiyonu tek boyutta (x ekseni/sütun) hesaplama yapacak şekilde tasarlanmıştır. Burada tanımlanan "i" değeri bloğun x eksen boyutu yani sütun sayısı ile bloğa ait indeks değeri çarpılır ve çekirdeği kullanacak olan iş parçacığına ait indeks değeri toplanarak bulunur. Bu hesaplama tek boyutta oluşturulmuş tüm ızgara yapısı için doğru indeks değerini verir. Örneğin ızgara tek bir bloktan oluşmuş olsun. Blok da 256 iş parçacığından oluşsun. Bu durumda blockDim.x değeri 256 olacak, blockIdx.x değeri 0 olacak ve her bir iş parçacığı da 0-255 arası değer alacağından indeks değeri doğru çıkacaktır. Yine ızgara 2 bloktan oluşsun. Blok da 256 iş parçacığından oluşsun. Birinci blok işleme alındığında ilk örnekteki gibi indeks değerleri oluşacaktır. 2'inci blok işleme alındığında blockDim.x değeri 256 olacak, blockIdx.x değeri 1 olacak, 30. İş parçacığı için indeks değeri (256*1)+30 hesaplanarak 286 bulunur ki bu doğru bir indeks değeridir.

İndeks değeri atandıktan sonra her bir iş parçacığı A ve B "i" inci indeks değerlerini toplayarak C dizisinin "i" inci indeks değerine atar. Burada tüm iş parçacıkları işlemlerini bitirdiğinde verilerin tekrar sistem ana belleğine kopyalanması gerekir.

```
vectorAdd << <blocksPerGrid, threadsPerBlock >> >(d_A, d_B, d_C, numElements);
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

Şekil 2.11. GPU da hesaplanmış verinin ana belleğe kopyalanması

Şekli 2.11'de kopyalama yönetmi cudaMemcpyDeviceToHost yazdığından dolayı d_C dizisi içindeki veriler "size" boyutunda h_C dizisine kopyalanmıştır. İşlemler tamamlandığında hem sistem belleği hemde GPU belleğinde ayrılan alanların temizlenmesi gerekir.

```
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);


free(h_A);
free(h_B);
free(h_C);


cudaDeviceReset();
```

Şekil 2.12. Host ve GPU belleğinin temizlenmesi ve GPU'nun resetlenmesi

Şekil 2.12'de cudaFree() metodu ile GPU üzerinde ayrılan alanlar ayrı ayrı temizlenir. free() metoduyla ana bellekte ayrılmış alanlar ayrı ayrı temizlenir. cudaDeviceReset() metoduyla ise GPU'da kullanılan tüm kaynaklar temizlenir.

2.14. CUDA Programlama API’leri

CUDA programlamada iki adet API (Application Programming Interface/Uygulama Programlama Arayüzü) bulundurur. Birinci API yüksek seviyeli CUDA Runtime API, diğeri ise düşük seviye CUDA Driver API.

CUDA Driver API ile modül başlatma ve bellek yönetimi daha ayrıntılı bir şekilde ele alınır. Detay ve kod daha fazladır. Özellikle çekirdekleri yapılandırmak çok zordur. Çünkü çekirdek parametreleri yürütme yapılandırması söz dizimi yerine açık işlev çağrılarıyla belirtilmelidir. Ancak çekirdek kodu için sürücü JIT (Just In Time/Zamanında) iyileştirmelerini devre dışı bıraktığından daha iyi kontrol ve yönetim yapma imkanı sunar.

CUDA Runtime API ise örtük başlangıç, içerik yönetimi ve modül yönetimi sağlayarak cihaz kodu yönetimini ve kullanımını kolaylaştırır. Runtime API üzerinde GPU belleği tahsisi, belleğin boşaltılması, veri kopyalanması ve sistemin yönetilmesi fonksiyonlarına sahiptir. Örneğin; bir çekirdek çağırılmak istendiğinde “kernel<<<,>>>()” söz dizimi yeterlidir.

Bu iki API birbirini dışlar, yani API’lerden biri tercih edilmelidir. Yeni nesil GPU’larda bu iki API’nin barış içinde olduğu belirtilse de biri tercih edilmelidir. Performans anlamında ise aralarında belirgin bir fark yoktur.

CUDA Runtime API yönetimi kolaylaştıracak iki adet kütüphane sunar. Bunlar CUBLAS ve CUFFT’dir.

2.14.1. CUBLAS

BLAS (Basic Linear Algebra Subprograms/Basit Lineer Cebir Altprogramları) kütüphanesinin CUDA ile birleştirilmesiyle oluşturulmuştur. İşlevleri: gerçek sayılar için 1, 2, ve 3 seviyesinde; karmaşık sayılar için 1 seviyesinde işlev görür. Seviye 1

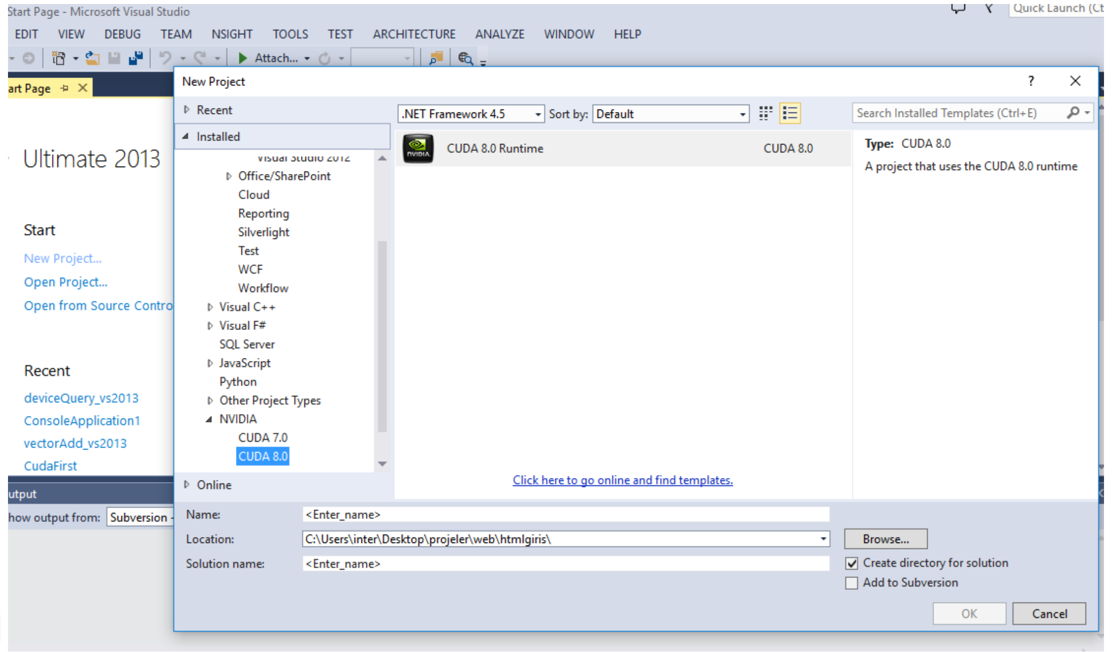
vektör-vektör işlemleri, seviye 2 vektör-matris işlemleri, seviye 3 ise matris-matris işlemleridir [34].

2.14.2. CUFFT

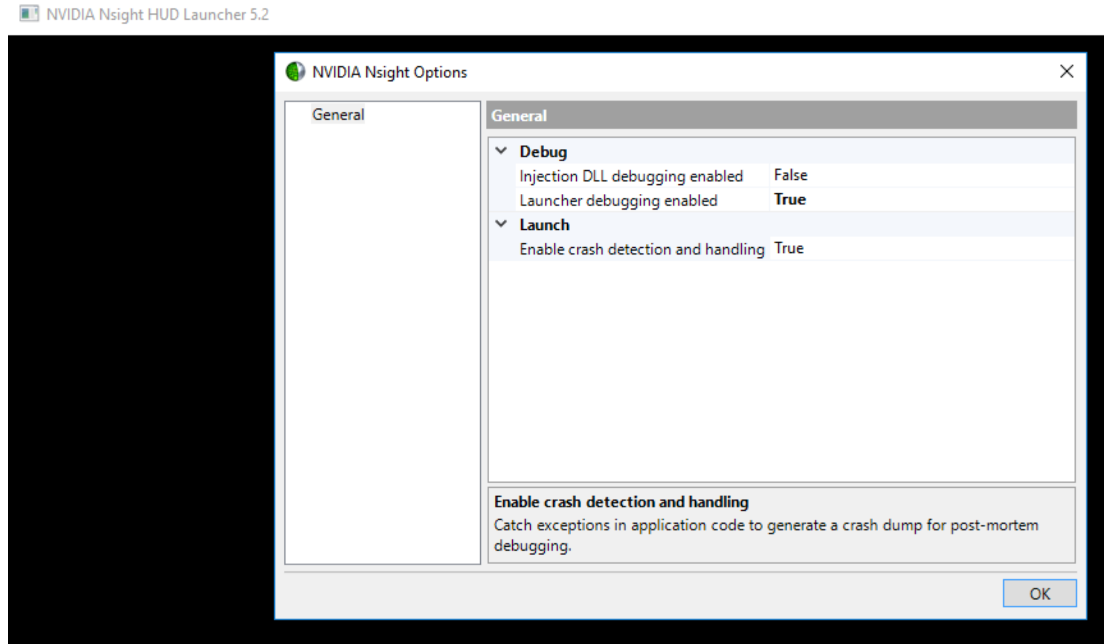
Sinyal analizi, filtreleme ve hızlı Fourier Dönüşümü işlemleri için yaygın kullanım alanı vardır. CUFFT karmaşık ve gerçek verilerin 1B, 2B, ve 3B dönüşümlerini, çeşitli 1B dönüşümlerin toplu yürütülmesini paralel olarak destekler [34].

2.15. CUDA Programlama Nasıl Yapılır?

NVIDIA CUDA ile programlama yapabilmek için C ve C++ dillerine doğrudan destek vermektedir [35]. Ayrıca 3. Parti API'ler ile Phyton, C#, Fortran gibi dillerde de geliştirme yapılabilir. Microsoft Visual Studio C ve C++ ile doğrudan geliştirme seçeneği sunmuştur [36]. Bu işlem için CUDA Runtime bilgisayara kurulu olmalıdır. CUDA Runtime kurulumu başarıyla tamamlandıysa Visual Studio üzerine otomatik eklenti olarak eklenecektir (Şekil 2.13). Ayrıca Visual Studio üzerinde yazılmış olan kod üzerinde debug/hata ayıklama yapabilmek için NVIDIA Parallel NSight HUD Launcher da otomatik kurulmuş olacaktır (Şekil 2.14). Visual Studio üzerinden CUDA debug işlemleri doğrudan yapılamamaktadır.

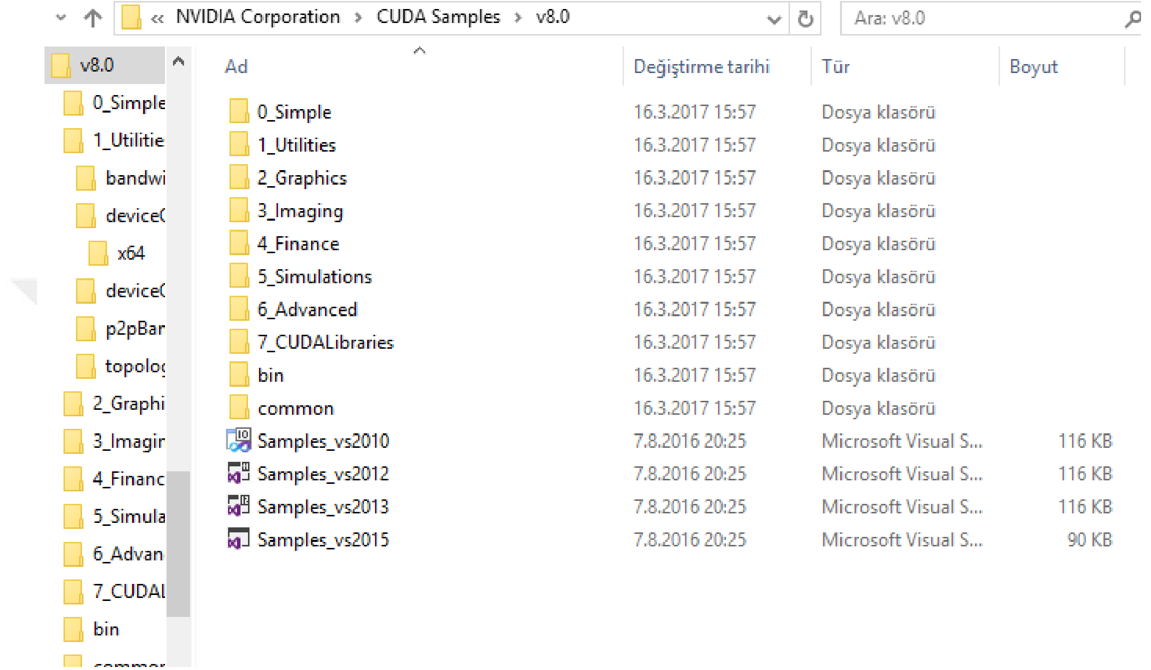


Şekil 2.13. Visual Studio üzerinde yeni bir CUDA Runtime projesi açmak



Şekil 2.14. NVIDIA Nsight HUD Launcher ile Debug Mod ayarları yapmak

CUDA Runtime kurulduktan sonra nasıl kod geliştirileceğini öğrenmek için örnek kodlar kurulum ile birlikte C:\ProgramData\NVIDIA Corporation\CUDA Samples\<CUDA_Runtime_Versiyon_Numarası> altında örnekler eklenmiş olacaktır (Şekil 2.15).



Şekil 2.15. CUDA kod örnekleri

Şekil 2.15'te kod örnekleri eklenmiştir. CUDA Runtime 8.0 ile Visual Studio 2010, 2012, 2013 ve 2015 üzerinde ayrı ayrı derlenmiş kod versiyonları da vardır. Bu kodlar C ve C++ dilleri ile geliştirilmiştir. C# dili ile kullanılmak istendiğinde 3. Parti API'ler yardımıyla kullanılabilir.

2.15.1. ManagedCuda

ManagedCuda C# ile CUDA çekirdeği geliştirmeyi sağlar. ManagedCuda kodlar arasında bir dönüşüm yapmaz. Yani CUDA çekirdeği C dilinde yazılır, GPU

kurulumu, verilerin ana bellekten GPU belleğine gönderilmesi ve tekrar ana belleğe kopyalanması işlemlerini C# ile yürütür. Nuget üzerinden indirilip direkt olarak kullanılabilir. Örnek kod:

```
//Çekirdek Kodu
extern "C"
{
__global__ void VektorEkle(const float* A, const float* B, float* C, int N)
	{
		int i = blockDim.x * blockIdx.x + threadIdx.x;
		if (i < N)
			C[i] = A[i] + B[i];
	}
}


//Host Kodu
int main()
{
	int N = 256;
	int deviceID = 0;
	CudaContext ctx = new CudaContext(deviceID);
	CudaKernel kernel = ctx.LoadKernel("VektorEkle.ptx", "VektorEkle");

	kernel.GridDimensions = 16;
	kernel.BlockDimensions = 16;


	float[] h_A = new float[N];
	float[] h_B = new float[N];


	CudaDeviceVariable<float> d_A = h_A;
	CudaDeviceVariable<float> d_B = h_B;
	CudaDeviceVariable<float> d_C = new CudaDeviceVariable<float>(N);

	kernel.Run(d_A.DevicePointer, d_B.DevicePointer, d_C.DevicePointer, N);


	float[] h_C = d_C;

	return 0;
}
```

2.15.2. CudaFy

C# üzerinden CUDA ile kod geliştirilebilen diğer bir API de CudaFy'dir. ManagedCuda'dan farklı olarak CudaFy'de çekirdek kodu ayrıca geliştirilmez, metod üzerine [CudaFy] özelliği kullanılarak çekirdek metodu yazılabilir. Bu API ile yazılan CUDA çekirdeği derlendiğinde bütün [CudaFy] özelliği bulunan metodlar

için ayrı ayrı C kodunda çekirdek üretir ve kullanır. Bu da CudaFy kullanıcıları için bir avantajdır. Örnek kod:

```
//Çekirdek Kodu
[Cudafy]
public static void VektorEkle(GThread thread, int[] a, int[] b, int[] c)
{
.
      int indeks = thread.blockIdx.x;

      if (indeks < a.Length)
         c[indeks] = a[indeks] + b[indeks];
}


//Host Kodu
static void Main(string[] args)
{
                GPGPU _gpu;

               CudafyModule km = CudafyTranslator.Cudafy();

                _gpu = CudafyHost.GetDevice(eGPUType.Cuda);
                _gpu.LoadModule(km);

                int boyut=256;
                int[] a = new int[boyut];
                int[] b = new int[boyut];
                int[] c = new int[boyut];

                for (int i = 0; i < boyut i++)
                {
                      a[i] = i;
                      b[i] = boyut - i;
                }

                int[] dev_a = _gpu.CopyToDevice(a);
                int[] dev_b = _gpu.CopyToDevice(b);

                int[] dev_c = _gpu.Allocate<int>(c);

                _gpu.Launch(N, 1).VektorEkle(dev_a, dev_b, dev_c);

                _gpu.CopyFromDevice(dev_c, c);

                _gpu.FreeAll();

                Console.ReadKey();

}
```

CUDA ile kod geliştirmeden önce aşağıdaki ilkelerin bilinmesi gerekir:

1. CUDA GPU mimarisi ve çalışma mantığı iyi öğrenilmemişse geliştirilen kod performanslı çalışmayacaktır. Öncelikle GPU ve programlama yapısı iyi öğrenilmelidir.
2. Recursive geliştirilen algoritmalar (Fibonacci, faktöriyel) birbirine bağımlı çalışır. CUDA da ise her iş parçacığı birbirinden bağımsız çalışır. Bu yüzden CUDA da recursive algoritmalar çalıştırılamaz.
3. Hesaplama yapılacak veri seti uzunluğu kadar iş parçacığı oluşturulmalıdır. Daha az veya daha fazla iş parçacığı oluşturulması hata veya zaman kaybına sebep olacaktır.
4. Çekirdek tasarlanırken önce blok içerisindeki iş parçacığı yapısını, sonra ızgara içerisindeki blok yapısını ayarlamak en uygun yöntemdir. Çünkü GPU işlemleri blok bazında işler, yani bir blok'un iş parçacıklarını bitirip diğer bloğa geçer. Blok içerisindeki iş parçacıkları da Warp boyutu kadar aynı anda işleme alır. Warp boyutu hesaplama kabiliyetine göre değişkenlik gösterir. Örneğin 2.0 kabiliyetli bir GPU'da Warp boyutu 32 dir. Bu yüzden blok içerisindeki iş parçacıkları Warp boyutunun katları şeklinde yapılandırmak maksimum performans anlamına gelir.
5. Çekirdek metodunun bir geri dönüş değeri yoktur ve void olmak zorundadır. Yapılan işlemin geri döndürülmesi için geri dönüş değeri bir parametre olarak çekirdek metoduna gönderilmelidir.
6. GPU, CPU ve sistem kaynaklarına erişemez.
7. İş parçacığı senkronizasyonu paylaşımlı bellek haricinde kullanılmaz. Paylaşımlı bellekte kullanılmasının sebebi; paylaşımlı bellek sadece blok içerisinde çok hızlı işlem yapabilmek için bloğa ait iş parçacıkları tarafından kullanılabilir. Yarış durumu ve kilitlenmelerden dolayı iş parçacığı senkronizasyonu gerekir. Diğer bellek işlemlerinde iş parçacığı senkronizasyonu kullanılırsa aşırı bir zaman kaybı söz konusu olur.

2.16. Medikal Görüntü İşleme

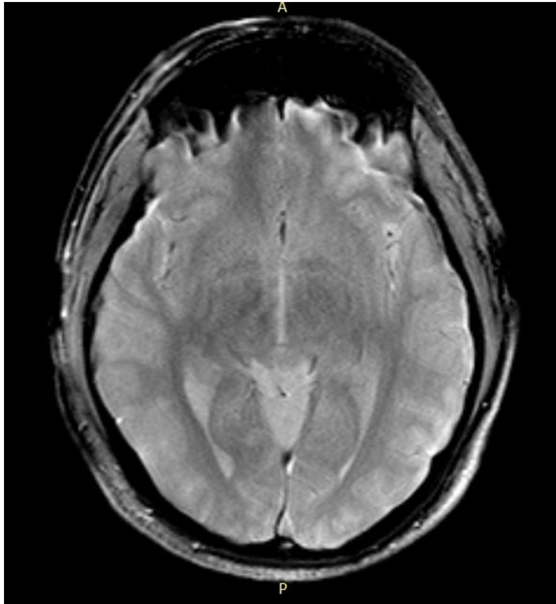
Sağlık hizmetlerinde görüntü işlemenin önemi giderek artmıştır. Erken tanı ve tedavi işlemlerinin hızlı ve etkin yapılmasını sağlar.

Tıbbi Görüntüleme, insan vücudunun içyapısının bazı yöntemler ile görülebilir hale getirilmesidir [37]. Bu tanım içine giren temel cihazlar; Radyografi Cihazları, MR Cihazları, Nükleer Tıp Görüntüleme Sistemleri, Tomografi, Ultrason olarak sıralanabilir. Günümüzde bu cihazlar, tıp, medikal fizik, biyomedikal, elektronik mühendisliği ve bilgisayar mühendisliğinin ortaklaşa çalışması ile üretilmiştir.

2.16.1. MR Görüntüsü

Manyetik Rezonans Görüntüleme yani MR, ağrısız, alerjiye yol açacak ilaç verilme zorunluluğu olmayan ve x-ışını gibi zararlı olabilecek araçlar kullanmayan bir tanı tekniğidir (Şekil 2.16) [38].

MR; Beyin lezyonlarının incelenmesinde, akciğer, bronş ve soluk borusu detaylı incelenmesinde, böbrek, idrar yolları ve mesane incelenmesinde, eklem yerleri ve romatizmal bulgularda, bağırsak incelemelerinde, yumuşak doku görüntüleme ve incelemesinde sıklıkla kullanılır [37].



Şekil 2.16. Beyin MR görüntüsü

2.17. Medikal Görüntülerin Arşivlenmesi

Medikal görüntülerin arşivlenmesi tıbbi dokümantasyonda önemli bir yer tutar. Bilgi çağında özellikle dijital arşivleme daha çok gündemde olan bir konudur. Çünkü medikal görüntülerin filmlere basılması için film maliyeti ve bu filmin arşivlenerek tekrar kullanılmak istendiğinde oluşacak arşiv taraması süresi fiziksel arşivlemenin dezavantajlarındandır.

Günümüz tıbbi görüntüleme cihazları IoT (Internet of Things/Nesnelerin İnterneti) standardında üretilmekte, oluşan medikal görüntünün ağ üzerinden sayısal ortama aktarılabilmesi sağlanmaktadır. Sayısal arşivlemenin avantajları; medikal görüntüyü hasta ile ilişkilendirerek anında ulaşma, görüntü işleme yazılımları ile medikal görüntü üzerinde dinamik analiz ve yorum yapabilme ve kullanıcıyı uyarabilme, medikal görüntüyü sayısal verilerle ilişkilendirerek dinamik olarak istatistik imkanı sunma, sunucu ve depolama alanı dışında fiziksel bir alana ihtiyacın olmaması vs... Sayısal arşivlemenin dezavantajları ise; IoT destekli cihazların bulunması zorunluluğu, ağ altyapısının kurulması, sunucu ve depolama sistemlerinin kurulması, ve görüntü işleme/görüntüleme yazılımları ihtiyaçlarıdır.

Sağlık alanında görüntü işleme sistemleri geliştikçe medikal görüntülerin hasta bilgileri ile birlikte nasıl saklanacağı ve istenildiğinde bu görüntüye verileriyle birlikte nasıl erişileceği konuları, sayısal sistemin önemle üzerinde durduğu konulardır. Bu konuda belli standart çalışmaları sonucunda DICOM veri seti standardı ortaya çıkmıştır [39].

2.18. DICOM Standardı

Arşivlenen medikal görüntünün nasıl kullanılacağı ile ilgili sorulara çözüm bulunması amacıyla geliştirilmiştir. DICOM (Digital Imaging and Communications/Sayısal Görüntüleme ve Haberleşme) dosya yapısı bir veritabanını mantığındadır. Veritabanlarındaki gibi dosya içerisine hem metin veri yazılabilmekte hem de ham görüntü verisi eklenebilmektedir. Tüm verileri tek dosya içerisinde

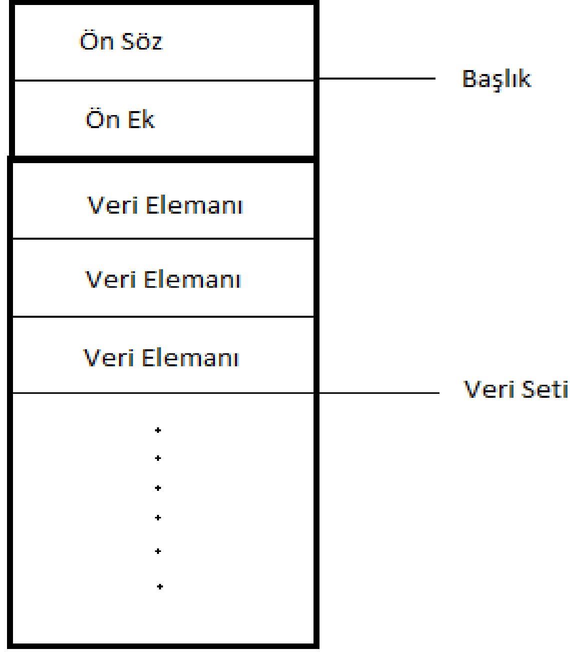
saklayan bu standartta verilerin tekrar elde edilmesinde karmaşa olmaması için etiketler bulunur [39].

Medikal görüntü arşivlemede bir standardın olması farklı bölgelerdeki hastanelerin ve doktorların medikal görüntü üzerinde ortak bir paylaşım yapabilme imkanı sunar. Standart olmaması durumunda medikal görüntülemede farklı cihazlar ve farklı firmalar bulunduğundan dolayı cihazı üreten firmanın yazılımına bağımlılığı gerektirecektir.

DICOM dosya biçimi, bilinen tüm formatlardan farklı olarak medikal görüntüleri, hasta bilgileri, hastane bilgileri ile ilgili daha fazla detay saklayabilen bir formattır. DICOM dosyası içerisinde normal medikal görüntüler ile birlikte hareketli görüntüler ve ses kayıtları da saklanabilir. Ayrıca bir hastaya tüm yapılan testler ve sonuçları ve doktor teşhislerinin ortak bir standart format üzerine kaydedilebiliyor olması bütün hastanelerde tüm doktorlar tarafından hastalığın teşhis ve tedavi süresini önemli ölçüde azaltmaya yardımcı olur.

2.19. DICOM Dosya Yapısı

Bir DICOM dosyası başlık ve veri setinden oluşur (Şekil 2.17). Başlık 128 byte'lık Dosya Önsözü'nden ve ardından 4 baytlık DICOM Öneki'nden oluşur. Dosya önsözü bir açıklama yazmak için ayrılmıştır. Eğer açıklama girilmezse 128 byte'ın tamamı 00H olarak ayarlanacaktır. Dört baytlık DICOM Öneki, büyük harflerle kodlanmış "DICM" karakter dizisini içermelidir [40].



Şekil 2.17. DICOM dosya yapısı

Şekil 2.17'de DICOM dosya yapısı görülmektedir. DICOM veri seti içerisinde bulunan medikal MR görüntüsü 12 veya 16 bit gri tonlamalı olup orijinal boyutu 256*256 pikseldir. Günümüzde yaygın olarak kullanılan ekranlar 8 bit görüntüleme standardında üretildiğinden dolayı, görüntü bit eşleme ile 8 bite dönüştürülerek kullanılır.

2.20. Görüntü Bölütleme

Görüntü bölütleme, görüntünün kendi içerisinde benzer özelliklere sahip diğerlerinden farklı anlamlı bölgelere ayrılmasıdır. Örneğin; bir manzara fotoğrafında ağaçların belirlenmesi, gökyüzünün belirlenmesi, yolun belirlenmesi vb.

Görüntü bölütleme için geliştirilmiş çeşitli yöntemler mevcuttur. Ancak bu yöntemler görüntüden görüntüye değişkenlik gösterir. Her görüntünün dinamik olarak bölütlenmesi çok zor bir işlemdir.

2.21. Görüntü Üzerinde Kenar Belirleme

Görüntünün parlaklıklarındaki belirgin değişiklik olan bölgelere kenar denir. Kenar belirleme işlemi, nesne tanımlama ve özelliklerini belirleme işlemlerinde çok kullanışlıdır [41].

Görüntünün birçok fiziksel özelliği kenar bilgisinden ortaya çıkarılabilir. Konunun önemi çerçevesinde bu bölümde kenar belirleme için çeşitli yöntemler sunulabilir. Bunun için, görüntüye ilişkin ilgilenilen bölgelerin kendi içerisinde yeterince homojen oldukları varsayılmalı ve dolayısıyla iki bölge arasındaki geçiş sadece gri seviye süreksizliklerine dayalı olarak tanımlanabilmelidir [42].

Kenar belirleme aşamasında görüntü içerisinde gürültü varsa, gürültü yüksek frekanslı bileşenlerden oluştuğu için gürültü ile kenarları ayırt etmek zorlaşacaktır. Yine kenar belirleme ve konumlama ölçütleri arasındaki karşılıklı ilişki ve kenarların çok ölçekli yapısı da kenar belirleme aşamasında karşılaşılacak diğer zorluklardan bazılarıdır [42].

2.21.1. Kenar Belirleme Yöntemleri

Güçlü bir kenar belirleyici kenarları iyi sezebilmeli, kenarları doğru konumlarda belirleyebilmeli ve bir kenar için tek bir kenar görüntüsü oluşturabilmelidir [41].

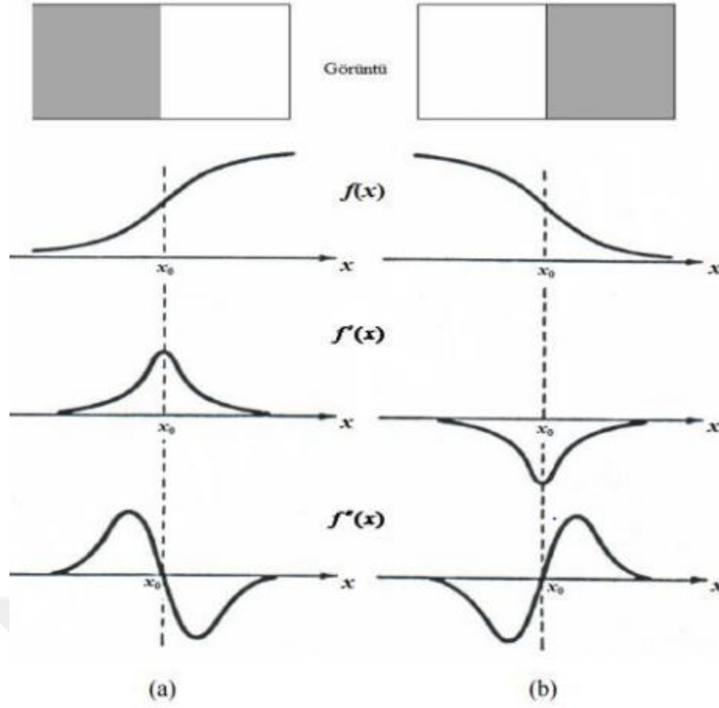
2.21.1.1. Türev Almaya Dayalı Kenar Belirleme Yöntemleri

Bir görüntü içerisindeki kenarları belirlemek için uygulanabilecek iyi yöntemlerden birisi, ani gri seviye değişimlerini tespit etmektir. Bunun için birçok kenar belirleme yönteminin kullandığı temel yaklaşım, bölgesel türev hesabına dayanır. Bölgesel olarak, görüntünün 1.türevi kenar bölgelerinde en büyük değere sahip olur görüntünün 2.türevi ise kenar bölgelerinde sıfır değerini üretir. Bölgesel olarak

görüntüye ilişkin 1. ve 2. türevleri hesaplayarak elde edilen yerel maksimum ve sıfır geçiş noktaları ile, ilgili görüntü bölgesi için kenarlar belirlenmiş olur [43].

2.21.1.1.1. 1.Türeve Dayalı Kenar Belirleme – Gradiyent Yöntemi

Şekil 2.18 (a) ve (b)'deki görüntülere karşı düşen 1B çizimlerde kenar noktaları, ani değişimden çok yumuşak bir geçiş biçiminde görünmektedir. Bu modelleme, gerçeğe yakın bir gösterim için kullanılmıştır. Şöyle ki, örneklemenin bir sonucu olarak sayısal görüntülerdeki kenarlar genel olarak hafif bulanıklaşır. Şekil 2.18'de gösterilen muhtemel kenar noktası x0'ı tanımlamaya ek olarak, $f'(x)$ aynı zamanda kenarın yönünü ve büyüklüğünü kestirmede kullanılabilir. Eğer $|f'(x)|$ çok büyük ise, $f(x)$ çok hızlı değişir ve bu durum parlaklıkta hızlı bir değişime karşı düşer. Eğer $f'(x)$ pozitif ise, $f(x)$ artan bir fonksiyondur [42].



Şekil 2.18. Türev operatörleri ile kenar belirleme: (a) Koyu arka plan üzerindeki beyaz bant görüntüsü; (b) Açık arka plan üzerindeki siyah bant görüntüsü. Bu görüntülere ilişkin 1-B çizimler ve bu çizimlerin 1. ve 2.türevleri [42]

Şekil 2.19. Kenar belirleme yöntemi blok şeması [42]

Şekil 2.19'daki şemaya göre, önce f(x)'den |f'(x)| hesaplanır. |f'(x)|, belli bir eşik değerinden büyükse bu görüntü pikseli, bir kenar adayı olacaktır. İlgilenilen kenar

noktasında eğer bu şart birden fazla x değeri için sağlanırsa bu durumda bir kenar, noktadan çok çizgi olarak görünecektir ve kalın kenarların oluşmasına neden olacaktır. Bu sorunu çözmek için, sadece |f ′(x)| değerlerinde lokal maksimuma sahip olan noktalar bulunur ve bu noktalar kenar noktaları olarak belirlenir.

Bir f(x,y) fonksiyonu için; gradiyent bağıntısı aşağıda açıklanmıştır.

(Gradiyent) $$\nabla f(x,y) := \left(\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \right) \tag{2.1}$$

(Gradiyent Genliği) $$M(x,y) = mag(\Delta f) = \sqrt{g_x^2 + g_y^2} \tag{2.2}$$

(Gradiyent Yönü) $$\alpha(x,y) = \tan^{-1}\left[\frac{g_x}{g_y} \right] \tag{2.3}$$

(Kenarın Yönü) $$\phi = \alpha - 90° \tag{2.4}$$

2.21.1.1.1.1. Sobel Kenar Dedektörü

Çizelge 2.2. Maskelenecek görüntü pikselleri

Z_1	Z_2	Z_3
Z_4	Z_5	Z_6
Z_7	Z_8	Z_9

Çizelge 2.3. Sobe1 kenar dedektörü (a)yatay filtre(b)dikey filtre

−1	−2	−1
0	0	0
1	2	1

(a)

−1	0	1
−1	0	1
−1	0	1

(b)

f(x,y) giriş görüntüsü, T eşik ve g gradiyent olmak üzere her bir piksel için gradiyent Çizelge 2.3 e göre oluşturulursa;

$$G_x=(Z_7 + 2Z_8 + Z_9) - (Z_1 + 2Z_2 + Z_3) \quad (2.5)$$

$$G_y=(Z_3 + 2Z_6 + Z_9) - (Z_1 + 2Z_4 + Z_7) \quad (2.6)$$

$$g=\sqrt[2]{G_x^2 + G_y^2} \quad (2.7)$$

$$g=\sqrt[2]{[(Z_7 + 2Z_8 + Z_9) - (Z_1 + 2Z_2 + Z_3)]^2 + [(Z_3 + 2Z_6 + Z_9) - (Z_1 + 2Z_4 + Z_7)]^2} \quad (2.8)$$

olarak hesaplanır.

g≥T ise f(x,y) kenar pikselidir.

2.21.1.1.1.2. Prewitt Kenar Dedektörü

Prewitt dedektörü Sobel dedektörüne göre basittir ancak gürültülü sonuçlar üretebilir.

Çizelge 2.4. Prewitt kenar dedektörü (a)yatay filtre (b)dikey filtre

−1	**−1**	**−1**
0	**0**	**0**
1	**1**	**1**

(a)

−1	**0**	**1**
−1	**0**	**1**
−1	**0**	**1**

(b)

Prewitt dedektörü ile gradiyent Çizelge 2.4 e göre oluşturulursa;

$$G_x = (Z_7 + Z_8 + Z_9) - (Z_1 + Z_2 + Z_3) \quad (2.9)$$

$$G_y = (Z_3 + Z_6 + Z_9) - (Z_1 + Z_4 + Z_7) \quad (2.10)$$

olarak hesaplanır.

2.21.1.1.1.3. Kirsch Kenar Dedektörü

Kirsch dedektörü örüntü tanımada şablon eşleştirmede kullanılır. Ayrıca kenar yönlerine çok duyarlıdır [44].

Çizelge 2.5. Kirsch kenar dedektörü (a)yatay filtre (b)dikey filtre

5	**5**	**5**
−3	**0**	**−3**
−3	**−3**	**−3**

(a)

5	**−3**	**−3**
5	**0**	**−3**
5	**−3**	**−3**

(b)

Kirsch dedektörü ile gradiyent Çizelge 2.5 e göre oluşturulursa;

$$G_x = (5Z_1 + 5Z_2 + 5Z_3) - (3Z_4 + 3Z_6 + 3Z_7 + 3Z_8 + 3Z_9) \quad (2.11)$$

$$G_y = (5Z_1 + 5Z_4 + 5Z_7) - (3Z_2 + 3Z_3 + 3Z_6 + 3Z_8 + 3Z_9) \quad (2.12)$$

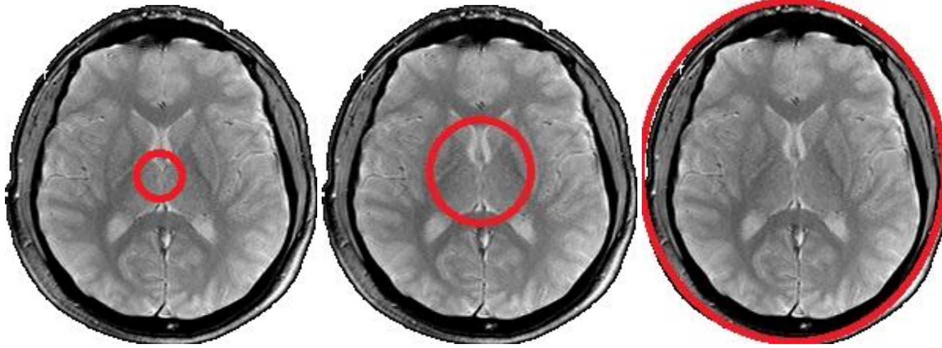
olarak hesaplanır.

2.21.1.2.Aktif Kontur Yöntemi

Aktif kontur model, görüntü üzerinde ilgi alanlarının (ROI) sınırlarına gelindiğinde duran bir modeldir (Şekil 2.20). Bu model yakın kenarlara odaklanan ve bu kenar sınır bölgelerini tespit eden, dış kuvvetler ve görüntü kuvvetleri tarafından yönlendirilen enerjiyi minimize ederek iteratif çalışan bir modeldir [45].

Aktif kontur modeli kullanılrken, ROI bölgesinin iç veya dış kısmına bir eğri çizilir ve eğri, ROI'nin etrafını sarana kadar iterasyonlar ile oluşturulan enerji minimizasyonu sayesinde hareket ettirilir [45].

Aktif kontur metodu, en düşük enerji durumu için otomatik uyum ve gürültüye karşı düşük duyarlılık gibi avantajları barındırır. Bu modelde dış kuvvet görüntü özelliklerinden gelen görüntü gradyan şiddeti gibi değerlere bağlıyken, iç kuvvet ise eğri şeklinin düz olmasını sağlar [45].



Şekil 2.20. Aktif Kontur Metodu iterasyonlar sonucu ROI tespit edilme aşamaları

2.21.1.3.Eşikleme Yöntemi

Eşikleme, görüntüdeki gri seviye dağılımlarından yararlanarak görüntü içindeki objeleri görüntü arkaplanından ayırt etmek veya bölütlemektir. Gri seviye dağılımına

ise histogram adı verilir. Eşikleme ile objeleri arkaplandan ayırt etmek için önce; histogramdan yararlanarak arkaplan ve objeyi ayırt edecek olan eşik değeri (T) belirlenmelidir. Daha sonra her piksel değeri için T değeri ile karşılaştırma yapılır. Örnek olarak; eğer T değeri piksel değerinden büyükse bu piksel arkaplan olarak belirlenir, eğer T değeri piksel değerinden küçük ise bu piksel obje olarak belirlenir. Ayrıca T eşik değeri birden fazla seçilip birden fazla obje tespiti/bölüteme için ayrı ayrı kullanılabilir.

2.21.1.3.1. Global Eşikleme

Bölütleme tek bir T eşik değeri kullanılarak yapılıyorsa bu global eşiklemedir. Sonuç olarak oluşan görüntü siyah ve beyaz ikili (tek bitlik) bir görüntüdür. Global Eşiklemede uygun eşik değeri aşağıdaki algoritmaya [41] göre belirlenebilir:

1. Görüntü histogramı içindeki en küçük ve en büyük gri piksel değerinin ortalamasını T olarak belirle.
2. T'yi kullanarak görüntüyü bölütle. Bu iki B1 ve B2 gibi iki bölge oluştur.
3. B1 ve B2 bölümlerinin ayrı ayrı ortalama gri seviye değerini O1 ve O2 hesapla.
4. O1 ve O2 değerleri ortalanası ile yeni T değerini oluştur.
5. 2. ve 4. adımları, belirli bir Tmin değerine ulaşıncaya kadar iteratif olarak yap.

Global eşiklemenin uygulanacak görüntülerde, aydınlatma önemli bir etkendir. Yani aydınlatmanın kontrol edilebildiği görüntülerde global eşikleme başarıyla kullanılabilir [42].

2.21.1.3.2. Adaptif Eşikleme

Aydınlatmadan kaynaklanan bozulmalar global eşikleme için oldukça büyük problem çıkarır. Adaptif eşikleme, görüntünün küçük bölgelere ayrılarak eşik değerinin bölgesel olarak seçilmesidir. Görüntü içerisindeki piksel değerleri, belirli

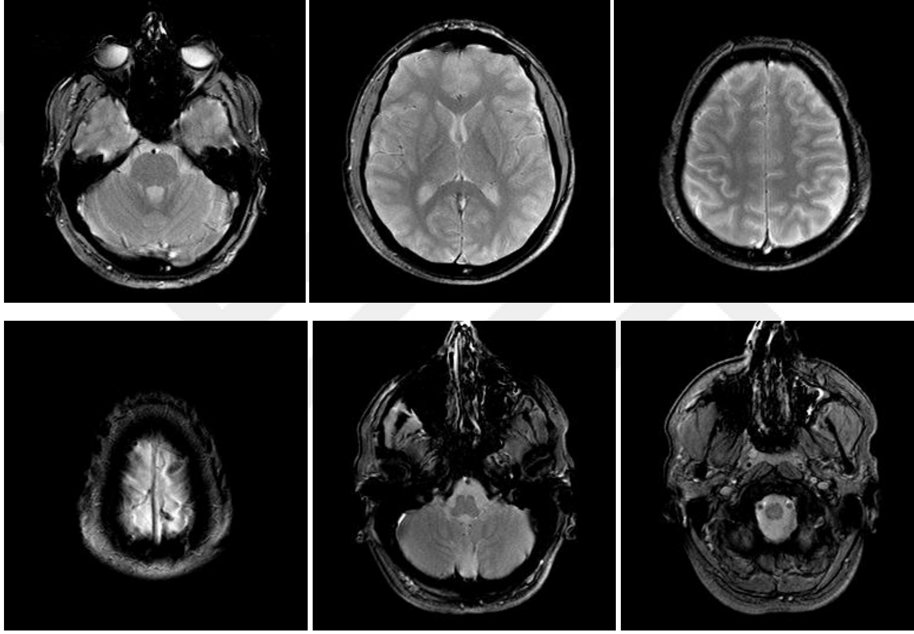
bir bölge içinde değerlendirilir ve eğer bölgenin ortalama gri seviyesi üzerinde ise obje, değilse arkaplan olarak belirlenir.

2.21.1.3.3. Çoklu Eşikleme

3 veya daha fazla dağılıma sahip görüntülerde bölütleme yapmak amacı ile kullanılır. Görüntüdeki birden fazla objeyi tespit edebilmek için kullanılır.

3. ARAŞTIRMA BULGULARI

Medikal görüntüler arşivlendiğinde büyük saklama alanlarına ihtiyaç duyulmaktadır. Ayrıca görüntü boyutu arttıkça medikal sistem üzerinde görüntüye erişilmek istendiğinde ağdaki bant genişliğini daha fazla kullanacak bu da ağda yavaşlıklara yol açacaktır.



Şekil 3.1. Beyin MR Görüntüleri

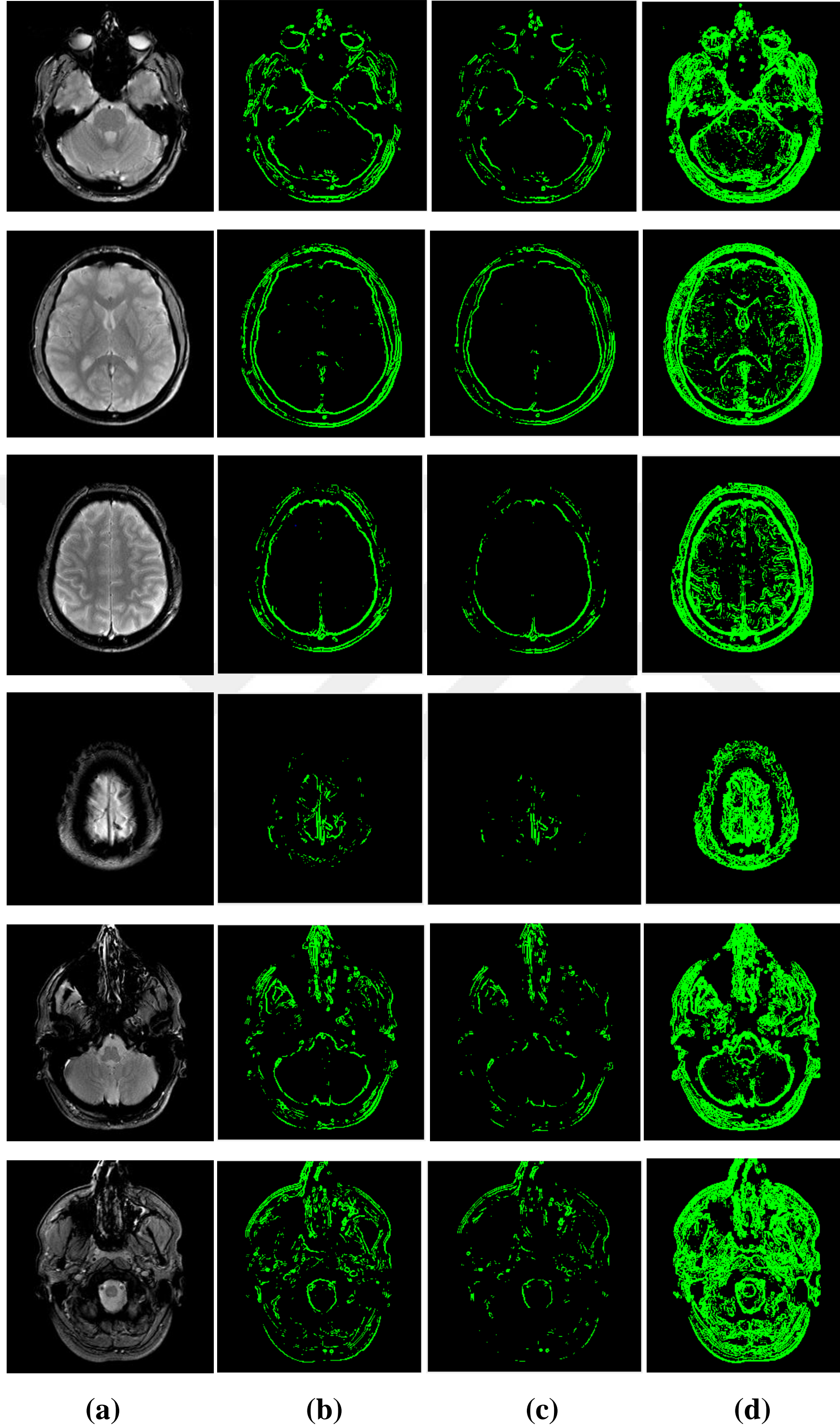
Şekil 3.1 incelendiğinde MR görüntüsü üzerinde ilgilenilecek olan kısım yani ROI (Region of Interest) bölgesi dışında kalan siyah alan yani NON-ROI bölgesi görüntü üzerinde büyük bir alan kaplamaktadır. Bu tez çalışmasında görüntü üzerinde kullanılan ROI bölgesi CUDA ile tespit edilerek, görüntüden çıkarılmış ve bu ROI bölgesi geliştirilen dosya yapısı ile ile sıkıştırılmıştır.

3.1. ROI Bölgesinin Tespit Edilmesi

MR görüntüsü üzerinde ROI bölgesi belirlenirken her piksel için;

1. Türeve dayalı Gradiyent hesaplanır.
    a. En iyi sonuçları almak için bu işleme en uygun Gradiyent matris seçilmelidir.
2. Hesaplanan Gradiyent değeri eşik değeriyle karşılaştırılır ve yerel maksimum durumu kontrol edilir.
    a. Belirlenen eşik değerinden büyükse ve yerel maksimum ise piksel kenar noktası olarak işaretlenir.
    b. Değilse bir sonraki piksele geçilir.

3.2. En Uygun Filtre Matrisinin Seçilmesi



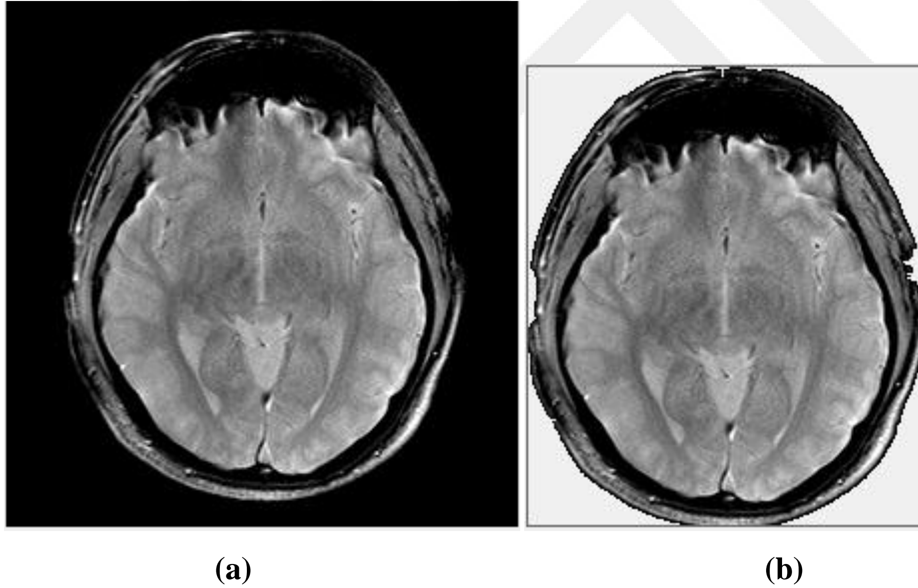
(a) (b) (c) (d)

Şekil 3.2. Beyin MR görüntülerine farklı filtrelerin uygulanması (a)orijinal görüntü (b)Sobel Filtre Matrisi (c)Prewitt Filtre Matrisi (d)Kirsch Filtre Matrisi uygulandıktan sonraki durumlar

Şekil 3.2'de de görüldüğü gibi Sobel, Prewitt ve Kirsch Filtre matrisleri ayrı ayrı kullanılarak Gradiyent işlemi uygulandıktan sonra eşik değeri 255 seçilerek eşikleme yapıldığında Sobel ve Prewitt matrisleri kenar bölgelerini tespit edebilmiş ancak tam anlamda ROI bölgesini belirlemede yetersiz kalmıştır. Kirsch matrisi ise farklı tür MR görüntülerinde kalın kenarlarla (kenarı birden fazla piksel ile ifade ederek) bulmasına rağmen ROI bölgesini yüksek oranda tespit etmede başarılı olmuştur.

3.3. ROI Bölgesinin Görüntüden Çıkarılması

ROI bölgesi tespit edildikten sonra görüntü üzerinde yukardan aşağı, sağdan sola, aşağıdan yukarı ve soldan sağa gezerek ROI bölgesinin en uç koordinatları tespit edilir ve görüntü çerçevesi bu koordinatlara göre daraltılır.



(a) **(b)**

Şekil 3. 3. (a)Orijinal görüntü (b)ROI bölgesi tespit edilmiş ve çerçevesi daraltılmış görüntü

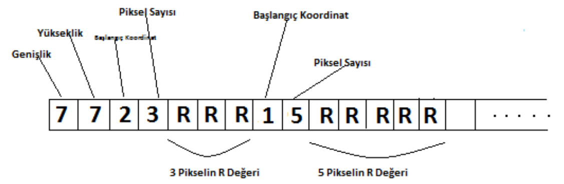
Şekil 3.3 (a)'da orijinal görüntü üzerinden ROI bölgesi tespit edilip çıkarıldıktan ve görüntü çerçevesi daraltıldıktan sonra Şekil 3.3 (b)'deki görüntü elde edilmiştir.

Dikkat edilecek olursa görüntü format olarak dikdörtgen yapıya sahip olduğundan sadece ROI bölgesinin bulunduğu kısım çıkarılsa bile, ROI bölgesi dairesel yapıya sahip olduğundan Şekil 3.3 (b)'de de görüldüğü gibi NON-ROI bölgesinin yine önemli ölçüde yer kapladığı görülmüştür. Bu da görüntü boyutunu artıran bir sorundur. Bu sorunu çözmek için geliştirilen dosya yapısına uygun şekilde NON-ROI bölgesinin Alpha (RGB.A) değeri 0 yapılarak transparan hale getirilir.

3.4. ROI Bölgesini Önerilen Dosya Yapısı İle Saklama

Şekil 3.3 (b)'de de görüldüğü üzere NON-ROI/ROI oranı fazla olduğundan görüntünün Image formatları (Jpeg, Png vb.) ile saklanması çok etkili bir yöntem olmayacaktır. Bu tez çalışması ile sadece ROI bölgesini saklamak için bir dosya yapısı geliştirilmiştir.

0	1	2	3	4	5	6	
		R	R	R			0
	R				R		1
R						R	2
R						R	3
R						R	4
	R					R	5
		R	R	R	R		6



(a) **(b)**

Şekil 3.4. ROI Bölgesini Dizi içerisine geliştirilen dosya yapısı ile saklama (a)7x7 lik görüntü örneği (b)dosya yapısına aktarma işlemi

Şekil 3.4. (a)'da gösterilen bu dosya yapısında görüntü bir matris ile ifade edilirse; Görüntü 8 bit gri tonlu olduğundan ROI bölgesi boyutu ve her satırdaki piksel özellikleri (koordinat ve piksel sayısı) ile RGB (Red Green Blue/Kırmızı Yeşil Mavi) değerinden herhangi biri (R) Şekil 3.4. (b)'deki gibi bir dizide saklanacaktır. Dosya yapısı şu şekilde açıklanabilir:

1. Görüntü genişlik ve yükseklik bilgisi dizinin ilk 2 elemanına yerleştirilir.
2. Görüntü matrisi satır satır dolaşılarak her satırda ROI başlangıç ve bitiş koordinatı elde edilir. Buradan her satırda ROI'ye ait kaç adet piksel bulunduğu hesaplanır. Dizinin sıradaki elemanına önce piksel başlangıç koordinatı yazılır. Sonraki elemana ise piksel sayısı yazılır.
3. Her satırda piksel sayısı kadar R değeri sırasıyla dizinin bir sonraki boş elemanına yerleştirilir.
4. Bir sonraki satıra geçildiğinde başlangıç koordinatı ve piksel sayısı eklenir ve aynı işlemler tüm satırlarda uygulanır.

Yöntemin kaba kodu aşağıdaki gibidir:

```
ArrayListROIdizi = newArrayList();

ROIdizi.Add(Image.Width);
ROIdizi.Add(Image.Height);

for (int i = 0; i <Image.Height; i++)
{
        ROIdizi.Add(PikselBaslangicKoordinati[i]);
        ROIdizi.Add(PikselBitisKoordinati[i]-PikselBaslangicKoordinati[i]+1);

        for (int h = 0; h <Image.Width; h++)
        {
                if (h >= PikselBaslangicKoordinati[i]&& h <= PikselBitisKoordinati[i])
        }

        ROIdizi.Add(pixelDegeri.R);
}
```

3.5. Performans Testleri

Bu tez çalışması ile yapılan tüm işlemler hem CPU (Sequential/Seri) üzerinde hem de CUDA ile 2 farklı Grafik Kartı GPU'su üzerinde ayrı ayrı uygulanmıştır. CPU

kodları C# dili ile yazılmıştır. GPU kodları ise C dili altyapısı üzerine C# dili Cudafy kütüphanesi kullanılarak yapılmıştır. Cudafy kütüphanesi ile yazılan çekirdek kodları ".cu" uzantılı bir Cuda Runtime C dili koduna dönüştürülür. Sonra kodlar NVIDIA NVCC derleyicisine gönderilir ve NVCC derleyicisi kodu GPU'nun yorumlayacağı alt seviye makine diline dönüştürür ve kod GPU tarafından işlenir.

Kullanılan CPU özellikleri:

- Intel Core I7 2.20-2.20 GHz İşlemci
- 8GB RAM Bellek
- 8 Çekirdek

1. GPU Özellikleri:

- NVIDIA NVS 4200
- 48 Çekirdek
- Ayrılmış Bellek 1GB
- CUDA Hesaplama Kabiliyeti 2.1 Versiyonu

2. GPU Özellikleri:

- NVIDIA Quadro K620
- 384 Çekirdek
- Ayrılmış Bellek 2GB
- CUDA Hesaplama Kabiliyeti 5.0 Versiyonu

Çizelge 3.1. MR görüntülerinin orijinal, ROI ve geliştirilen dosya yapısı boyutları

Görüntü	Orijinal Görüntü Boyutu (Bitmap)	Orijinal Görüntü Boyutu (Jpeg)	ROI Görüntü Dizi Boyutu	Geliştirilen Dosya Yapısı Boyutu
1	256 KB	35,4 KB	177KB	33 KB
2	256 KB	31,8 KB	178KB	24KB
3	256 KB	34,2 KB	187KB	35KB
4	256 KB	33,1 KB	151KB	30KB
5	256 KB	25,4 KB	115KB	22KB
6	256 KB	16,3 KB	81KB	15KB
Ortalama	**256 KB**	**29,3 KB**	**148KB**	**26KB**

Çizelge 3.1 de, Şekil 3.2'deki 6 MR görüntüsünün orijinal Bitmap ve Jpeg formatında veri boyutları, yapılan çalışmalar ile elde edilen ROI görüntüsü ve geliştirilen dosya yapısı ile elde edilen veri boyutları gösterilmektedir. Uygulamanın ilk aşamasında oluşan ROI görüntü boyutu orijinal Bitmap formatında görüntü boyutundan %43 daha azdır. ROI görüntüsü dikdörtgen olduğundan dolayı görüntü içerisinde kullanılmayacak ancak önemli ölçüde NON-ROI kısmı bulunur. Bunun için bir dosya yapısı geliştirilmiştir. Geliştirilen dosya yapısı ile oluşan veri boyutu ise orijinal MR görüntüsü boyutundan %90 daha azdır. Jpeg görüntü formatı ise sıkıştırılmış bir format olduğundan uygulama sonrası oluşan veri boyutlarını kıyaslayabilmek için bu veriler üzerinde sıkıştırma işlemi yapılması gerekmektedir.

Çizelge 3.2. Orijinal, ROI ve geliştirilen dosya yapısındaki verilerin sıkıştırılmış boyutları

Görüntü	**Orijinal Görüntü Boyutu (Jpeg)**	**Sıkıştırılmış Orijinal Veri Boyutu**	**Sıkıştırılmış ROI Veri Boyutu**	**Sıkıştırılmış Geliştirilen Dosya Yapısı Boyutu**
1	35,4 KB	46 KB	43KB	27KB
2	31,8 KB	37 KB	35KB	23KB
3	34,2 KB	44KB	42KB	29KB
4	33,1 KB	37 KB	35KB	25KB
5	25,4 KB	30KB	28KB	19KB
6	16,3 KB	22KB	20KB	12KB
Ortalama	**29,3 KB**	**36KB**	**30KB**	**22KB**

Çizelge 3.2 de, Şekil 3.2'de bulunan 6 MR görüntüsünün orijinal veri ve uygulama sonrası oluşan ROI görüntüsü ve sonrasında geliştirilen dosya yapısı ile oluşan verilerin bir sıkıştırma algoritması ile ayrı ayrı sıkıştırıldıktan sonra oluşan veri boyutları gösterilmektedir. Orijinal sıkıştırılmış görüntü ile ROI sıkıştırılmış veri boyutu orijinal görüntü boyutundan ortalama %15 daha azdır. Geliştirilen dosya yapısı ile oluşan veri sıkıştırıldığında, her bir görüntüye ait işlem için orijinal veri boyutuna göre ortalama %92, orijinal verinin sıkıştırılmış boyutuna göre ise ortalama %40 daha azdır. Ayrıca geliştirilen dosya yapısı ile elde edilmiş veri sıkıştırıldığında, veri boyutu her bir görüntüye ait işlem için Çizelge 3.1 deki Jpeg formatı boyutuna göre daha düşüktür ve boyutu Jpeg formatından ortalama %25 daha azdır.

Çizelge 3.3. CPU üzerinde çalışan seri kod ile CUDA ile kodlanmış GPU kodunun çalışma zamanı karşılaştırması

Görüntü	Seri CPU Kodu Çalışma Zamanı	CUDA NVS4200 Çalışma Zamanı	CUDA K620 Çalışma Zamanı
1	325 ms	9 ms	4 ms
2	315 ms	10 ms	5 ms
3	314 ms	12 ms	6 ms
4	318 ms	9 ms	4 ms
5	354 ms	13 ms	7 ms
6	436 ms	8 ms	4 ms
Ortalama	**343 ms**	**10 ms**	**5 ms**

Çizelge 3.3, Şekil 3.2’de bulunan 6 MR görüntüsü üzerinde yapılan ROI görüntüsünün belirlenmesi ve sonrasında geliştirilen dosya yapısısına aktarma işlemlerinin CPU ve CUDA ile GPU üzerinde çalıştırılan seri ve paralel kodlarının çalışma zamanlarını göstermektedir. CUDA ile yazılan paralel GPU kodu, CPU üzerinde yazılmış olan seri koda göre her görüntüde önemli bir başarı sağlamış ve NVS4200 ile ortalama 34 kat, K620 ile 68 kat daha hızlı işlem yapmıştır.

4. TARTIŞMA VE SONUÇ

Saniyelerin hatta saliselerin bile önemli olduğu sağlık alanında işlemleri çok kısa sürede bitirilmesi hayati önem taşır. Bunun için ise bütün sağlık sistemlerinin aksamadan, etkin ve hızlı biçimde çalışması gerekir. Ayrıca bu zaman aralığında çok hızlı karar almak da çok önemlidir. Bu karar verme sürecini hızlandırmaya yardımcı olabilecek en büyük sistem, sayısal bir sistemdir.

Sağlık alanında bulunan bir sayısal sistemde en çok önem arz eden şey hastanın bilgilerine, geçmişte geçirdiği sağlık olaylarına ve bu olayların tespiti için gerekli ölçümlere hızlı bir şekilde ulaşmak; bir sağlık sorununda ise hastalığın bulgularını ölçüp bu ölçüm sonuçlarını hızlı bir şekilde değerlendirerek bu sağlık sorununu en hızlı bir biçimde çözmektir. Bu ölçümlerden biri ise görüntüleme sistemiyle yapılan ölçümlerdir. Görüntü işlemenin sağlık alanının en büyük ve önemli bir bölümü olan radyoloji sisteminde kullanılması iç hastalıklarının tespitinde önemli bir rol oynar.

Bu tez çalışması ile MR görüntülerinin sayısal sistemde daha az bir bellek alanı kullanarak saklanması ve bu görüntülere daha hızlı bir şekilde ulaşılmasını sağlayacak yöntemler uygulanmıştır. MR görüntüsünün ilgilenilen alanı (ROI) dışında kalan (NON-ROI) bölgenin alanının önemli ölçüde fazla olmasından dolayı sadece ROI alanının saklanması daha mantıklı olacağı düşünülmüştür. Buradan yola çıkarak ROI bölgesi tespit edilip görüntüden çıkarılmış, ancak görüntü dikdörtgen bir yapıda olduğundan bu görüntü üzerinde halen NON-ROI bölgesi bulunması ve bu bölgenin de önemli bir alan kaplaması sonucu bir dosya yapısı geliştirilerek dikdörtgen görüntü yapısı yerine sadece ROI verilerinin bulunduğu bir dizi tasarlanmıştır. Bu diziye ROI verilerinin kayıpsız bir şekilde tüm verilerinin ve indeks bilgilerinin geliştirilen dosya yapısına aktarılması sağlanmıştır.

Yapılan literatür taramalarında, MR görüntülerinin sıkıştırılarak elde edilen en iyi başarı %92.12 ile %97.84 arasındadır. MR görüntülerinin GPU ile yapılan uygulamalarında CPU'ya göre en iyi sonuç 55 kat olarak tespit edilmiştir.

Geliştirilen bu dosya yapısı ile oluşan veri boyutu orijinal görüntünün ortalama % 8'i kadardır. Her bir MR görüntüsü için %92'lik bir alan tasarrufu, hastanelerde kayıtlı milyonlarca MR görüntüsünün bulunduğu düşünüldüğünde ciddi bir veri alanı tasarrufu demektir. Bu oran literatürdeki en iyi sonuca %0.01 ila %6 arasında yaklaşık bir sonuç olarak belirlenmiştir.

Verinin arşiv sisteminde sıkıştırılarak saklanması ise ikinci bir olası durumdur. Geliştirilen dosya yapısı ile oluşan veri sıkıştırıldığında oluşan veri boyutu orijinal görüntünün sıkıştırıldığında oluşan verinin ortalama % 60'ı boyutundadır. Bu durumda da her bir MR görüntüsü için ortalama %40'lık bir alan tasarrufu sağlanmış olur. Yine geliştirilen bu dosya yapısı ile oluşan veri sıkıştırıldığında orijinal MR görüntüsünün Jpeg formatındaki boyutunun ortalama %75'i kadardır. Bu %25'lik fark da geliştirilen dosya yapısının sıkıştırılmış bir format olan Jpeg görüntü formatına göre başarılı olduğunu gösterir.

Uygulama hem CPU üzerinde hem de CUDA ile GPU üzerinde ayrı ayrı kodlanarak çalıştırılmış ve çalışma zamanları incelenmiştir. Buna göre; CUDA ile yazılmış paralel uygulamanın, CPU üzerinde yazılmış uygulamadan NVS4200 ile 34 kat, K620 ile 68 kat daha hızlı çalıştığı görülmüştür. Bu çalışma ve literatürdeki çalışmalar da GPU sistemlerinin görüntü işleme alanında CPU performansına göre oldukça yüksek oranda hızlı olduğu görülmektedir. Bu çalışmada kullanılan CUDA ile kodlanmış paralel GPU uygulaması, literatürdeki MR görüntüsü GPU uygulamaları performans ortalamalarının üzerinde bir sonuç elde etmiştir.

Bu çalışmada geçmişten günümüze kadar ilerlemiş GPU programlama uygulamaları incelenmiş, bu çalışmaların devamı niteliğinde CUDA ile MR görüntüleri üzerinde yeni bir dosya yapısı metodu geliştirilmiş ve başarılı sonuçlara ulaşılmıştır. Gelecek çalışmalarda da bu yöntemlere yenileri eklenmeye devam edilecektir.

KAYNAKÇA

[1] Yıldız, E., NVIDIA CUDA İle Yüksek Performanslı Görüntü İşleme, Yüksek Lisans Tezi, İstanbul Üniversitesi Fen Bilimleri Enstitüsü, İstanbul, 2011.

[2] NVIDIA CUDA C Programming Guide 3.1, http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf, (Erişim Tarihi: 25.11.2017).

[3] Podlozhnyuk, V., Image Convolution with CUDA , http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf, (Erişim Tarihi: 25.11.2017).

[4] Podlozhnyuk, V., FFT based 2D Convolution, http://developer.download.nvidia.com/compute/cuda/2_2/sdk/website/projects/c onvolutionFFT2D/doc/convolutionFFT2D.pdf, (Erişim Tarihi: 27.11.2017).

[5] Garland, M., Le Grand, S., Nickolls, J., Anderson, J.,Hardwick, J., Morton, S., Phillips, E., Zhang, Y. ve Volkov, V., Parallel Computing Experiences with CUDA, Micro, IEEE, 28(4), s13-27, 2008.

[6] Yang, Z., Zhu, Y. ve Yong, P., 2008, Parallel Image Processing Based on CUDA, International Conference on Computer Scienceand Software Engineering, s198-201, 2008.

[7] Huang, J.,Ponce, S.P., Park, S.I, Yong, C. ve Querk, F., GPU Accelerated Computation for Robust Motion Tracking Using the CUDA Framework, 5th International Conference on Visual Information Engineering, s437-442, 2008.

[8] Cabido, R.,Concha, D., Pantrigo, J.J. ve Montemayor, A.S., High Speed Articulated Object Tracking using GPUs: A Particle Filter Approach, 10th

International Symposium on PervasiveSystems, Algorithms, and Networks, s757-762, 2009.

[9] Erguzen, A. ve Erdal, E., Medical Image Archiving System Implementation with Lossless Region of Interest and Optical Character Recognition, Journal of Medical Imaging and Health Informatics, Volume 7, Number 6, s1246-1252(7), 2017.

[10] Pan L.,Gu L. ve Jianrong X., Implementation of Medical Image Segmentation in CUDA, International Conference on Information Technology and Applications in Biomedicine, ITAB, ISBN: 978-1-4244-2254-8, 2008.

[11] Eklund A.,Dufirt P., Forsberg D. ve Laconte S.M.,Medical Image Processing on the GPU – Past, Presentand Future, Elseiver, Medical Image Analysis, Volume 17, Issue 8, s1073-1094, 2013.

[12] Owens, J., D., Ozcelik, P.M., Xia, J. ve Samant, S.S., Implementation of Medical Image Segmentation in CUDA, International Conference on Computational Sciences and Its Applications, ICCSA, ISBN: 978-0-7695-3243-1, 2008.

[13] Shebab, M.A., Al-Ayyoub, M. ve Jararweh, Y., Improving FCM and T2FCM Algorithms Performance Using GPUs for Medical Images Segmentation, 6th International Conference on Information and Communication Systems (ICICS), ISBN: 978-1-4799-7349-1, 2015.

[14] Harish P. ve Narayanan P. J., Accelerating Large Graph Algorithms on the GPU Using CUDA, International Conference on High-Performance Computing, 2007, s197-208.

[15] Manavski S.A. ve Valle G., CUDA Compatible GPU Cards As Efficient Hardware Accelerators For Smith-Waterman Sequence Alignment, Italian Society of Bioinformatics (BITS), V 9(Suppl 2):S10, 2008.

[16] Bell N. ve Garland M., Efficient Sparse Matrix-Vector Multiplication on CUDA, NVIDIA Technical Report NVR-2008-004, 2008.

[17] Lin. C.Y., Lee. W.S., Tang. Y.T., Parallel Shellsort Algorithm For Many-Core GPUs With CUDA, International Journal of Grid and High Performance Computing (IJGHPC) 4(2), s1-16, 2012.

[18] Cuda Nedir?, http://www.nvidia.com.tr/object/cuda-parallel-computing-tr.html, (Erişim Tarihi: 09.12.2017).

[19] OpenCL vs. CUDA: Which Has Better Application Support?, https://create.pro/blog/opencl-vs-cuda/, (Erişim Tarihi: 10.12.2017).

[20] OpenCL vs CUDA vs CPU, https://forums.adobe.com/thread/1894349, (Erişim Tarihi: 10.12.2017)

[21] GPU ve CPU Mimarisi Arasındaki Farklar, http://ugurkontel.net/cuda-cekirdegi-nedir/, (Erişim Tarihi: 14.12.2017)

[22] Sözen, N., CUDA Mimarisi, http://nezihesozen.github.io/mydoc/cuda6.html, (Erişim Tarihi: 15.12.2017).

[23] Gahagan M., Tesla mimarisi, https://cseweb.ucsd.edu/classes/fa12/cse141/pdf/09/GPU_Gahagan_FA12.pdf, (Erişim Tarihi: 17.12.2017).

[24] Neden Tesla?, http://www.nvidia.com.tr/object/why-choose-tesla-tr.html, (Erişim Tarihi: 18.12.2017).

[25] GPU Tesla Hesaplama Uygulamaları, http://www.nvidia.com.tr/object/gpu-applications-tr.html, (Erişim Tarihi: 22.12.2017).

[26] Baskaran S., Fermi Mimarisi, https://www.linkedin.com/pulse/nvidia-fermi-vs-kepler-maxwell-pascal-gpus-sangeetha-baskaran, (Erişim Tarihi: 28.12.2017).

[27] Fermi Mimarisi, http://www.nvidia.com.tr/object/quadro-fermi-highlights-tr.html, (Erişim Tarihi: 02.01.2018).

[28] Fermi GPU Uygulamaları, http://www.nvidia.com.tr/object/quadro-graphics-for-energy-exploration-tr.html, (Erişim Tarihi: 02.01.2018).

[29] Kepler Mimarisi, http://www.nvidia.com.tr/object/nvidia-kepler-tr.html, (Erişim Tarihi: 04.01.2018).

[30] Maxwell Mimarisi, http://www.nvidia.com.tr/object/maxwell-gpu-architecture-tr.html, (Erişim Tarihi: 07.01.2018).

[31] Sözen, N., Grid Block Thread Yapısı, http://nezihesozen.github.io/mydoc/cuda2.html, (Erişim Tarihi: 10.01.2018).

[32] Sözen, N., CUDA Hafıza Organizasyonu, http://nezihesozen.github.io/mydoc/cuda3.html, (Erişim Tarihi: 11.01.2018).

[33] Sözen, N., Hafıza Ayırma Fonksiyonları, http://nezihesozen.github.io/mydoc/cuda5.html, (Erişim Tarihi: 14.01.2018).

[34] CUDA Libraries, http://ixbtlabs.com/articles3/video/cuda-1-p4.html, (Erişim Tarihi: 17.01.2018).

[35] CUDA ile Paralel Hesaplama, http://www.nvidia.com.tr/object/cuda-parallel-computing-tr.html, (Erişim Tarihi: 20.01.2018).

[36] CUDA Installation Guide for Microsoft Windows, http://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html, (Erişim Tarihi: 22.01.2018).

[37] Tıbbi Görüntüleme, http://www.medikalteknik.com.tr/saglik-hizmetlerinde-medikal-goruntuleme/, (Erişim Tarihi: 25.01.2018).

[38] MR Nedir, http://www.akcaabatdh.gov.tr/detay.php?id=598&cid=156, (Erişim Tarihi:25.01.2018).

[39] Ulaş, M. ve Boyacı, A., PACS ve Medikal Görüntülerin Sayısal Olarak Arşivlenmesi, Akademik Bilişim'07 - IX. Akademik Bilişim Konferansı Bildirileri, Dumlupınar Üniversitesi, Kütahya, s69-74,2007.

[40] Temel DICOM Dosya Yapısı, https://www.leadtools.com/sdk/medical/dicom-spec1, (Erişim Tarihi: 28.01.2018).

[41] Aydın, İ., Kenar Belirleme Yöntemleri, http://web.firat.edu.tr/iaydin/bmu357/bmu_357_bolum7.pdf, (Erişim Tarihi:02.02.2018) .

[42] Kızılkaya, A., Kenar Belirleme Yöntemleri, http://akizilkaya.pamukkale.edu.tr/Bölüm4_goruntu_isleme.pdf, (Erişim Tarihi: 02.02.2018).

[43] Yılmaz, İ., Görüntü İşleme Teknikleri Kullanılarak Renkli Obje Takibi Yapma, Yüksek Lisans Tezi, Karabük Üniversitesi, Karabük, 2016.

[44] Güllü, M.K., Kirsch Operatörü, http://www.yalova.edu.tr/Files/UserFiles/70/aliiskurt/IMAGE_PROCESS/kulis_ilk_konular.pdf, (Erişim Tarihi: 05.02.2018).

[45] Işıkçı, E. ve Duru, D.G., Multipl Skleroz Manyetik Rezonans Görüntülerinde Aktif Kontur Modeli ile Lezyon Tespiti, Tıp Teknolojileri Ulusal Kongresi, s122-125, 2015.

EKLER

EK-1. Uygulamanın C# ile yazılmış CUDA Çekirdek kodları

```
//Bu Method DICOM  içerisindeki 16 bit görüntüyü  8 bit formata dönüştüren CUDA Çekirdeğidir
[Cudafy]
public static void UshortToByte(GThread thread, byte[] pixelData, double level,ushort pixelRepresentation,
double slope, double intercept, double window, byte[] outPixelData, ushort mask, double maxval)
{
        int i = (thread.blockDim.x * thread.blockIdx.x + thread.threadIdx.x) * 2;

        int index = ((i / 2) * 4);

        ushort gray=(ushort)((ushort)(pixelData[i])+(ushort)(pixelData[i + 1] << 8));

        double valgray = gray & mask;

        if (pixelRepresentation == 1)
        {
           if (valgray > (maxval / 2))
              valgray = (valgray - maxval);

        }

        valgray = slope * valgray + intercept;


        double half = ((window - 1) / 2.0) - 0.5;

        if (valgray <= level - half)
           valgray = 0;
        else if (valgray >= level + half)
           valgray = 255;
        else
           valgray =((valgray -(level - 0.5)) / (window - 1) + 0.5) *255;

        outPixelData[index] = (byte)valgray;
        outPixelData[index + 1] = (byte)valgray;
        outPixelData[index + 2] = (byte)valgray;
        outPixelData[index + 3] = 255;

    }
```

```
//Görüntüye Kircsh filtresi ve gradiyent methodu  uygulanması

[Cudafy]
public static void update_bitmap(GThread thread, int stride, int width, int height, byte[] input, byte[]
output, double[,] xFilterMatrix, double[,] yFilterMatrix)
    {
        int esik = 255;

        double blueX = 0.0;
        double greenX = 0.0;
        double redX = 0.0;

        double blueY = 0.0;
        double greenY = 0.0;
        double redY = 0.0;

        double blueTotal = 0.0;
        double greenTotal = 0.0;
        double redTotal = 0.0;

        int filterOffset = 1;
        int calcOffset = 0;

        int byteOffset = 0;

        int offsetX = thread.blockDim.x * thread.blockIdx.x + thread.threadIdx.x;
        int offsetY = thread.blockDim.y * thread.blockIdx.y + thread.threadIdx.y;

        blueX = greenX = redX = 0;
        blueY = greenY = redY = 0;

        blueTotal = greenTotal = redTotal = 0.0;

        byteOffset = offsetY *
                stride +
                offsetX * 4;

        for (int filterY = -filterOffset;
          filterY <= filterOffset; filterY++)
        {
          for (int filterX = -filterOffset;
            filterX <= filterOffset; filterX++)
          {
            calcOffset = byteOffset +
                  (filterX * 4) +
                  (filterY * stride);

            blueX += (double)(input[calcOffset]) *
                  xFilterMatrix[filterY + filterOffset,
                      filterX + filterOffset];

            greenX += (double)(input[calcOffset + 1]) *
                  xFilterMatrix[filterY + filterOffset,
                      filterX + filterOffset];

            redX += (double)(input[calcOffset + 2]) *
                  xFilterMatrix[filterY + filterOffset,
                      filterX + filterOffset];

            blueY += (double)(input[calcOffset]) *
                  yFilterMatrix[filterY + filterOffset,
```

```
                    filterX + filterOffset];

            greenY += (double)(input[calcOffset + 1]) *
                      yFilterMatrix[filterY + filterOffset,
                          filterX + filterOffset];

            redY += (double)(input[calcOffset + 2]) *
                      yFilterMatrix[filterY + filterOffset,
                          filterX + filterOffset];
        }
    }

    blueTotal = 0;
    greenTotal = Math.Sqrt((greenX * greenX) + (greenY * greenY));
    redTotal = 0;

    if (blueTotal > 255)
    { blueTotal = 255; }
    else if (blueTotal < 0)
    { blueTotal = 0; }

    if (greenTotal > 255)
    { greenTotal = 255; }
    else if (greenTotal < 0)
    { greenTotal = 0; }


    if (greenTotal < (esik))
    {
        greenTotal = 0;
    }
    else
    {
        greenTotal = 255;
    }



    if (redTotal > 255)
    { redTotal = 255; }
    else if (redTotal < 0)
    { redTotal = 0; }

    output[byteOffset] = (byte)(blueTotal);
    output[byteOffset + 1] = (byte)(greenTotal);
    output[byteOffset + 2] = (byte)(redTotal);
    output[byteOffset + 3] = 255;
}
```

```
//Tespit edilen NON-ROI bölgesinin çıkarılması

[Cudafy]
public static void KirpYA(GThread thread, int stride, int height, byte[] input, byte[] output_ragba, int[]
xmin, int[] ymin)
    {
      int i = thread.blockDim.x * thread.blockIdx.x + thread.threadIdx.x;
      int x = 0, y = 1;

      ymin[x] = height;
      ymin[y] = height;
      xmin[x] = height;
      xmin[y] = height;

      for (int j = 0; j < height; j++)
      {

        int byteOffset = j * stride + i * 4;

        if (output_ragba[byteOffset + 1] == 255)
        {
          if (xmin[x] > i)
          {
            xmin[x] = i;
            xmin[y] = j;
          }
          if (ymin[y] > j)
          {
            ymin[x] = i;
            ymin[y] = j;
          }
          break;
        }
        else
        {
          input[byteOffset + 3] = 0;
        }
      }

    }



[Cudafy]
public static void KirpAY(GThread thread, int stride, int width, int height, byte[] input, byte[]
output_ragba, int[] xmax, int[] ymax)
    {
      int i = width - (thread.blockDim.x * thread.blockIdx.x + thread.threadIdx.x) - 1;
      int x = 0, y = 1;

      ymax[x] = 0;
      ymax[y] = 0;
      xmax[x] = 0;
      xmax[y] = 0;

      for (int j = height - 1; j > 0; j--)
      {
        int byteOffset = j * stride + i * 4;

        if (output_ragba[byteOffset + 1] == 255)
        {
```

```
                if (xmax[x] < i)
                {
                    xmax[x] = i;
                    xmax[y] = j;
                }
                if (ymax[y] < j)
                {
                    ymax[x] = i;
                    ymax[y] = j;
                }

                break;
            }
            else
            {
                input[byteOffset + 3] = 0;
            }
        }

    }
```

```
[Cudafy]
public static void KirpSoldanSaga(GThread thread, int stride, int width, int height, byte[] input, byte[]
output_ragba)
    {
        int i = width - (thread.blockDim.x * thread.blockIdx.x + thread.threadIdx.x) - 1;


        for (int j = 0; j < height; j++)
        {
            int byteOffset = i * stride + j * 4;

            if (output_ragba[byteOffset + 1] == 255)
            {
                break;
            }
            else
            {
                input[byteOffset + 3] = 0;
            }
        }

    }
```

```
[Cudafy]
public static void KirpSagdanSola(GThread thread, int stride, int width, int height, byte[] input, byte[]
output_ragba)
    {
        int i = width - (thread.blockDim.x * thread.blockIdx.x + thread.threadIdx.x) - 1;


        for (int j = height - 1; j > 0; j--)
        {
            int byteOffset = i * stride + j * 4;
```

```
        if (output_ragba[byteOffset + 1] == 255)
        {
            break;
        }
        else
        {
            input[byteOffset + 3] = 0;
        }
    }

}
```