GPU PROGRAMLAMA KULLANAN SEVİYE KÜMELERİ VE UYGULAMALARI

Zafer GÜLER

Yüksek Lisans Tezi
Bilgisayar Mühendisliği Anabilim Dalı
Danışman: Yrd. Doç. Dr. Ahmet ÇINAR

T.C
FIRAT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

GPU PROGRAMLAMA KULLANAN SEVİYE KÜMELERİ VE UYGULAMALARI

YÜKSEK LİSANS TEZİ

Zafer GÜLER

(111129105)

Tezin Enstitüye Verildiği Tarih : 3 Haziran 2013
Tezin Savunulduğu Tarih : 3 Temmuz 2013

Tez Danışmanı : Yrd. Doç. Dr. Ahmet ÇINAR (F.Ü)

Diğer Jüri Üyeleri : Doç. Dr. Arif GÜLTEN (F.Ü)

Yrd. Doç. Dr. Burhan ERGEN (F.Ü)

TEMMUZ - 2013

ÖNSÖZ

Bu çalışma, Fırat Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı Yüksek Lisans Tezi olarak hazırlanan "GPU Programlama Kullanan Seviye Kümeleri ve Uygulamaları" isimli tezi içermektedir.

Yüksek lisans öğrenimim sırasında ve tez çalışmalarım boyunca gösterdiği her türlü destek ve yardımdan dolayı çok değerli hocam Yrd. Doç. Dr. Ahmet Çınar'a en içten dileklerimle teşekkür ederim.

Bu Yüksek Lisans tez çalışmamı, öğrenim hayatım boyunca destekleri ile bu noktaya ulaşmamda büyük katkıları olan değerli aileme armağan ediyorum.

Zafer Güler

Elazığ, 2013

İÇİNDEKİLER

Sayfa No

ÖZET

Son yıllarda grafik işlemcilerinin gelişmesi ile birlikte ekran kartları, genel amaçlı hesaplamalar yapmak için yaygın olarak kullanılmaya başlanmıştır. Özellikle 2007 yılında yayınlanan CUDA C dili ile birlikte birçok araştırmacı, CUDA C programlama dilini yüksek hesaplama gereken işlemler için kullanmıştır.

Bu tez çalışmasında görüntü bölütleme için kullanılan seviye kümesi yöntemi ölçeklendirme yaklaşımı ile GPU üzerinde çalıştırılmıştır. Seviye kümeleri yaklaşımı ağırlıklı olarak kısmi diferansiyel denklemlerin çözümüne dayanmaktadır. Sunulan yöntem kısmi diferansiyel denklem çözümüne gerek duymaz. Temel geometrik dönüşümlerden olan ölçeklendirme yaklaşımını kullanır. Böylece çözüm için gerekli olan hesaplama yükünü hafifletir. GPU üzerinde CUDA programlama kullanılarak performans ve harcanan zaman bakımından avantaj sağlanmış ve çok daha hızlı sonuçlar elde edilmiştir. GPU kullanılması ile gerçek zamanlı işlem yapmaya da olanak sağlamıştır. Geliştirilen uygulama MRI beyin görüntüleri üzerinde tümör bulunması amacı ile kullanılmıştır.

Anahtar Kelimeler: CUDA, Seviye Kümesi Yöntemi, GPU

SUMMARY

Level Set and Application Using GPU Programming

In recent years, with the development of graphics processors, graphics cards have been widely used to perform general-purpose calculations. Especially with release of CUDA C programming languages in 2007, most of the researchers have been used CUDA C programming language for the processes which needs high performance computing.

In this thesis, a scaling approach for image segmentation using level sets is carried out by the GPU programming techniques. Approach to level sets is mainly based on the solution of partial differential equations. The proposed method does not require the solution of partial differential equation. Scaling approach, which uses basic geometric transformations, is used. Thus, the required computational cost reduces. The use of the CUDA programming on the GPU has taken advantage of classic programming as spending time and performance. Thereby results are obtained faster. The use of the GPU has provided to enable real-time processing. The developed application in this study is used to find tumor on MRI brain images.

Key Words: CUDA, Level Set Method, GPU

ŞEKİLLER LİSTESİ

Sayfa No

TABLOLAR LİSTESİ

Sayfa No

KISALTMALAR LİSTESİ

AMD : Aritmetik mantık birimi

CPU : Ana işlemci birimi

CUDA : Compute Unified Device Architecture

GPU : Grafik işlemci birimi

GPGPU : Grafik işlemcide genel amaçlı programlama

L1 : Birinci seviye önbellek

L2 : İkinci seviye önbellek

L3 : Üçüncü seviye önbellek

PDE : Kısmi diferansiyel denklemler

SIMD : Tek komut çok veri akışı

SM : Streaming Multiprocessor

SEMBOLLER LİSTESİ

$\boldsymbol{\varphi, \varphi_x, \varphi_y, \varphi_z}$: Seviye kümesi fonksiyonu

$\boldsymbol{C_p}$: Parabolün merkezi

$\boldsymbol{C_c}$: Çemberin merkezi

$\boldsymbol{r}$: Yarıçap

$\boldsymbol{C}$: Kontur eğrisi

$\boldsymbol{\vec{x}}$: Seviye kümesindeki noktalar

$\boldsymbol{\vec{V}}$: Hız terimi ifadesi

$\boldsymbol{\nabla}$: Kontur eğriliği

$\boldsymbol{D^+\varphi}$: Upwinding

$\boldsymbol{R}$: Kırmızı renk bilgisi

$\boldsymbol{G}$: Yeşil renk bilgisi

$\boldsymbol{B}$: Mavi renk bilgisi

$\boldsymbol{\sigma}$: Standart sapma

$\boldsymbol{\overline{x}}$: X koordinatlarının ortalaması

$\boldsymbol{\overline{y}}$: Y koordinatlarının ortalaması

$\boldsymbol{n}$: Konturdeki nokta sayısı

$\boldsymbol{sr1, sr2}$: Ölçeklendirme katsayıları

1. GİRİŞ

Görüntü analizinde ilk ve en önemli adım görüntüyü bölütlemedir. Bölütleme görüntünün kendisini oluşturan parçalara bölünmesi işlemidir ve bilgisayar görmesinde [1] ve tıbbi görüntülemede [2] önemli işlemler arasındadır. Bölütlemenin bir diğer amacı da görüntülerdeki nesne veya nesnelerin sınırlarının bulunmasıdır. Bu amaçla kullanılan yöntemlerden birisi olan seviye kümesi yöntemi ile görüntüler üzerinde nesne sınırlarını bulmada oldukça başarılı sonuçlar elde edilebilmektedir. Seviye kümesi yöntemi Osher ve Sethian [3] tarafından ilk kez sunulduktan sonra, nesne bulmada/izlemede ve hareket eden yüzeylerin elde edilmesi ve izlenmesinde favori teknik haline gelmiştir. Seviye kümesi yönteminin genel amacı, hareket eden yüzeylerin ve eğrilerin izlenmesidir. Burada hareket eden yapılar hız terimi ile takip edilebilmektedir ve hız terimi zamana, uzaya ve o anki geometriye bağlıdır. Bu yöntem ile topolojik değişimler otomatik olarak yönetilebilir ve zaman değişimi ile birlikte bir nesne birden fazla nesneye bölünebilir veya tam tersi durumlar oluşabilmektedir.

Seviye kümesi yöntemi çamur ve su gibi fiziksel tabanlı simülasyonlar [4] için oldukça uygun olmasına rağmen, yöntem için gerekli olan kısmi diferansiyel denklemlerin(PDE) çözümü yüksek hesaplama gerektirdiğinden sorun oluşturmaktadır. Bu sorunla başa çıkmak için birçok yöntem geliştirilmiştir. Geliştirilen yöntemlerden iki önemli yaklaşım dar band (narrow band) [5] ve seyrek alan tekniği (sparse field technique) [6] yöntemleridir. Bu yöntemlerde sadece sıfır seviye kümesi etrafında PDE gelişimi hesaplanmış ve hız artışı sağlanmıştır. Fakat yeterli hızlanma sağlanamamıştır. Daha sonra yapılan çalışmalarda seviye kümesi yöntemi paralelleştirmeye uygun olduğu görülmüş ve yöntem paralel algoritma ile geliştirilmiştir [7]. Özellikle ekran kartındaki hızlı gelişmeler ile birlikte seviye kümesi yönteminin paralel çalıştırılması hız artırımında önemli bir etken haline gelmiştir.

Son on yıldır çok işlemcili yapılar ortaya çıkmış ve günümüzde yaygın şekilde kullanılmaya başlanmıştır. Bu yapılar hem endüstride hem de akademik araştırmalarda kullanılmış ve problemlere ve zorluklara çözümler getirmiştir. Günümüzde bu yapılar üzerinde birçok problem, algoritma kolay bir şekilde gerçeklenebilmektedir. Çok işlemcili yapılar olarak sadece ana işlemci birimi (CPU) düşünmemek gerekir. Aksine grafik işlemci biriminin (GPU) asıl özelliği işlemlerin paralel çalışmasına izin vermesidir. Fakat GPU

üzerinde uygulama geliştirmek bazı özel kütüphanelerin bilinmesini gerektirmekteydi. Son on yıldır yapılan çalışmalar ile Grafik işlemciler bile kolay bir şekilde genel amaçlı uygulamaları çalıştırır hale gelmiştir.

Grafik işlemcilerinin genel amaçlı uygulamalar için kullanılması son zamanlarda ortaya çıkan bir yaklaşım değildir, fakat 2007 de NVIDIA firması tarafından geliştirilen C programlama dili tabanlı CUDA (Compute Unified Device Architecture) mimarisi ile birlikte hızla yaygınlaşmıştır. CUDA mimarisi yazılımcılara grafik işlemci bilgisi olmaksızın genel amaçlı programlar yapmasını sağlamıştır. GPU tabanlı uygulamalar sadece bilimsel alanda değil, görüntü ve video işleme, akışkanlar dinamiği simülasyonu gibi diğer yüksek performans gerektiren alanlarda da kullanılmaktadır [8].

1.1. Tezin Amacı

Bu tez çalışmasının öncelikli amacı, görüntü işlemede yaygın olarak kullanılan seviye kümesi yönteminin GPU üzerinde CUDA C dili ile daha performanslı ve yeni bir yöntem kullanarak geliştirilmesidir.

Bu amaç doğrultusunda seviye kümesi yöntemi kullanılarak yapılmış çalışmalar incelenmiştir. Ayrıca, seviye kümesi yönteminin gerçekleştirimleri incelenerek, bu tekniğin problemlerini analiz edilmiştir. Analiz tamamlandıktan sonra, bu problemlere nasıl çözüm bulunabilir sorusunun cevabı aranmıştır ve bu doğrultuda bir çözüm gerçekleştirilmiştir. Ayrıca NVIDIA firması tarafından geliştirilen ekran kartları üzerinde çalışan, CUDA C dili incelenmiş ve seviye kümesi yönteminin GPU ile geliştirilmiş çalışmaları analiz edilmiştir. Bunlara ek olarak programda görselliği sağlamak amacıyla OpenGL ile ilgili örnekler incelenerek OpenGL – CUDA entegrasyonu sağlanmıştır.

1.2. Tezin Yapısı ve İçeriği

Bu tez çalışmasında NVIDIA CUDA C dili kullanılarak, OpenGL entegrasyonlu yeni bir seviye kümesi yöntemi geliştirilmiştir. Yazılım Visual Studio 2010 ortamında NVIDA Paralel Nsight eklentisinden faydalanılarak geliştirilmiştir. Geliştirilen yöntem ile görüntü üzerindeki nesneler çok hızlı bir şekilde bulunabilmektedir.

Tezin ikinci bölümünde GPU ve CUDA C diline ait tarihsel gelişim verilerek, CUDA C dilinin mimarisi ve bellek yönetimi anlatılmıştır. Daha sonra, GPU tabanlı seviye kümesi ile ilgili yapılan çalışmalar verilmiştir. Üçüncü bölümde, deforme şablonlar ve seviye kümesi yöntemi kısa bir şekilde tanıtıldıktan sonra, dördüncü bölümde geliştirilen GPU tabanlı algoritma ayrıntılı şekilde anlatılmış ve elde edilen sonuçlar verilmiştir.

2. ÖNCEKİ ÇALIŞMALAR

Bu bölümde grafik işlemci biriminin tarihsel gelişimi verilerek, CPU ve GPU donanımlarını hesaplama gücü ve bant genişliği bakımından karşılaştırılmıştır. Ayrıca GPU mimarisi hakkında bilgi verilerek GPU üzerinde kullanılan bellek türlerine değinilmiş ve kullanılan bellek türünün performans üzerinde son derece etkili olduğu vurgulanmıştır. Son olarak ise geleneksel seviye kümesi yönteminde bahsedilerek, literatürde GPU üzerinde çalıştırılan seviye kümesi yöntemleri verilmiştir.

2.1. Grafik İşlemcilerin Tarihsel Gelişimi ve GPGPU Kavramı

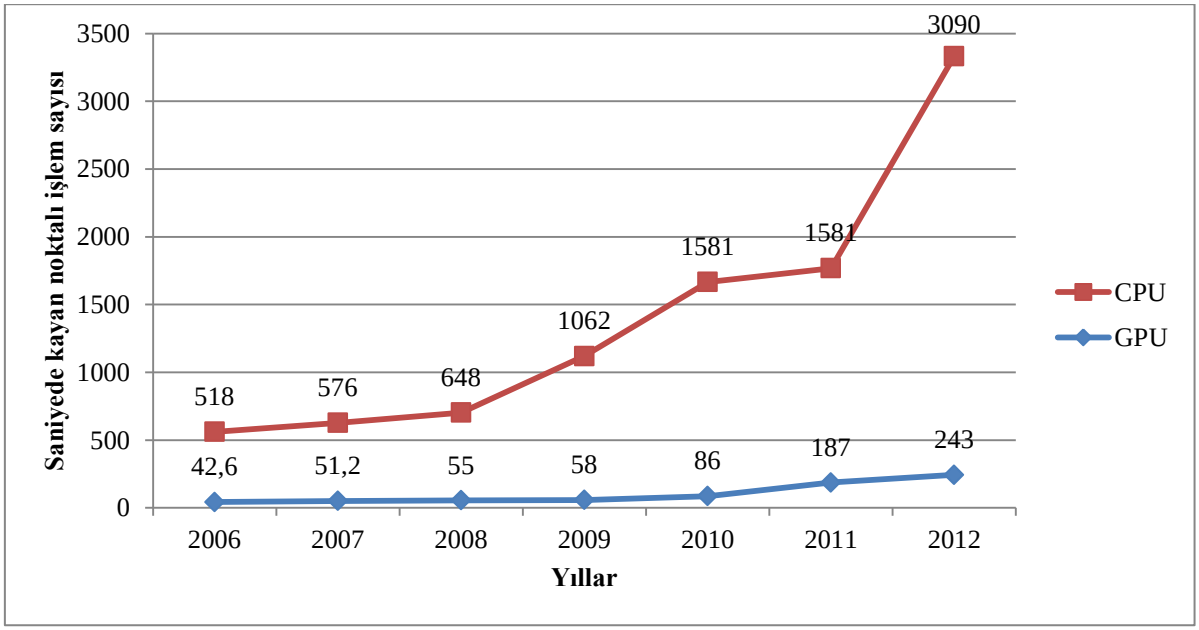
GPU gömülü sistemlerde, cep telefonlarında, kişisel bilgisayarlarda, iş istasyonlarında ve oyun konsollarında yaygın olarak kullanılmaktadır. Bilgisayarlar açısından düşünürsek CPU'ya bazı temel operasyonları sağlarlar. Bunlar arasında bellekte resmi çevirme ve bu resmi ekrana gösterme işlemleri vardır. GPU tipik olarak karmaşık çokgenler kümesini işler yani işlenecek resmin haritasını çıkartır. Daha sonra çokgenlere doku uygulanır ve sonrasında gölgelendirme ve ışıklandırma hesaplamaları yapılır. GPU kavramını, NVIDIA firmasının 1999 yılında piyasaya çıkarttığı GeForge 256 ekran kartı ile birlikte popülerlik kazanmıştır. Yine NVIDIA'nın 5000 serisi foto-gerçekçi etkisini ilk gerçekleştiren karttır [7] ve son on yılda ekran kartı teknolojisi hızlı bir şekilde gelişmiştir.

Önemli gelişim adımlarının bir değeri ise programlanabilir shederlardır. Bu yapılar ile farklı özelliklerdeki uygulamalar küçük birer program gibi çalıştırılabilir. Böylece resim çevirme işlemi sadece GPU üzerinde değil, yüklenebilen shedarlar ile de yapılabilmesine olanak sağlanmıştır. Bu yapı ile birlikte grafik işlemci birimi genel amaçlı programları çalıştırabilir hale gelmiştir. Shaderlar yapısı gereği çokgen haritasını temsil etmek için 3 boyutlu noktalar kullanır ve böylelikle birçok veri kümesinde yüksek oranda paralellik sağlayarak çalışabilirler. Bu ise hesaplama gücünden yüksek oranda verim edilmesi anlamına gelmektedir. Günümüzde RGB resmi gibi bazı veri kümeleri, 3 nokta kümesi şeklinde ifade edilebilmesine rağmen bazı veri kümelerini bu şekilde ifade edilememektedir. Yapılan çalışmalar sonucunda bazı yapılar (BrookGPU, CTM gibi) geliştirilmiştir [9]. Bu yapılar ile genel amaçlı programların GPU üzerinde çalıştırılmasına çalışılmıştır. Ancak bu yapıların avantajları olduğu kadar dezavantajları da vardır ve hepsi

de öğrenilmesi zor yapılardan oluşmaktadırlar. Bu sorunlardan dolayı bu yapılar kitle pazarında etkili olamamışlardır. Fakat CUDA C dilinin duyurulması ile birlikte programcılara gerçek anlamda genel amaçlı program yazmaları sağlanmıştır.

Günümüzde GPU'lar bilgisayar grafiklerini oluşturma konusunda çok verimli çalışmaktadırlar. GPU'ların paralel mimarileri, genel amaçlı CPU'lara göre büyük boyutlardaki veriyi paralel bir şekilde çalıştırma konusunda daha verimli kılmaktadır. Günümüzde paralel çalışan GPU'lar CPU ya karşı hesaplama güncünde oldukça başarılı sonuçlar elde etmektedir. Bu gelişim ile birlikte GPU üzerinde genel amaçlı hesaplama, (General Purpose Computing on GPU - GPGPU) GPU Hesaplamanın bir alt dalı olarak ortaya çıkmıştır. GPGPU yöntemi makine öğrenmesi, petrol arama çalışmaları, bilimsel görüntü işleme, lineer cebir, istatistik, 3-boyutlu modelleme hatta hisse senedi fiyat tahmini gibi birçok bilim dalında kendine yer edinmiştir [10].

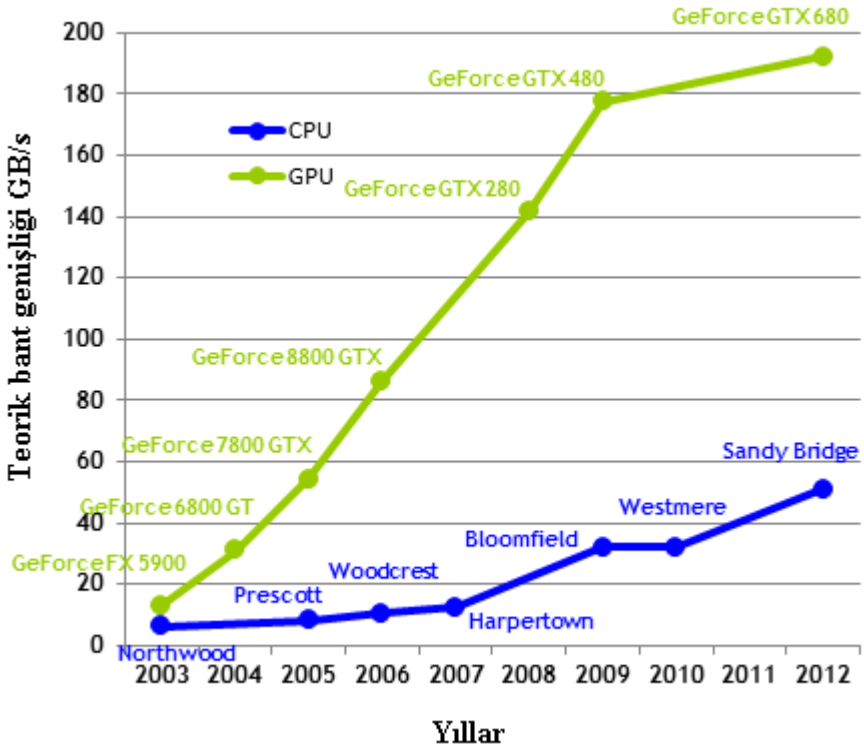
2000'li yıllar ile piyasa tarafından gerçek zamanlı, yüksek çözünürlüklü 3-boyutlu grafiklerin talep edilmesi ile birlikte GPU donanımları çok hızlı bir şekilde gelişmiş ve hesaplama gücü ve bant genişliği kat ve kat artmıştır. Şekil 2.1'de GPU ve CPU kıyaslaması saniyede kayan noktalı işlem sayısına göre (Floating-Point Operations per Second) Şekil 2.1'de bant genişliğine göre kıyaslanmıştır.



Şekil 2.1. Kayan noktalı işlem sayısına göre GPU ve CPU kıyaslaması [10].

Şekilde de görüldüğü gibi 2009 yılından itibaren CPU ile GPU hesaplama gücünde bir ayrışma görülmektedir. 2009 yılında GPU tarafında, G80 donanımından G200 donanımına geçilmiş ve bunu Fermi evrimi izlemiştir. Bu geçişler sırasında ise yüksek hesaplama artışı sağlanmıştır. CPU tarafında ise Intel'in 2 işlemcili mimariden Nehalem mimarisine geçişinde sadece küçük bir artış sağlanmıştır. CPU performansında sadece Sandy Bridge mimarisinde önemli bir artış elde edilmiştir.

Şekil 2.2'de ise CPU ve GPU donanımları bant genişliğine göre kıyaslanmıştır. Şekilde görüldüğü gibi 2003 yılından itibaren GPU ile CPU bant genişliği arasında bir ayrışma görülmektedir.

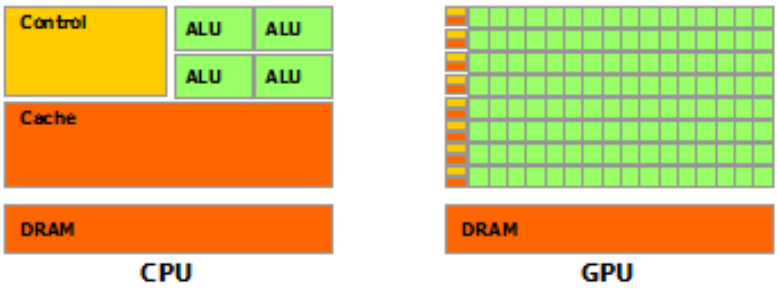


Şekil 2.2. Bant genişliğine göre CPU ve GPU karşılaştırması [10].

2.2. NVIDIA CUDA Mimarisi

Grafik işlemcilerinin hızla gelişmesi ve genel amaçlı uygulamalar için kullanılabilir hale gelmesi ile birlikte, birçok alanda uygulama hızını arttırmak amaçlı kullanılır hale gelmiştir[8]. CUDA mimarisi NVIDIA tarafından geliştirilen ve GPU'nun gücünden yararlanarak hesaplama performansında yüksek artışlara olanak veren bir mimaridir. Bu

mimari, görüntü ve video işleme ve akışkanlar dinamiği simülasyonları gibi temeli yüksek hesaplamaya dayanan birçok alanda kullanılmış [11]. Aslında bu fark GPU mimarisinin paralellik gerektiren işlemler için geliştirilmiş olmasından kaynaklanmaktadır. Şekil 2.3'te CPU ve GPU mimarisi verilmiştir. Şekil 2.3'te görüldüğü gibi GPU çok sayıda aritmetik mantık birimi(AMB) içermesine rağmen ön bellek kapasitesi düşüktür [8].



Şekil 2.3. CPU ve GPU mimarileri [8].

GPU'lar veri kümesi üzerinde aynı anda çok sayıda kanal çalıştıracak şekilde, SIMD(Single Instruction Multiple Data) mimarisinde tasarlanmıştır. Bu nedenle sadece paralelleştirmeye uygun veriler üzerinde başarılı sonuçlar elde edilebilmektedir. Örneğin programınız birçok akış kontrolü içeriyorsa bu hızda artış yerine düşüşe neden olabilmektedir [7].

CUDA, yazılım geliştiricilerin GPU üzerinde genel amaçlı uygulamalar yapabilmeleri için CUDA C dili ile birlikte gelir. CUDA C dili, C dilinin genişletilmiş halidir ve kernel adı verilen C fonksiyonlarını tanımlamaya olanak sağlar. Çağrıldığı zaman N tane CUDA kanal tarafından, paralel bir şekilde çalıştırılırlar. Kernel fonksiyonları __global__ ifadesi ile tanımlanırlar ve aşağıdaki söz dizimi ile çağrılırlar.

```
kernel_fonksiyon<<<blok_sayisi, kanal_sayisi>>>(param1, param2, ...)
```

Yukarıda kullanılan `blok_sayisi` ve `kanal_sayisi` ifadeleri int veri türünde veya dim3 veri türünde verilebilir. `kanal_sayisi` parametresi ile kernel içinde kaç tane kanal çalışması gerektiği ayarlanabilmektedir. Fakat bu sayı donanıma bağımlıdır. Eski ekran kartlarında her bir blokta çalışacak kanal sayısı 512 ile sınırlandırılmasına rağmen günümüzde bu sayı 1024 ile sınırlandırılmıştır [10]. 1024 kanal sayısı CPU tarafında

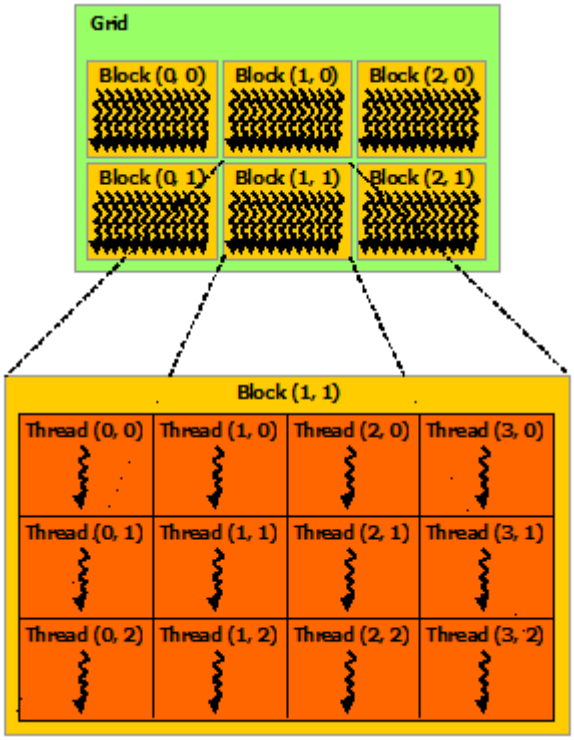
çalışan programcılar için çok büyük bir sayı olabilir fakat işin GPU tarafında verimliliğin elde edilmesi için on binlerce kanal'ın eş zamanlı çalışması gerekmektedir. Bu ise bloklar kullanılarak yapılabilir. Örneğin bir kernel'in 2 blok ve 128 kanal ile çağrıldığını düşünürsek toplam aktif kanal sayısı 2x128=256 olacaktır. Blok sayısında donanımsal sınır ise 65536'dır. Hem kanal sayısı hem de blok sayısını maksimum kullandığımızda ise Fermi donanımında çalıştırılabilecek maksimum kanal sayısı yaklaşık 64 milyon şeklinde hesaplanabilir.

Kernel kanal tarafından çalıştırıldığında, eşsiz bir threadID verilir ve bu değere "threadIDx" değişkeni ile ulaşılır. Bu kimlik numarası ile kernel içerisinde o anda hangi kalalın çalıştığı belirlenebilir. Aynı kanallarda olduğu gibi bloklara da "blockIdx" kimlik numarası ile erişilebilir. Böylece büyük veri kümelerinde hangi bloktaki hangi kanalın çalıştığı kolayca belirlenebilir. Kanal ve bloklar 1 boyutlu, 2 boyutlu, 3 boyutlu olabilmektedir. 2 ve 3 boyutlu kanal ve blokların kullanılması hangi kanalın çalıştığının belirlenmesinde ekstra işlemler yapılmasını gerektirir. Örneğin 2 boyutlu kanal ve blok için kanal_id aşağıdaki şekilde hesaplanabilmektedir. Kernel içerisinde blok boyutlarına erişmek istendiğinde yerleşik "blockDim" değişkeni ile erişilebilir.

```
unsigned int idx        = (blockIdx.x * blockDim.x) + threadIdx.x;
unsigned int idy        = (blockIdx.y * blockDim.y) + threadIdx.y;
unsigned int thread_idx = ((gridDim.x * blockDim.x) * idy) + idx;
```

Bloklar ise gridleri oluştururlar. Gridler 1 veya 2 boyutlu olabilirler ve kernel içerisinde gridDim yerleşik değişkeni ile sayısına erişilebilir. Şekil 2.4'te grid, blok ve kanalın genel yapıları gösterilmiştir.

NxN boyutlu A ve B matrisleri toplayarak C matrisine kaydeden CUDA C kodu çoklu blok kullanılarak kodlanmış ve Şekil 2.5'te verilmiştir.



Şekil 2.4. Grid içindeki blok[10].

```
// Kernel tanımı
__global__ void MatrisEkleme(float *A, float *B, float *C){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main(){
    ….
    // Kernelin çağrılması
    dim3 threadSayisi(16,16);
    dim3 blokSayisi ((N / threadSayisi.x)+1, (N / threadSayisi.y)+1);
    MatrisEkleme<<< blokSayisi, threadSayisi>>> (A,B,C);
    ….
}
```

Şekil 2.5. Matris toplamı için örnek CUDA C kodu

Programda kullanılan 16x16 (256) boyutunda kanal bloğu kullanılmıştır. Griddeki blok sayısı ise her bir elemana bir kanal düşecek şekilde oluşturulmuştur. Programda seçilen kanal ve blok sayıları rastgele seçilmesine rağmen 16x16'lik kanal bloğu genel olarak tercih edilmektedir.

Kanal blokları herhangi bir sırada her hangi bir zamanda çalıştırılabilirler. Dolayısıyla kanal blokları birbirinden bağımsız bir şekilde çalışabilecek bir yapı oluşturulmalıdır. Bu bağımsızlık bellek paylaşımı ve senkronizasyon işlemlere gereksinimi ortaya çıkartır. Sekronizasyon işlemi kernel içerisinde __syncthreads() fonksiyonu kullanılarak gerçekleştirilebilir. __syncthreads() fonksiyonu ile bloktaki tüm kanalların aynı noktada olmaları sağlanır [10].

2.3. CUDA ile Bellek Yönetimi

Geleneksel CPU modelinde doğrusal veya düz şeklinde isimlendirilen bir bellek yönetimi mevcuttur. Bu modelde herhangi bir CPU işlemcisi belleğin herhangi bir yerine izin almaksızın erişebilmektedir. Pratikte CPU donanımında birinci düzey (L1), ikinci düzey (L2) ve üçüncü düzey (L3) önbellek görülmektedir. CPU kodunu optimize ederek yüksek performans elde etmek isteyen araştırmacılar bu yapılar ile çalışmaktadırlar.

CUDA mimarisinde ise farklı karakteristiklerde birçok bellek türüne erişimi desteklemektedir. Bu bellek çeşitleri global, yerel, paylaşılan, doku ve kayıtçılardır. Tablo 2.1'de her bir bellek türü bant genişliği ve gecikme süresi verilmiştir. Tablo 2.1'de de görüldüğü gibi kayıtçılar en hızlı çalışan ve gecikme süresi en az olan bellek türüdür. Daha sonra paylaşılan bellek programcı tarafından verimlilik için tercih edilmektedir. En sonda ise doku, sabit ve global bellek gelmektedir. Bu bellek erişimlerinin her biri aşağıda değerlendirilmiştir.

Tablo 2.1. Bellek türüne göre erişim zamanı [12].

	Kayıtçı	Paylaşılan Bellek	Doku Belleği	Sabit Bellek	Global Bellek
Bant Genişliği	~8 TB/sn	~1.5 TB/sn	~200 MB/sn	~200 MB/sn	~200 MB/sn
Geçikme Süresi	1 devir	1 ile 32 devir arası	~400 ile 600	~400 ile 600	~400 ile 600

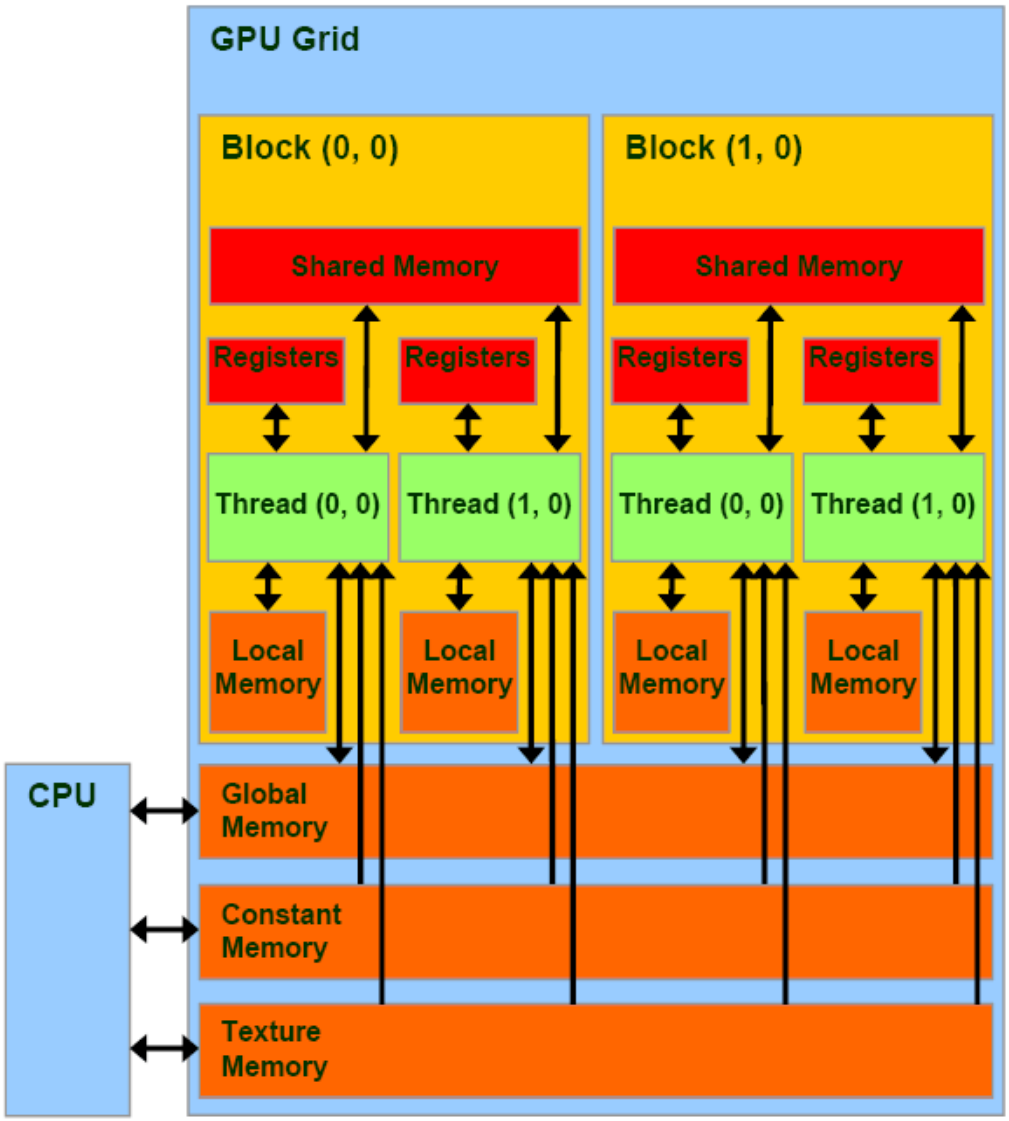
Şekil 2.6'da kanallar, bloklar ve erişebildikleri bellek türleri gösterilmiştir. Her bir kanal kayıtçılara, yerel belleğe, paylaşılan belleğe, global belleğe erişebilmektedir. Bunların yanında tüm kanallar tarafından sadece okuma yapılabilen sabit bellek ve doku bellek alanları da vardır. Uygulama tarafında ise global belleğe, sabit belleğe ve doku belleğe erişilebilmektedir.

GPU tarafında kullanılan bellek çeşitleri ve her birinin özellikleri aşağıda verilmiştir.

Kayıtçı: Kayıtçı kanal içinde tanımlı olan değişkenlerdir. Kanal dışından bu değişkenlere erişilemez. Genel olarak fonksiyon içindeki local değişkenleri kaydetmek için kullanılırlar. Programlamada herhangi bir ekstra işlem gerektirmezler. Kayıtçı olarak tanımlanan bir değişkene veri yazmak veya okumak çok hızlı gerçekleşmesinden dolayı verilerin olabildiğince kayıtçılarda tutulması istenir. Fakat kayıtçı bellegi çok küçük boyuttadır. Hesaplama kapasitesi 2.x ve 3.0 olan kartlar için her bir kanal içinde maksimum tanımlanabilecek 32 bitlik kayıtçı sayısı 63'tür [10].

Yerel Bellek: Aynı kayıtçılar gibi sadece kanal içinde geçerlidir fakat fiziksel yeri çip dışında olduğu için bellek erişimi, global bellek erişimleri gibi yavaştır. Yerel değişken tanımlama işlemi __local__ anahtar sözcüğü ile tanımlanırlar. Hesaplama kapasitesi 2.0 ve üstü kartlarda önbellek işlemi yapılmaktadır.

Paylaşılan Bellek: Blok içindeki tüm kanallar tarafından erişilebilen bellek yerleridir. Cip üzerinde olduğu için yerel ve global belleğe göre daha yüksek bant genişliği ve daha düşük gecikmeye sahiptir. Genel olarak kayıtçılar kadar hızlıdırlar. __shared__ anahtar sözcüğü ile tanımlanırlar. Bir blok içinde sadece bir değişkenin paylaşılan şeklinde tanımlanması gerekmektedir [8,10]. Paylaşılan bellekte eş zamanlı bellek isteklerine karşı yüksek bant genişliği sağlamak için, paylaşılan bellek eşit bir şekilde bellek modüllerine bölünür (bank), böylece belleğe eş zamanlı erişimler sağlanabilmektedir.



Şekil 2.6. GPU dananım mimarisi ve bellek çeşitleri [13].

Global Bellek: Tüm uygulama tarafında erişilebilen bellek çeşididir. __device__ anahtar kelimesi ile tanımlanır. Veri transferi çok yavaştır [8,10,12,14]. Hesaplama kapasitesi 2.0 ve üstü kartlarda önbellek işlemi yapılmaktadır.

Sabit Bellek: GPU tarafından sadece okunabilen bellek bölgesidir. Toplam 64KB lik sabit bellek bölgesi bulunmaktadır. Her bir streaming multiprocessor(SM) kendi içinde önbellekleme yapar. Çalışan her bir kanal sabit bellekten aynı anda okuma yapabilir. Böylelikle çok hızlı veri transferleri elde edilir [8,12].

Doku Bellek: Sadece okuma yapılabilen önbelleğe alınmış bellek çeşididir. Eğer istenilen veri önbelleğe alınmışsa sadece bir önbellekten okuma işlemi ile veri elde edilebilir. Eğer istenilen veri önbelleğe alınmamışsa global bellekten okuma ile veri elde edilebilir. Doku önbelleği 2 boyutlu mekânsal yerellik için optimize edilmiştir. Böylece en iyi performans elde edilmesi sağlanmıştır [8].

Bellek tiplerinin çeşitli özellikleri Tablo 2.2'de verilmiştir.

Tablo 2.2. Bellek tiplerinin çeşitli özellikleri [15].

Bellek	Çip Üzerindeki Yeri İçinde / Dışında	Önbellege Alınmış	Erişim	Faaliyet Alanı	Yaşam Ömrü
Kayıtçı	İçinde	Hayır	Okuma/Yazma	1 kanal	Kanal
Yerel	Dışında	*Evet	Okuma/Yazma	1 kanal	Kanal
Paylaşılan	İçinde	Hayır	Okuma/Yazma	Bloktaki tüm kanallar	Blok
Global	Dışında	*Evet	Okuma/Yazma	Tüm kanallar + CPU	CPU yer ayırma
Sabit	Dışında	Evet	Okuma	Tüm kanallar + CPU	CPU yer ayırma
Doku	Dışında	Evet	Okuma	Tüm kanallar + CPU	CPU yer ayırma

* Hesaplama kapasitesi 2.0 ve üstünde geçerlidir.

2.4. GPU Üzerinde Görüntü İşleme ve Seviye Kümesi Yöntemi

CUDA C dili geliştirilmeden önce ekran kartını genel amaçlı programlar için kullanmak oldukça zor bir süreç gerektirmekteydi. Ayrıca GPU programlamada kullanılan OpenGL ve DirectX gibi kütüphaneleri kullanımının zor olması sonucu, herkesin yararlanabileceği programlama modellerinin geliştirilmesi yönünde çalışmalara başlanmıştır. Yapılan çalışmalar sonucunda bazı yapılar (BrookGPU, CTM gibi)

geliştirilmiştir. Bu yapılar ile genel amaçlı programların GPU üzerinde çalıştırılmasına çalışılmıştır. BrookGPU projesi CUDA C dilinin de başlangıcı olmuştur.

CUDA, NVIDIA tarafından geliştirilmiş ve ilk SDK (Software Development Kit) Şubat 2007'de yayınlanmıştır. Bu tarihten itibaren CUDA hem ticari hem de araştırma amaçlı birçok bilim dalında kullanılmıştır [10]. Örneğin veri madenciliği alanında 2013 yılında You Li ve diğ. [16] küme analizinde önemli rol oynayan k-Means algoritmasını GPU üzerinde çalıştırmışlardır. Yapılan çalışma düşük boyutlu veri kümesi ve yüksek boyutlu veri kümesi şeklinde ikiye ayrılmıştır. Düşük boyutlu veri kümesi için kayıtçılar, yüksek boyutlu veri kümesi için ise paylaşılan bellek (shared memory) kullanılmıştır. Yapılan analizlerde bu yaklaşımın daha önce yapılan GPU tabanlı k-means algoritmalarında daha hızlı olduğu ortaya konmuştur.

Seviye kümesi algoritmasının ilk GPU gerçekleştirimi ise Rumpf ve Strzodka [17] tarafından 2001 yılında yapılmıştır. Sayısal hesaplamalar için modern grafik kartlarının yüksek performansından ve yüksek bant genişliğinden yararlanma düşüncesinden yola çıkarak seviye kümesi algoritmasının sayısal işlemleri GPU üzerinde gerçekleştirilmiştir. Yapılan çalışmada 2-boyutlu (128^2) resimler kullanılmış ve birinci dereceden temel seviye kümesi modelinde tek bir adımın hesaplama süresi 2ms olarak ölçülmüştür. Ayrıca çalışmada grafik donanımlarının gelişmesi ile birlikte 3-boyutlu çalışmaların yapılabileceği belirtilmiştir.

2002 yılında Aaron Lefohn ve Ross Whitaker [18] tarafından yapılan çalışmada 2-boyutlu ve 3-boyutlu düzey kümesi çözümünü GPU tabanlı olarak gerçekleştirilmiştir. Geliştirilen uygulama hız teriminin yanında eğrilik akışını hesaplayabilen ilk GPU tabanlı uygulamadır. Uygulama OpenGL tabanlı olup, ATI Radeon 8500 grafik işlemcisinde çalıştırılmıştır. Uygulama 256x256 resimlerde ve 256x256x175 MRI beyin veri kümesinde test edilmiştir. Yapılan testler sonucunda 2-boyutlu resimlerde tek bir adımın hesaplanması 4ms sürmektedir. Bu ise yüksek derecede optimize edilmiş CPU tabanlı uygulamalara göre yaklaşık olarak benzerdir. 3 boyutlu resimlerde ise geliştirilen uygulama 2-kat hızlı olmasına rağmen birçok hesaplamanın 10 kata varan hızlandıkları ölçülmüştür. Bazı yapılan iyileştirmeler ile 20 kata varan hızlanmaların elde edilebileceği vurgulanmıştır. 2004 yılında Aaron Lefohn ve diğ. [19] tarafından yapılan bir diğer çalışmada ise, 2002 yılında yapılan çalışma [18] geliştirilmiştir. Yapılan çalışmada önerilen çözüm, dar-bant algoritmasının gerçek zamanlı aktarılmasıdır. Yeni algoritma ile seviye kümesi eğrisinin verisi çok boyutlu sanal bellek sistemi aracılığıyla 2 boyutlu doku belleğine yüklenir. Eğri

hareket ettiğinde, bu doku tabanlı gösterim GPU'dan CPU'ya mesaj geçiş algoritması ile dinamik olarak güncellenmektedir. Böylece kullanıcı seviye kümesi algoritması çalışırken eğri gelişimini takip edebilmektedir. Geliştirilen uygulama Intel Xeon 1,7 GHz işlemcide, 1 GB RAM ile ATI Radeon 9800 Pro GPU üzerinde çalıştırılmıştır. 256x256x175 boyutundaki veri üzerinde seviye kümesi algoritması çalıştırılmış ve tümör bölütlemede saniyede 70 adım tamamlanmıştır. Buna karşılık optimize edilmiş CPU versiyonu (Insight Toolkit 2003) [20], saniyede 7 adım tamamlayabilmiştir. Yapılan testlerde elde edilen sonuçlara göre CPU tabanlı gerçekleştirime(Insight Toolkit 2003) göre 10 ile 15 kat hız artışı sağlamıştır.

2007 yılında Oliver Klar [21] tarafından yapılan çalışmada büyük miktarda 3-boyutlu imgeler üzerinde seviye kümesi yöntemi GPU tabanlı olarak uygulanmıştır. Geliştirilen 3-boyutlu bellek yönetimi ile sadece hesaplamada kullanılacak alanlar GPU Ram'ine alınmaktadır. Daha sonraki döngülerde ihtiyaca göre GPU belleğinde dinamik bir şekilde yer değişimine izin verilmiştir. Geliştirilen algoritma üç aşamadan meydana gelmektedir. Birinci aşamada seviye kümesi fonksiyonunun ayarlanmasıdır. Bu aşamada işaretli uzaklık alanı kullanılmış ve 0 ile 1 arasındaki değerler hesaplama alanına katılmıştır. İkinci aşamada hesaplama alanındaki her bir nokta için kernel fonksiyonu çalıştırılmıştır. Kernel fonksiyonunda seviye kümesi algoritması için gerekli olan kısmi diferansiyel denklem çözülmüşütr. Üçüncü aşamada ise hesaplama alanının güncelleştirmeleri yapılarak kullanıcıya sunulmuştur. Yapılan çalışma 768 MB video belleğine sahip GeForge 8800 GTS ekran kartında çalıştırılmış ve 512x512x184 boyutundaki MR-beyin veri kümesinde test edilmiştir. 3-boyutlu veri kümesinde seviye kümesinin ayarlanması 0.08 saniye sürerken, tam bir döngü ise 0.68 saniyede hesaplanmıştır.

2009 yılında Hormuz Mostofi ve Keble College [22] tarafından yapılan çalışmada geleneksel seviye kümesi algoritması GPU'ya uyarlanmıştır. Yapılan çalışmada 2-boyutlu ve 3-boyutlu imgeler kullanılmış ve 4 farklı uygulamanın çalışma süreleri karşılaştırılmıştır. Birinci uygulama MATLAB kullanılarak yapılan CPU tabanlı uygulamadır. İkinci uygulama C programlama dili kullanarak yapılan CPU tabanlı uygulamadır. Üçüncü uygulama optimize edilmemiş GPU tabanlı uygulamadır ve son uygulama ise optimize edilmiş GPU tabanlı algoritmadır. GPU algoritması optimize edilirken paylaşılan bellek kullanılmıştır. Tüm uygulamalar Intel Core 2 Duo T8100 2.1 GHz işlemcili, 4GB ram belleğe sahip bilgisayarlarda çalıştırılmıştır. Grafik donanımı olarak ise NVIDIA GeForge 8600M GT ekran kartı kullanılmıştır. 2 boyutlu 256x256

boyutundaki resimlerde 5000 adımı MATLAB ile 425,95 saniyede, C dizi ile 55,44 saniyede, optimize edilmemiş CUDA ile 8.38 saniyede ve optimize edilmiş CUDA ile 1.73 saniyede tamamlanmıştır. Geliştirilen dört farklı versiyon, 3 boyutlu 181x217x181 boyutundaki BrainWeb MRI verisi üzerinde 1000 adım için test edilmiştir. MATLAB ile çözüme ulaşılamamış, C programlama dili ile 4697,5 saniyede, optimize edilmemiş CUDA ile 392,8 saniyede, optimize edilmiş CUDA ile ise 141,2 saniyede sonuçlar elde edilmiştir. Yapılan testlerde optimize edilmemiş GPU tabanlı uygulamanın bile CPU tabanlı uygulamalardan kat kat hızlı olduğu gözlemlenmiştir.

2009 yılında Aaron Hagan ve Ye Zhoa [23] tarafından büyük boyutlu 3D görüntüler üzerinde seviye kümesi algoritması GPU kümesi üzerinde çalıştırılmıştır. Geliştirilen yöntem imge bölütleme için seviye kümesi algoritmasını kullanmaktadır. Seviye kümesi denkleminin çözümü için Lattice Boltzmann Modelin (LBM) geliştirilmesi ile alternatif sayısal çözüm sağlanmıştır. Seviye kümesi denkleminin çözümü için LSB metodunun kullanılması bazı avantajlar sağlamaktadır. Bunlardan ilki, gerçekleştiriminin anlaşılır ve kolay olmasıdır. Bir diğeri, eğrilik hesabını tamamen içermesidir. Üçüncüsü, sonuçların yumuşaklığını kontrol etmek için eşsiz bir parametre içermesidir. Son olarak ise, düşük bütçeli grafik donanımları için paralelleştirmeye uygundur. Geliştirilen uygulama tek bir GPU üzerinde kolayca çalıştırılabilmektedir fakat uygulamanın büyük veri kümelerinde çalışması amacıyla 17 adet GPU kullanılmıştır. 3 boyulu veri kümesi 16 parçaya bölünerek, her bir parça 768 MB bellek kapasitesine sahip Nvidia 8800 GTX ekran kartında çözümlenmiştir. Kalan bir adet GPU ise verileri bölme, toplama gibi işlemleri üstlenmiştir. Uygulama çeşitli veri kümeleri üzerinde çalıştırılmıştır. Örneğin Bonsai veri kümesinde (1024x1024x308 boyutlu) çalıştırılan uygulamada bir döngü 6,81sn de tamamlanmıştır. Bonsai veri kümesinin bölütlenmesi 45 döngüde tamamlanmaktadır. Böylece toplam bölütleme süresi yaklaşık olarak 306sn olarak ölçülmüştür.

2010 yılında Gabor J. Tornai ve György Cserey [24] tarafından 2-boyutlu ve 3-boyutlu imgelerde seviye kümesi algoritması GPU üzerinde çalıştırılmıştır. Yapılan çalışmada kullanılan seviye kümesi algoritması Yonggang Shi ev William Clem Karl [25] tarafından geliştirilmiştir. Yapılan uygulama Intel Core 2 Duo @3.1GHz işlemcili, 8 GB ram belleğe sahip, NVIDIA GTX 280 GPU üzerinde çalıştırılmıştır. Yapılan uygulamanın GPU tabanlı algoritması verilmiş ve CPU tabanlı uygulamadan çok daha hızlı çalıştığı vurgulanmış fakat tam bir oran vermenin ölçmedeki zorluklardan dolayı sağlıklı olmayacağı vurgulanmıştır.

2010 yılında Mike Robert ve diğ. [26] tarafından yapılan çalışmada level set algoritması başarılı bir şekilde GPU'ya uyarlanmıştır. Geliştirilen algoritmanın adım-karmaşıklığı logaritmik, çalışma-karmaşıklığı ise lineerdir ve her ikisi de tüm seviye kümesi alanı yerine aktif hesaplama alanına bağımlıdır. Geliştirilen algoritma Intel 2,5 GHz Xeon işlemcili, 4 GB ram belleğe sahip, NVIDIA GTX 280 GPU üzerinde çalıştırılmıştır. Geliştirilen algoritma 3 boyutlu medikal imgeler üzerinde uygulanmıştır. Daha önceki GPU tabanlı seviye kümesi algoritmalarına göre işlenen seviye kümesi elemanı sayısı 16 kat azalmıştır. Hız olarak ise bilinen GPU tabanlı seviye kümesi algoritmalarına göre, bölütleme hassasiyetinde azalma olmaksızın, 14 kat hızlı çalıştığı vurgulanmıştır.

2013 yılında Andrei C. Jalba ve diğ. [27] tarafından seyrekleştirilmiş level set yöntemi 3-boyutlu yüzey oluşturma amaçlı olarak kullanılmış ve GPU üzerinde gerçekleştirilmiştir. Bu çalışmada konu üzerinde yapılan CPU tabanlı uygulamalara göre 20x'e varan hız artışı elde edilmiştir. GPU tarafında ise Mike Roberts [26] tarafından yapılan çalışma ile kıyaslanmıştır. Yapılan karşılaştırmalarda dar banttın daha büyük seçilmesine rağmen 5x'e varan hız artışları elde edilmiştir.

2.5. Sonuç

Tezin bu bölümünde GPU mimarilerinin tarihsel gelişimi, günümüzde kullanılan, NVIDIA firması tarafından geliştirilen GPU mimarisinin yapısı ve bellek yönetimi verilmiştir. Daha sonra ise seviye kümesi yönteminin GPU üzerinde çalıştırıldığı çalışmalar verilmiştir. GPU mimarileri yapısı gereği paralel programlamaya uygun donanımlardır. Dolayısıyla bir problemin çözümü paralel bir şekilde çalışmaya uygunsa, bu problem GPU mimarisine dönüştürülerek daha performanslı bir şekilde gerçeklenebilir. Ayrıca problemin tüm parçalarının paralel çalıştırılmasına da gerek duyulmaz. Yüksek performans gerektiren paralel çalışmaya uygun yapılar GPU üzerinde çalıştırılırken, giriş-çıkış işlemleri gibi paralel çalışmaya uygun olmayan ve çok hesaplama gerektirmeyen işlemler CPU üzerinde çalıştırılabilir. Bu tür programlamaya heterojen programlama denilmektedir.

Yapılan çalışmalardan elde edilen sonuçlardan görüldüğü gibi, GPU kullanan programlar CPU kullanan programlara göre kat ve kat hızlı çalışmaktadır. Fakat yeterli hız artışı sağlamak için kullanılacak ayarlar ve bellekler dikkatli seçilmelidir. Örneğin tüm

işlemleri global bellek üzerinde yapılması büyük performans kayıplarına yol açmaktadır. Bunun yerine olabildiği ölçüde, daha hızlı erişimlerin olduğu paylaşılan bellek ve kayıtçı kullanılması gerekmektedir.

3. SEVİYE KÜMESİ YÖNTEMİ İLE BÖLÜTLEME

Verilen verilere dayanarak eğri yüzeyinin veya yüzeylerinin tekrar oluşturulması bilinen bir stratejidir. Bu stratejide, ölçülen yüzeylerin bilinen özelliklerini veya eğilimleri ile veri birleştirilerek sonuç elde edilmeye çalışılır. En yüksek benzerlik ile yüzeyin bulunması genel olarak lineer olmayan optimizasyon problemleridir.

Optimizasyon problemlerinin çözümü için ortak ve genellikle etkili bir strateji varyasyon yaklaşımı kullanmaktır. Bu yaklaşımda başlangıç değeri ile başlanır ve parametreler değiştirilerek genel çözüm iyileştirilmeye çalışılır. Varyasyon yaklaşımda bulunan hedef odaklı deformasyonlar tabi ki bir modelleme teknolojisini gerektirir. Genellikle deforme modeller şeklinde isimlendirilen bu tür modeller, bilgisayar grafikleri ve bilgisayar görmesinde yaygın olarak kullanılmaktadır [28, 29]. Bu modeller genellikle amaç fonksiyonu üzerinde iteratif bir şekilde bazı stratejileri uygularlar.

Geleneksel geometrik modeller parametriktir. Bu parametreleştirme işlemi, şeklin sonlu sayıdaki tüm olası şekillerin bulunmasını sağlayan bir fonksiyona dayanmaktadır. Bu yöntemdeki kısıtlama ise modelin temsil edebileceği şekillerin sınırlı olmasıdır. Model genel olarak başlangıç durumundan çok da fazla değişememektedir. Bu ise böyle bir parametreleştirmenin geçersiz olması anlamına gelmektedir.

Parametrik yönteme alternatif olacak yöntem ise kapalı (örtülü) modeldir. Bu modelde φ sayısal fonksiyonu, seviye kümesi olarak belirtilir. Bu sayısal fonksiyon ayrık ızgara üzerinde örneklenebilir. Seviye kümesi fonksiyonu pratik ve teorik olarak geleneksel parametrik modellere göre özellikle deformasyon ve yeniden oluşturma konularında avantajları vardır. Birincisi, seviye kümesi topolojik olarak esnektir. Bu ise modelin parçalara ayrılabilmesine ve birden çok nesneyi temsil etmesini sağlamaktadır. İkincisi, φ'nin gelişimi diferansiyel ifadesidir. Bu ise dönme ve öteleme gibi geometrik dönüşümlerden etkilenmemesini sağlar. φ seviye kümesi ile oluşturulan şekiller, sadece temsil etmek için kullanılan ayrıklaştırma örneğinin çözünürlülüğü ile sınırlıdır. Son olarak, modelin temsil edilmesinde büyük ölçekli çözümler geliştirilebilir. Örneğin, seviye kümesi fonksiyonu çözüme daha büyük ızgara boyutları ile başlarken, gelişim ile birlikte, daha sık ızgara aralıkları kullanılır. Bu tür bir ızgara yapısı kullanmak hesaplama gereksinimini azaltacaktır.

Bu bölümde öncelikli olarak deforme şablonlar hakkında bilgi verildikten sonra, seviye kümesi yöntemi anlatılacaktır.

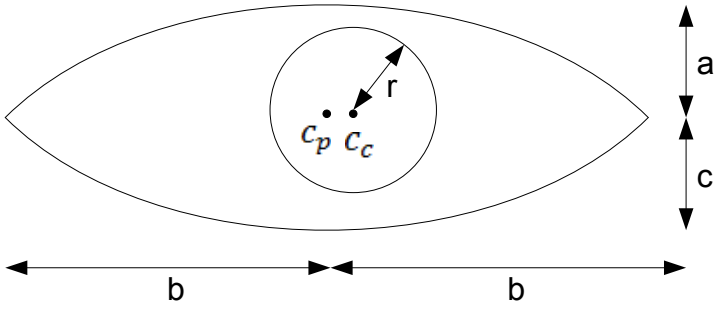
3.1 Deforme Şablonlar

Nesne çıkarımında, tahmininde kullanılan tekniklerden biri deforme şablonlardır. Bu yöntemde tanımlanacak nesne hakkında doğrudan bilgi sahibi olmak gerekmektedir. Bu bilgi kenar bilgisi, ikilik şablon veya parametrik prototip olabilmektedir ve şablon şeklinde isimlendirilir. Ayrıca bu bilginin resimdeki sınıflar ile tamamen eşleşmesi gerekmemektedir. Bu tür modeller ikilik şablon veya bir grup parametre ile tanımlanabilecek özel şekil bilgileri için kullanılır.

Şablon, nesne şeklinin en bilinen veya en çok karşılaşılan örneği olarak tanımlanır. Daha sonra şablona parametrik dönüşümler ile sınırları deforme edilerek şeklin çok farlı kombinasyonları elde edilir. Burada deforme edilirken deforme parametreleri kullanılır ve bu parametreler değiştirilerek aynı nesneye ait farklı şekiller elde edilir. Örneğin kullanılacak şekil insan kalbi ise, birbirine yapı olarak benzerler fakat kişiden kişiye farklılık göstermektedir. Bu küçük değişimler şablondaki küçük değişimler ile yakalanabilir böylece deforme şablonlar orijinal şablona göre daha iyi eşleşme sağlayacaktır. Nesne gürültülü veya bozulmuş olabilir, fakat aynı şekilde orijinale göre deforme şablonlar daha iyi sonuç verecektir.

Deforme şablon analizinde önceki çalışmalardan birisi tanıma amaçlı yüze ait olan özellikleri bulmayı amaçlamıştır [30]. Bu yaklaşım gözün göz akı ve bunun içinde olan iristen oluştuğunu ve bunların parabolüm içinde bir çember ile modellenebileceğini söylemektedir. Bu iki şekli birleştirdiğimizde ve bunların boyut ve yönelim değişimine izin verdiğimizde deforme şablona ulaşırız. Parabol şekli, birbiri ile ilişkili noktalar kümesi (x,y) şeklinde aşağıdaki denklem ile ifade edilebilir [31]. Kullanılan göz şablonu ise Şekil 3.1'de gösterilmiştir.

$$y = a - \frac{a}{b^2}\, x^2 \qquad \textbf{(3.1)}$$



Şekil 3.1. Kullanılan göz şablonu [31].

Şekilde a parabolün yüksekliğini ve b ise yarıçapını göstermektedir. Böylece maksimum yükseklik a, minimum yükseklik ise 0'dır. Aynı denklem b ve c değerleri ile alt parabol içinde geçerlidir. Her iki parabolüm merkezi c_p noktasıdır. Çember ise r yarıçapı ve c_c merkezi ile temsil edilir. Böylece bu şablon ile en iyi eşleşmeyi sağlayan görüntü verisini aramaya başlayabiliriz.

Bu yapıdaki problem ise, çok fazla değişkenin olmasıdır. Örneğin yukardaki göz şablonu için toplam parametre sayısı 11'dir [32]. Yine de tüm olası değerleri deneyerek en iyi şablon bulunabilir fakat her bir parametre için 100 olası değer alınırsa, toplam bakılacak şablon sayısı 10^{22} şeklinde bulunur. Bu ise tek bir bilgisayarın işlem gücünü aşmaktadır. Burada uygulanabilecek iki teknik vardır. Birincisi, optimizasyon teklikleridir. Bunlar arasında azalan eğim tekniği ve genetik algoritmalar teknikleri vardır. Şu an için en favori olan ve deforme şablonlarda en iyi sonucu veren teknik genetik algoritma tekniğidir. Alternatif ve farklı bir teknik ise etkisi az olan veya değişimin çok az olduğu parametreleri çıkartarak, deforme şablonun oluşturulmasıdır.

3.2. Seviye Kümesi Yöntemi

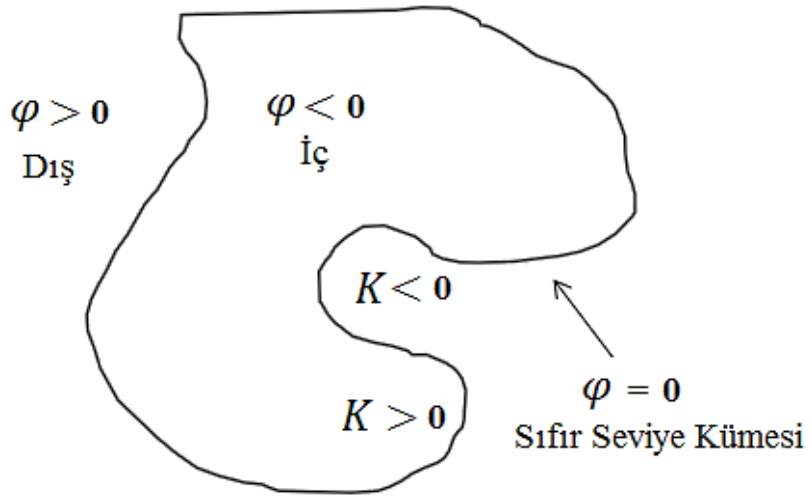
Eğri gelişim ve aktif kontur fikri ilk olarak [28] tarafından tanıtılmıştır. Yapılan çalışmanın ana fikri, eğrinin iç ve dış özelliklerine bağlı olarak enerji fonksiyonunu minimize etmesine dayanıyordu. Bu model Lagrangian yapısı kullanarak kısmi diferansiyel denklemin çözülmesine bağlıydı. Daha sonra Osher ve Sethian [3] tarafından kapalı hiper yüzeyde sıfır seviye kümesi şeklinde tanımlanmıştır. Bu Euler-Lagrange yapısının eğri gelişimi için ilk kullanılmasıdır. Bu yöntemde Euler-Lagrange denklemini sayısal olarak

çözmek için Hamilton-Jacobi metodu kullanılmıştır. Seviye kümesi yönteminde ilk noktaların, noktadan noktaya izleme işlemine gerek kalmamıştır. Ayrıca birleştirme ve ayrılma gibi işlemler otomatik olarak yapılabilmektedir.

1987 yılında Osher ve Sethian [3] tarafından geliştirilen yöntem ile dinamik arayüz ve şekillerin seviye kümesi yöntemi ile yakalanması sağlanmıştır. Seviye kümesi yönteminin temel fikri, seviye kümesi fonksiyonu şeklinde isimlendirilen yüksek dereceli bir fonksiyonun, sıfır seviye kümesi (zero level set) şeklinde kontur ile ifade edilmesi ve konturün hareketinin ise seviye kümesi fonksiyonunun gelişmesi ile formüle edilmesidir. Bu çalışmadan sonra seviye kümesi yönteminin popülaritesi artmış ve sayısal geometri, akışkanlar dinamiği, görüntü işleme ve bilgisayar görmesi gibi geniş kapsamlı birçok uygulamada kullanılmıştır. Seviye kümesi denklemi aşağıda verilmiştir [31].

$$\varphi_t = F|\nabla\varphi| = 0 \tag{3.2}$$

Burada φ kapalı bir fonksiyonudur ve sıfır seviye kümesi C eğrisi şeklinde tanımlanmıştır. Kapalı fonksiyon, φ işaretli uzaklık fonksiyonu iken, C fonksiyonu olarak tanımlanır. Izgara üzerindeki her bir noktanın değeri, o noktanın C eğrisine uzaklığıdır. Eğer nokta eğrinin içinde ise negatif değerler, eğer nokta eğrinin dışında ise pozitif değerler, eğer nokta C eğrisinin üzerinde ise 0 değerini almaktadır.



Şekil 3.2. φ işaretli uzaklık fonksiyonu ve değerleri

Şekil 3.2'de görüldüğü gibi seviye kümesi yöntemi 2 boyutlu problemler için kullanılabildiği gibi daha yüksek boyutlu yapılarda da kullanılabilir.

Seviye kümesi yöntemi dengeli ve yakınsaktır. Dengeli olmasının nedeni, sayısal hataların toplanıp büyük hatalara sebebiyet vermemesidir. Yakınsak olmasının nedeni ise çözümün belli sayıda adımın sonunda sabitleşmesidir [33].

3.2.1. Dışardan Oluşturulan Hız Terimi ile Hareket

$\varphi(\vec{x}) = 0$ olduğu tüm $\vec{x}$ noktaları için $\vec{V}(\vec{x})$ hız terimi ifadesinin bilindiği düşünülürse, yüzey üzerindeki tüm noktalar $\vec{V} = < u, v, w >$ hız terimi ile hareket ettirilebilir. Bunu yapmak için en basit yöntem denklem (3.3)'de verilen adi diferansiyel denklemin her bir $\vec{x}$ noktası için çözülmesidir.

$$\frac{d\vec{x}}{dt} = \vec{V}(\vec{x}) \tag{3.3}$$

Verilen denklemin çözümü için öncelikle ayrıklaştırılması gerekmektedir, çünkü eğri üzerinde sonsuz nokta bulunur ve her bir nokta için çözümü sağlamak imkânsızdır. Ayrıca bu işlem her bir zaman adımında doğru sayısal çözümü elde etmek için ayrıklaştırma işlemi tekrar sağlanmalıdır. Eğrinin 2 boyutlu olduğu durumlarda bölümlere (segment) ve üç boyutlu olduğu durumlarda ise üçgenlere bölerek ayrıklaştırılabilir. Fakat çok basit bir hız terimi kullanıldığında bile, büyük oranda bozulma meydana gelmektedir. Eğer periyodik olarak ayrıklaştırma düzeltilmezse, yöntemin doğruluğu çok kolay bir şekilde bozulmaktadır [31].

Bu bozulmadan kaçınmak için, φ kapalı fonksiyonu hem eğriyi temsil etmek için hem de eğri gelişimi için kullanılabilir. φ kapalı fonksiyonun gelişimini tanımlamak için aşağıdaki basit bir taşınım denklemi kullanılır.

$$\varphi_t + \vec{V}.\nabla\varphi = 0 \tag{3.4}$$

Bu denklemde t ifadesi, t zaman değişkenine göre zamansal kısmi türevi ifade ederken, ∇ ise eğriliği ifade etmektedir. $\nabla\varphi = < \varphi_x, \varphi_y, \varphi_z >$ olduğu düşünüldüğünde ve

eğrilik ifadesi açık bir şekilde yazıldığında denklem (3.5) elde edilir. Denklemin ayrıklaştırma işlemi ise ileri ve geri Euler yöntemi ile yapılmaktadır.

$$\vec{V}.\nabla\varphi = u\varphi_x + u\varphi_{y+}u\varphi_z \quad \textbf{(3.5)}$$

Denklem (3.5)'in ayrıklaştırılması ileri ve geri Euler yöntemi ile yapılmaktadır. Denklem (3.6)'da elde edilen denklem verilmiştir [33].

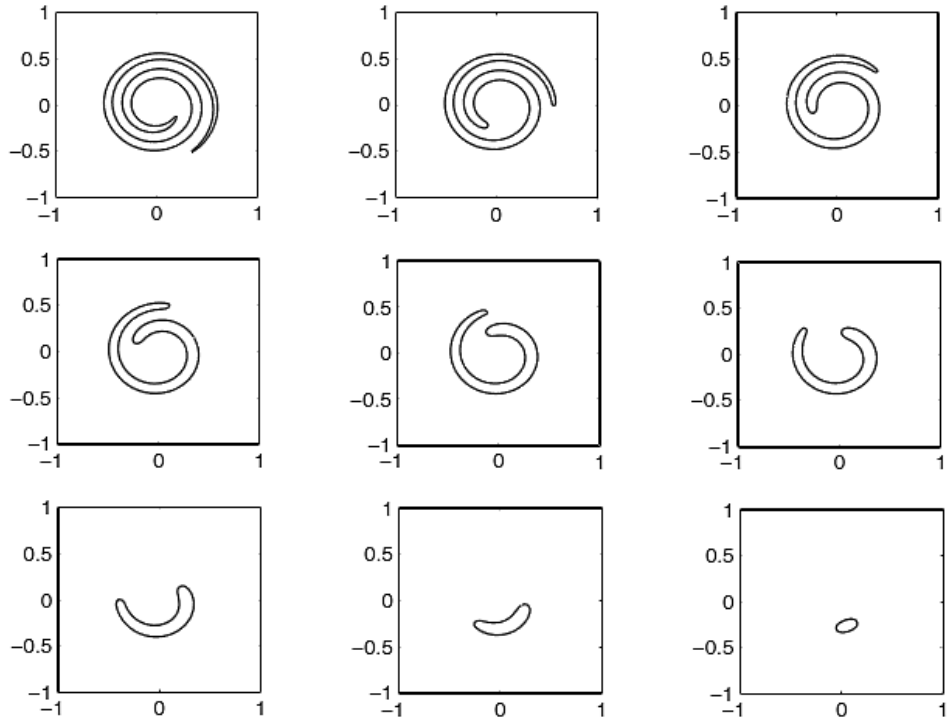
$$\frac{\partial\varphi}{\partial x} = \frac{\varphi_{i+1} - \varphi_i}{\Delta x} \quad \textbf{(3.6)}$$

Verilen denklem birinci dereceden ileri fark denklemidir. Yapılan işlem $D^+\varphi$ şeklinde sembolize edilir ve "upwinding" olarak isimlendirilir. Upwinding yöntemi ile daha iyi sonuçlar elde edilebilir. Daha iyi sonuçların elde edilmesi 2. 3. ve 5. dereceden yaklaşımlarla mümkündür [33].

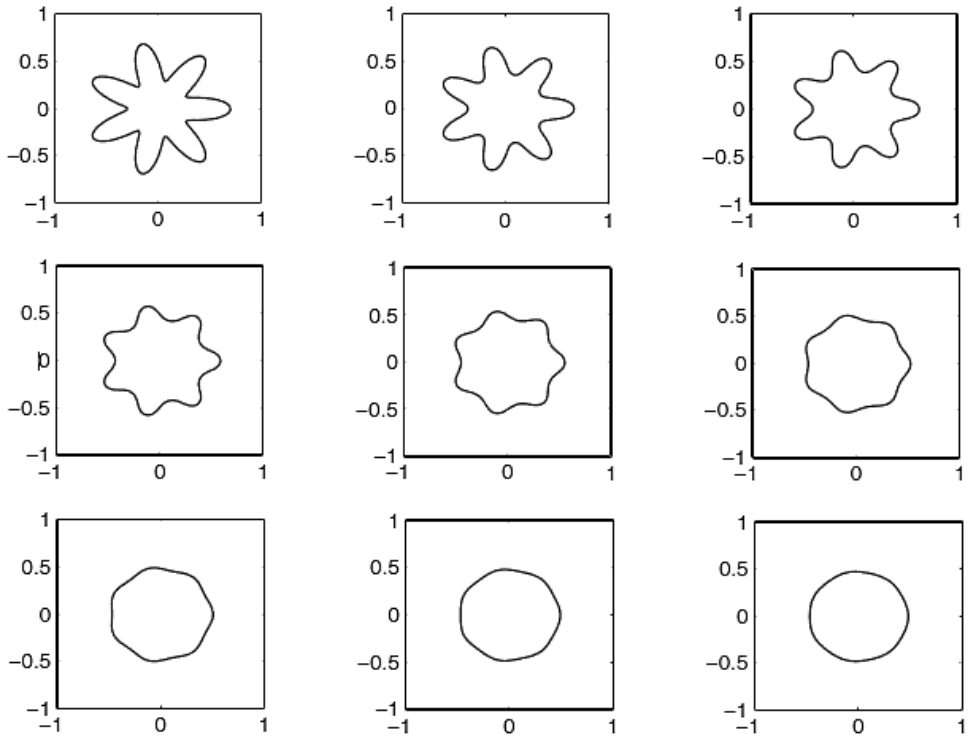
3.2.2. Ortalama Eğrilik ile Hareket

Bir önceki bölümde dışardan oluşturulan $\vec{V}(\vec{x},t)$ hız terimi ile eğri gelişimi sağlanmıştır. Bu bölümde, φ seviye kümesi fonksiyonuna bağlı olarak kendi kendine üretilen $\vec{V}$ hız terimi ile eğri gelişimi sağlanacaktır. Bu durumda eğri gelişimi, her bir noktanın eğriliğine bağlı olarak dikey doğrultuda değişim gösterir. Hız ifadesi $\vec{V} = -bk\vec{N}$ şeklinde ifade edilir. Burada b sabittir ve sıfırdan büyüktür, k ise eğriliği ifade etmektedir [31].

Bu yöntemde eğriliği yüksek olan noktalar hızlı hareket ederken, eğriliği az olan noktalar yavaş hareket etmektedirler. Böylece eğri kenarları yumuşar ve daha düzgün hale gelir. En sonunda çember halini almaktadır. Şekil 3.3'te spiral şeklindeki eğrinin gelişimi verilmiştir. Spiralin bittiği yüksek eğriliğe sahip noktalar, spiralin gövde kısmına göre nispeten daha hızlı hareket etmektedir. Şekil 3.4'te ise yıldız şekilli eğrinin eğri gelişimi verilmiştir. Yıldızların uç kısımları içe doğru hareket ederken, uçlar arasındaki bölgeler dışa doğru hareket etmektedir.



Şekil 3.3. Spiralin eğri gelişimi [31].

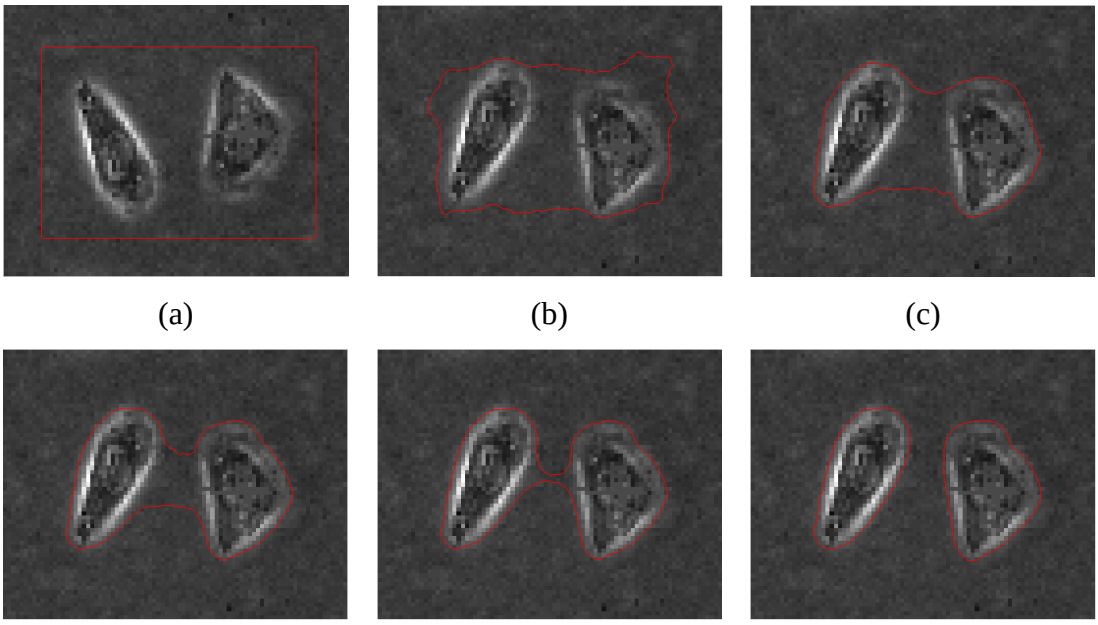


Şekil 3.4. Yıldız şeklindeki eğrinin gelişimi [31].

3.2.3. Seviye Kümesi Yöntemi ve Görüntü Üzerine Uygulanması

Seviye kümesi yönteminin dezavantajı eğri gelişiminde kullanılan kısmı diferansiyel denklemin çözümünün uzun sürede tamamlanmasıdır. Şekil 3.5'te seviye kümesi algoritmasının hücreler üzerine uygulaması gösterilmiştir.

Bunun sonucunda seviye kümesi yöntemi gerçek zamanlı uygulamalar için imkânsız hale gelmiştir. Fakat kısmı diferansiyel denklemler yerine, pikseller üzerinde çalışılır ve şekli bulmada, şekil ile arka plan arasındaki renk geçişi kullanılırsa çok daha hızlı çözümler elde edilebilecektir.



Şekil 3.5. Seviye kümesi algoritmasının örnek bir resim üzerine uygulanması. (a) Başlangıç durumu. (b),(c),(d),(e) Konturün gelişmesi, (f) Sınırların bulunması [34].

3.3. Sonuç

Tez çalışmasının bu bölümünde, öncelikle deforme şablonlar hakkında bilgi verilmiş ve eksiklikleri belirtilmiştir. Deforme şablonlar istenilen şekli bulabilmesine rağmen bulunacak şeklin yapısının bilinmesi gerektirir. Ayrıca şeklin birçok parametresi olabilmekte ve bu da birçok olasılık anlamına gelmektedir. Deforme şablonun

eksikliklerine çözüm olarak seviye kümesi yöntemi ortaya konulmuştur. Bu modelin yapısı ise PDE çözülmesi ile eğri gelişiminin sağlanmasına dayanmaktadır. PDE çözülmesi işlemi için Hamilton-Jacobi gibi birçok sayısal yöntem olmasına rağmen, bu yöntemler yüksek hesaplama gerektirmektedir. Daha sonra yapılan çalışmalarda bu probleme çözümler aranmış ve çeşitli yapılar geliştirilmiştir (dar bant ve seyrek alan tekniği gibi). Bu yapılar ile de yeterli hızlanma sağlanamamasından dolayı, son çalışmalar seviye kümesi yönteminin paralel çalıştırılmasına yönelmiştir.

4. GELİŞTİRİLEN YÖNTEM

Üçüncü bölümde eğri gelişimi için kullanılan seviye kümesi yönteminin kısa bir tanıtımı verilmiş ve bu yöntemin dezavantajlarından bahsedilmiştir. Bu bölümde geliştirilen algoritmanın CPU versiyonu verildikten sonra GPU versiyonu verilecek ve eğri gelişimini nasıl hesapladığı CUDA tabanlı olarak anlatılacaktır.

4.1. Ölçeklendirme Tabanlı Seviye Kümesi Yöntemi

Bu bölümde imgeler üzerindeki şekilleri bulmada seviye kümesi algoritmasından farklı bir bakış açısıyla GPU tabanlı bir program geliştirilmiştir. Uygulama Visual Studio 2012 ortamında CUDA C dili kullanılarak gerçekleştirilmiştir. Geliştirilen yöntem bölüm 4.1.1. ile bölüm 4.1.5 arasında verilen 5 temel aşamadan oluşmaktadır. Bu aşamalar ve her bir aşamanın çalıştırıldıkları donanımlar Şekil 4.1'de sunulmuştur.

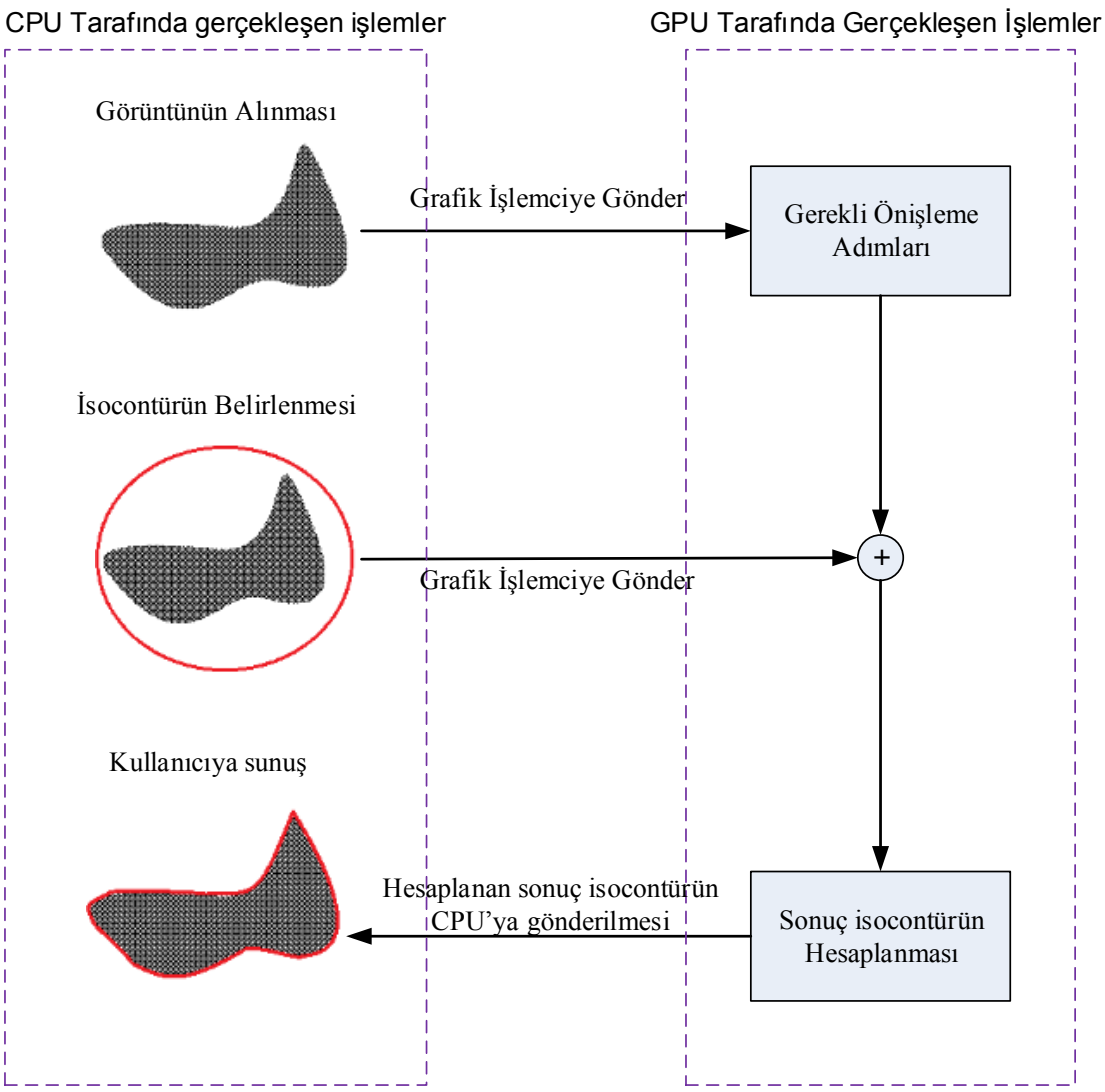
4.1.1. Hazırlık Aşaması

Geliştirilen arayüz ile kullanıcının istediği resmi alması sağlanmıştır. Alınan resmin kullanıcıya gösterilmesi OpenGL kütüphanesinde yararlanılmıştır. Görüntü alındıktan sonra önişleme işlemi için grafik işlemcisine gönderilmiştir.

4.1.2. Önişleme

Verilen görüntü üzerinde bazı önişleme adımları uygulanmıştır. Önişleme aşamasında yapılan işlemlerden biri gürültü azaltmadır. Görüntü üzerindeki doğal olarak oluşan gürültüleri azaltmada ve görüntü üzerinde bulunan keskin geçişlerin yumuşatılmasında 5x5 boyutunda Gaussian filtre kullanılmıştır. Gerçekleştirilen bir diğer işlem ise, Bölüm 4.1.4'te bahsedilen "*KENAR*" algoritmasına yardımcı olmak amacıyla Sobel kenar bulma algoritması kullanılmıştır. Son olarak yapılan çalışma gri seviye görüntüler üzerine uygulanmıştır. Renkli resimler için gri seviye dönüşümü önişleme kısımda yapılmıştır. Yukarıda bahsedilen tüm işlemler (filtreler, dönüşümler) GPU üzerinde çalıştırılmıştır.

Burada karşılaşılan asıl zorluk, sıralı bir şekilde tasarlanan algoritmaların paralel bir şekilde CUDA C diline uygun olarak geliştirilmesidir.



Şekil 4.1. Geliştirilen uygulamanın genel algoritması

4.1.2.1. Gri Seviye Dönüşümü

Genel olarak bir görüntüde her bir renk bileşeni 8 bitlik alanlarda saklanır. Bu ise RGB düzeyindeki bir pikselin 24 bit ile ifade edilmesi sağlar. Eğer görüntü RGBA ise pikseller 32 bitlik alanlarda saklanır. Gri seviye resimler ise 8 bit ile ifade edilebilir. Geliştirilen uygulamada gri seviye resimlerde çalışılması tercih edilmiştir.

Renkli bir görüntünün gri seviye bir görüntüye dönüştürmek için denklem (4.1)'de verilen formül kullanılmıştır. Böylece 24 birlik bir görüntü 8 bit ile ifade edilmiştir. Burada elde edilen her bir pikselin değeri 0 ile 255 arasında değişmektedir.

$$Gri\ Seviye\ Görüntü = R.0{,}299 + G.0{,}587 + B.0{,}114 \quad \textbf{(4.1)}$$

BU işlem GPU tarafında çalıştırılmış ve çok hızlı bir şekilde sonuç elde edilmiştir. Bu işlem sırasında tüm pikseller sırasıyla gezilmesi gerekmektedir. Bu işlem CPU tarafında döngüler ile yapılır fakat GPU tarafında döngülere gereksinim duyulmaz. Her bir kanal bir pikselin hesaplamasından sorumlu tutularak işlem gerçekleştirilir. Geliştirilen uygulama 512^2 görüntülerde test edilmiştir. Günümüz GPU'larda paralel çalışabilecek kanal sayısının 64 milyona yakın olduğu düşünüldüğünde 262144 piksel çok kolay bir şekilde kanallara bölünebilir. Böylece çalışma zamanı O(1) seviyesine düşmektedir. Şekil 4.2'de renkli bir görüntünün gri seviye dönüşümü için kullanılan CUDA fonksiyonu verilmiştir.

```
__global__ void gri_seviye(unsigned char *ptr, unsigned char *gri){
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    float t=0;
    t += (float)ptr[offset*3] * 0.299;
    t += (float)ptr[offset*3+1] * 0.587;
    t += (float)ptr[offset*3+2] * 0.114;

    gri[offset] = (unsigned char)t;
}
```

Şekil 4.2. Gri seviye dönüşümü için CUDA fonksiyonu

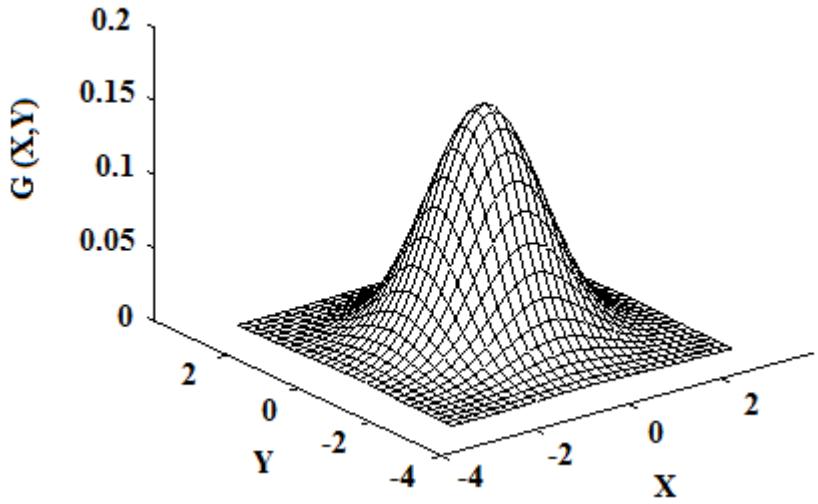
4.1.2.2. Gürültü Azaltma

Resimlerde gürültü resimlerdeki küçük kusurlar şeklinde isimlendirilebilir. Gürültüyü ışık durumu, sıcaklık ve değişik kamera modları etkileyebilir. Resimlerdeki gürültüden tamamen kurtulmak imkânsızdır fakat etkisi gürültü temizleme işlemi ile azaltılabilir [32].

Gürültü temizle işlemi, median filtre, average filtre ve gaussian filtre gibi birçok yöntemle gerçekleştirilebilir. Bu tez çalışmasında 5x5 boyutunda gaussian filtre kullanılmıştır.

İki boyutlu düzlemde Gaussian fonksiyonu aşağıdaki şekilde elde edilmektedir [35]. Dağılımı ise Şekil-4.3'te gösterilmiştir.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad \textbf{(4.2)}$$



Şekil 4.3. 2 boyutlu Gaussian Dağılımı (ortalama (0,0) ve $\sigma = 1$) [35].

Teori olarak Gaussian dağılımı sonsuza gitmektedir ve tüm değerleri sıfırdan farklıdır. Bu ise sonsuz büyüklükte bir filtre gerektirmektedir. Pratikte ise gaussian dağılımı belli bir standart sapmadan sonra kesilmektedir. Sekil 4.4'te verildiği gibi uygun bir şablon oluşturulduktan sonra, Gaussian filtre standart filtreleme işlemleri gibi kullanılabilmektedir.

Tez için yapılan uygulamada Gaussian filtre algoritması paralel olarak geliştirilmiştir. Şablon olarak ise Şekil 4.4'te verilen şablon kullanılmıştır. Bu şablon programlama kısmında sabit belleğe alınmıştır. Böylece önbellek ile şablona hızlı bir erişim sağlanmıştır.

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Şekil 4.4. Gaussian fonksiyonunun ayrıklaştırılmış hali ($\sigma = 1$) [35].

4.1.2.3. Sobel Kenar Bulma

Dijital bir görüntüde kenar bulma, bazı matematiksel metotların görüntüdeki yoğunluk değişiminin keskin olduğu yerlerin belirlenmesi şeklinde tanımlanır. Eğer bir noktada yoğunluk değişimi yüksekse, bu noktanın kenar adı verilen bir eğri üzerinde bulunduğu söylenebilir. Görüntü işlemede kenar bulma algoritması için birçok yöntem geliştirilmiştir. Bu yöntemlerden birisi olan sobel kenar bulma algoritması en yaygın kullanılan kenar bulma algoritmalarından biridir. Popüler olmasının nedeni, örneğin Prewitt kenar bulma algoritması gibi diğer kenar bulma algoritmalarından genel olarak daha performanslı çalışmasıdır [32].

Sobel kenar bulma algoritması 2 şablonun resim üzerinde gezdirilmesi ile bulunmaktadır. Şekil 4.5'de tez çalışmasında kullanılan Sobel şablonları verilmiştir. Sobel kenar bulma algoritmasında büyük boyutlu şablonların oluşturulması paskal üçgeni ve fark fonksiyonu ile mümkün olabilmektedir. Tüm şablon tabanlı teknikler 5x5 ve 7x7 gibi büyük boyutta şablonlar uygulanmaktadır. Sobel metodunda şablonu büyütmek gürültü azalmayı sağlar buna karşın kenarların bulanıklaşması büyük problem oluşturmaktadır.

1	0	-1
2	0	-2
1	0	-1

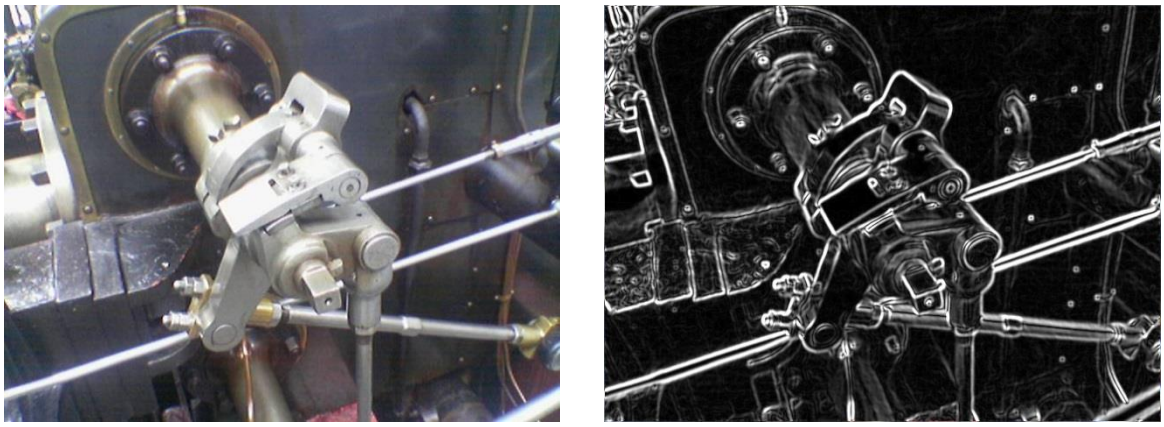
(a) Yatay Sobel Şablonu

1	2	1
0	0	0
-1	-2	-1

(b) Dikey Sobel Şablonu

Şekil 4.5. Sobel kenar bulma algoritmasında kullanılan şablonlar [32].

Şekil 4.5(a)'da verilen şablon yatay kenarları bulmada kullanılırken Şekil 4.5(b)'de verilen şablon dikey kenarları bulmada kullanılmaktadır. Bu şablonlar yardımıyla her bir piksel için yoğunluk değerleri hesaplanır. Genel yoğunluk değeri ise bu iki yoğunluğun karelerinin toplamı şeklinde bulunmaktadır. Sobel kenar bulma algoritmasının örnek bir sonucu Şekil 4.6'da verilmiştir.



Şekil 4.6. Sobel kenar bulma algoritmasından örnek bir sonuç

Tez için geliştirilen uygulamada Sobel kenar bulma algoritması paralel olarak geliştirilmiştir. Daha sonra bulunan Sobel ağırlıkları belirli bir eşik değerinden geçirilerek kenarlar belirlenmiştir. Burada belirlenen kenarlar Bölüm 4.1.4'te bahsedilen "KENAR" fonksiyonunda kullanılmaktadır.

4.1.3. Konturün ve Merkezin Belirlenmesi

Konturün alınması nesnenin hızlı bulunması için oldukça önemlidir. Geliştirilen uygulama MRI beyin tümörleri üzerinde test edilmiştir. Burada uzman kişinin tümörlü alanı kabataslak belirlemesi beklenmektedir. Belirlenen kontur eğrisi başlangıçta çember olarak çizilmektedir. Daha sonra bu eğri grafik işlemcisine gönderilerek eğri gelişimi sağlanmıştır.

Eğri gelişim algoritmasının temel elemanı olan merkez bilgisi uzman tarafından alınabileceği gibi sistem tarafından otomatik olarak da ayarlanabilmektedir. Tez çalışmasında öncelikli olarak tek merkez üzerinde durulmuştur. Bu yaklaşımda nesne hakkında bilgi sahibi olmadığımız için, kontur noktalarının aritmetik ortalaması olarak seçilmiştir. Ortalama aşağıdaki denklemlere göre hesaplanmaktadır.

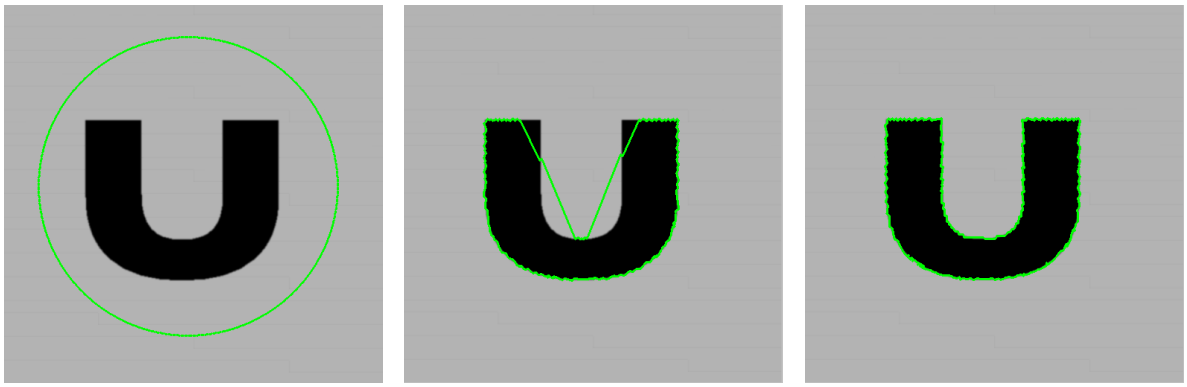
$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \textbf{(4.3)}$$

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \qquad \textbf{(4.4)}$$

Denklemlerde kullanılan $\bar{x}$, noktaların x koordinatlarının aritmetik ortalaması $\bar{y}$ ise noktaların y koordinatlarının aritmetik ortalaması, n ise konturde bulunan toplam nokta sayısıdır. Dolayısıyla merkez nokta $M(\bar{x}, \bar{y})$ şeklinde gösterilebilir. Bu yaklaşımın avantajı sistem tarafından kolayca hesaplanabilmesi ve kullanıcı girişlerinin azaltılması şeklinde sıralanabilir. Fakat bu yaklaşım her zaman doğru sonuç vermemektedir. Çizilen konturün nesne ile tam uyuşmadığı durumlarda, hesaplanan merkez nokta nesnenin dışına düşmektedir. Bu ise nesnenin tam olarak bulunamamasına yol açmaktadır.

İkinci yöntem olarak ise merkez bilgisinin uzman tarafından alınmasıdır. Böylelikle doğru bir merkez belirlenmesi sağlanmıştır. Burada karşılaşılan zorluk ise iç bükey resimlerde tam olarak sonuçların hesaplanamaması şeklindedir. Böylelikle birden fazla merkezin seçilmesine izin verilerek bu sorunda çözülmüştür. Örneğin "u" şeklinde bir nesnenin sınırları bulunmak istendiğinde, tek merkez kullanıldığında Şekil 4.7(b)'daki

sonuç elde edilmektedir. Görüldüğü gibi nesne başarılı bir şekilde bulunamamıştır. Fakat birden fazla merkez kullanıldığında, Şekil 4.7(c)'deki başarılı sonuç elde edilmektedir.



(a) Başlangıç Durumu

(b) Tek merkez kullanıldığında elde edilen sonuç

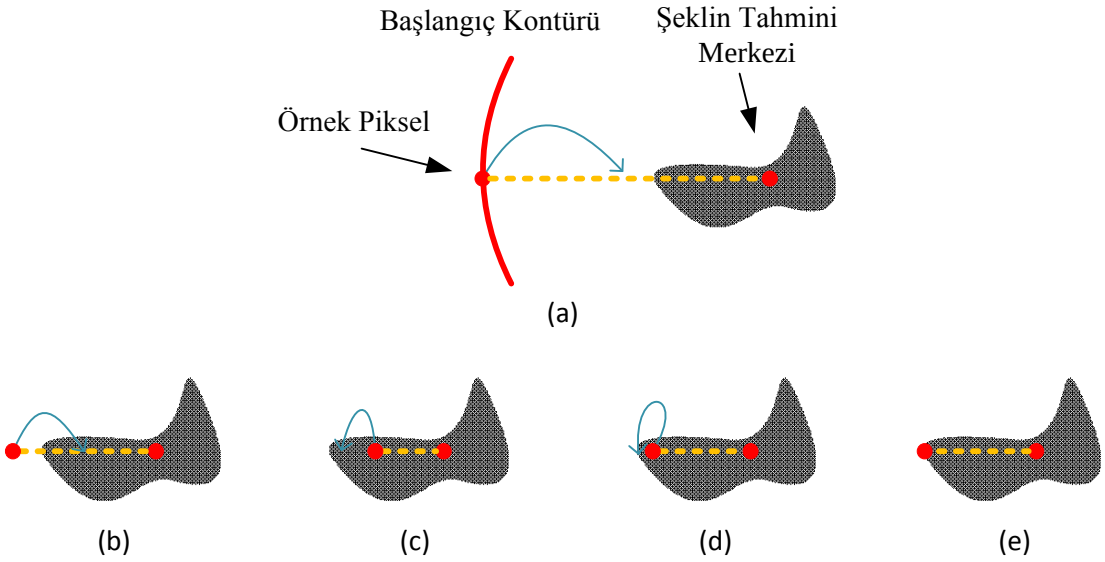
(c) 3 merkez kullanıldığında elde edilen sonuç

Şekil 4.7. U şekilli nesne üzerinde tek merkez ve çoklu merkez kullanımının karşılaştırılması

4.1.4. Konturün Geliştirilmesi

Seviye kümesi yönteminde eğri gelişimi denklem (3.3)'te verilen kısmi diferansiyel denklemin çözümüne dayanmaktadır. Bu ise algoritmanın yavaş çalışmasına neden olmaktadır. Tez için geliştirilen algoritmada eğri gelişimi farklı bir yöntem izlenerek elde edilmektedir.

Geliştirilen yöntemde eğri gelişimi için temel olarak ikili arama mantığı kullanılmıştır. İkili arama, sıralı sayı dizisi üzerinde çalıştırılan ve dizi içerisinde aranan elemanın çok daha hızlı bir şekilde bulunmasını sağlayan bir algoritmadır. Geliştirilen algoritmada ise arama işlemi, sayılar üzerinde değil pikseller üzerinde yapılmıştır. İkili aramadaki sayı kümesi yerine piksel kümesi tanımlanmış ve her bir döngüde kümenin yarısı elenmiştir. Bu işlem kenar bulunana kadar veya kümede eleman kalmayana kadar devam ettirilmiştir. Piksel kümesinin elemanları ise, konturdeki bir piksel ile şeklin merkezi arasında kalan pikseller şeklinde tanımlanmıştır.



Şekil 4.8. Konturdeki bir pikselin hareketi. (a) Başlangıç durumu. (b) Birinci adım sonu. (c) İkinci adım sonu. (d) Üçüncü adım sonu. (e) Dördüncü adım sonu – Kenarın bulunma durumu

Şekil 4.8'de algoritmanın çalışması bir piksel için gösterilmiştir. Şekil 4.8(a)'da gösterilen örnek pikselin hareketi Şekil 4.8 (b,c,d,e)'de gösterilmiştir. Böylece piksel piksel ilerlediğimizde kenar bulma işlemi O(n) sürüyorken, geliştirilen algoritma ile kenar bulma işlemi O(logn) sürecektir (n → ilk piksel ile kenar noktası arasındaki piksel sayısı). Bu işlem başlangıç konturü üzerindeki tüm noktalar üzerinde tekrarlanarak konturün gelişimi sağlanmaktadır.

Burada pikselin hareketini sağlamak amacıyla temel geometrik dönüşümlerden olan ölçeklendirme kullanılmaktadır. Başlangıç olarak her piksel için iki tane ölçeklendirme katsayısı tanımlanmıştır. Bu iki değişken "sr1" ve "sr2" şeklinde isimlendirilir ve başlangıçta sırasıyla değerleri 1 ve 2 olacak şekilde ayarlanmaktadır. "sr1" değişkeni bir önceki döngüde uygulanan ölçeklendirme katsayısını ifade ederken, "sr2" ise o döngüde uygulanan ölçeklendirme katsayısını ifade etmektedir. Başlangıçta ölçeklendirme işlemi uygulanmadığı için "sr1" değişkeni 1 olarak belirlenmiştir. "sr2" değişkeninin 2 olarak belirlenmesi ile de maksimum ölçeklendirme katsayısı belirlenmiştir. Her döngü sonunda bu iki değişken her pikselin hareketlerine göre aşağıdaki formüle göre hesaplanmaktadır.

$$sr1 = sr2 \tag{4.5}$$

$$sr2 = \begin{cases} sr2 + \dfrac{|sr1 - sr2|}{2} & Piksel\ noktası\ şeklin\ içinde\ ise \\ sr2 - \dfrac{|sr1 - sr2|}{2} & Piksel\ noktası\ şeklin\ dışında\ ise \end{cases} \tag{4.6}$$

Şekil 4.9'da geliştirilen algoritmanın GPU versiyonu verilmiştir. Algoritmada iki adet fonksiyon kullanılmıştır. Kullanılan birinci fonksiyon "KENAR" fonksiyonudur. Bu fonksiyon pikselin bir kenara yakın olup olmadığı hakkında bilgi vermektedir. Kullanılan bir diğer fonksiyon ise "ICINDE" fonksiyonudur. Bu fonksiyon pikselin, şeklin içinde mi, dışında mı olduğunu belirlenmesinde kullanılmaktadır.

```
Giriş: img  {Resim},
       C    {Kontur Noktaları},
       M    {Şeklin Merkezleri}

Çıkış: s    {Sonuç Konturü}

for all pixels in C do
    midx ← Cx + Mx
    midy ← Cy + My

    while (C != mid or M != mid)

        if KENAR(img, mid) then
            mid değişkenini s listesine ekle  {Artık sabit }
            break

        else if ICINDE(img, mid) then
            M = mid

        else
            C = mid
```

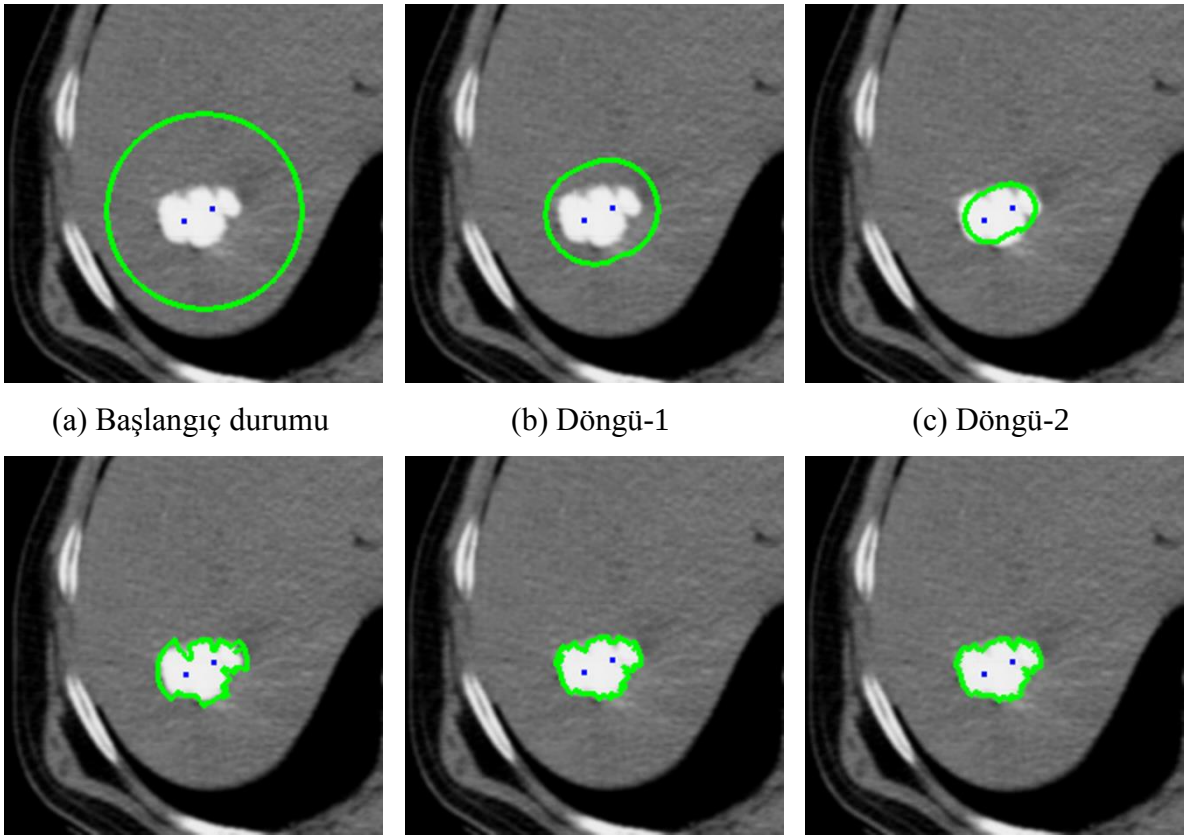
Şekil 4.9. Geliştirilen algoritmanın GPU versiyonu

Verilen algoritmada kullanılan "*KENAR*" fonksiyonu Bölüm 4.1.2.3'de hesaplan kenar bilgisi kullanılarak 0 ve 1 şeklinde sonuç döndürmektedir. Aynı şekilde "*ICINDE*" fonksiyonu da 0 ve 1 şekilde sonuç döndürmektedir. "*ICINDE*" fonksiyonu merkez değerleri, ilk kontur noktalarına ve kontur içindeki ortalama yoğunluğa bakarak karar vermektedir.

Geliştirilen GPU tabanlı uygulama CPU tabanlı algoritmayla benzerlik göstermektedir. Grafik işlemciler paralel işlemleri çalıştırmada çok başarılıdır. Fakat eğer doğru bir şekilde paralel hale getirilmezlerse, sonuçlar CPU versiyonuna göre çok fazla hız artışı sağlamamaktadır. Geliştirilen yöntemde kontur üzerindeki her bir piksel, bir CUDA kanal üzerinde çalıştırılarak uygulama paralel hale getirilmiştir. Ayrıca fonksiyonlar da CUDA kernel şeklinde yeniden tasarlanmıştır. Şekil 4.10'da geliştirilen algoritmanın GPU versiyonu verilmiştir.

```
Giriş: img  {resim},
       Tid  {kanal id},
       C    { ilgili kanalda işlenecek kontur piksel},
       M    {C'ye en yakın merkez noktası}

Çıkış: s   {sonuç konturu}

Y ← C
mid_x ← C_x + M_x
mid_y ← C_y + M_y

while (Y != mid || M != mid)

    if KENAR(img, mid) then
        mid değişkenini s listesine ekle  {Artık sabit }
        break

    else if ICINDE(img, mid) then    { Nokta nesnenin içinde mi?}
        M = mid

    else
        Y = mid
```

Şekil 4.10. Geliştirilen algoritmanın GPU versiyonu



(a) Başlangıç durumu (b) Döngü-1 (c) Döngü-2

(d) Döngü-3 (e) Döngü-5 (f) Döngü-7

Şekil 4.11. Eğri gelişim algoritmasının çalıma örneği

Eğri gelişim algoritmasının çalışması Şekil 4.11'de gösterilmiştir. Şekilde kullanılan görüntü bir karaciğer görüntüsüdür ve 512x512 boyutundadır. Görüntülerde görüldüğü gibi eğri gelişimi çok hızlı bir şekilde ilerlemektedir. Seviye kümesi yönteminde yüzlerde döngü sonucunda eğri gelişimi tamamlanırken, geliştirilen yöntemde yaklaşık 8 döngü sonunda eğri gelişimi tamamlanmaktadır.

4.1.5. Sonuçların Gösterilmesi

Bu aşamada tüm hesaplama işlemleri tamamlandıktan sonra else edilen sonuç konturu ve gaussian filtresinden geçirilen resim CPU belleğine kopyalanmıştır. Daha sonra görüntü ile kontur noktaları birleştirilerek kullanıcıya sunulmuştur.

4.2. Sonuçlar

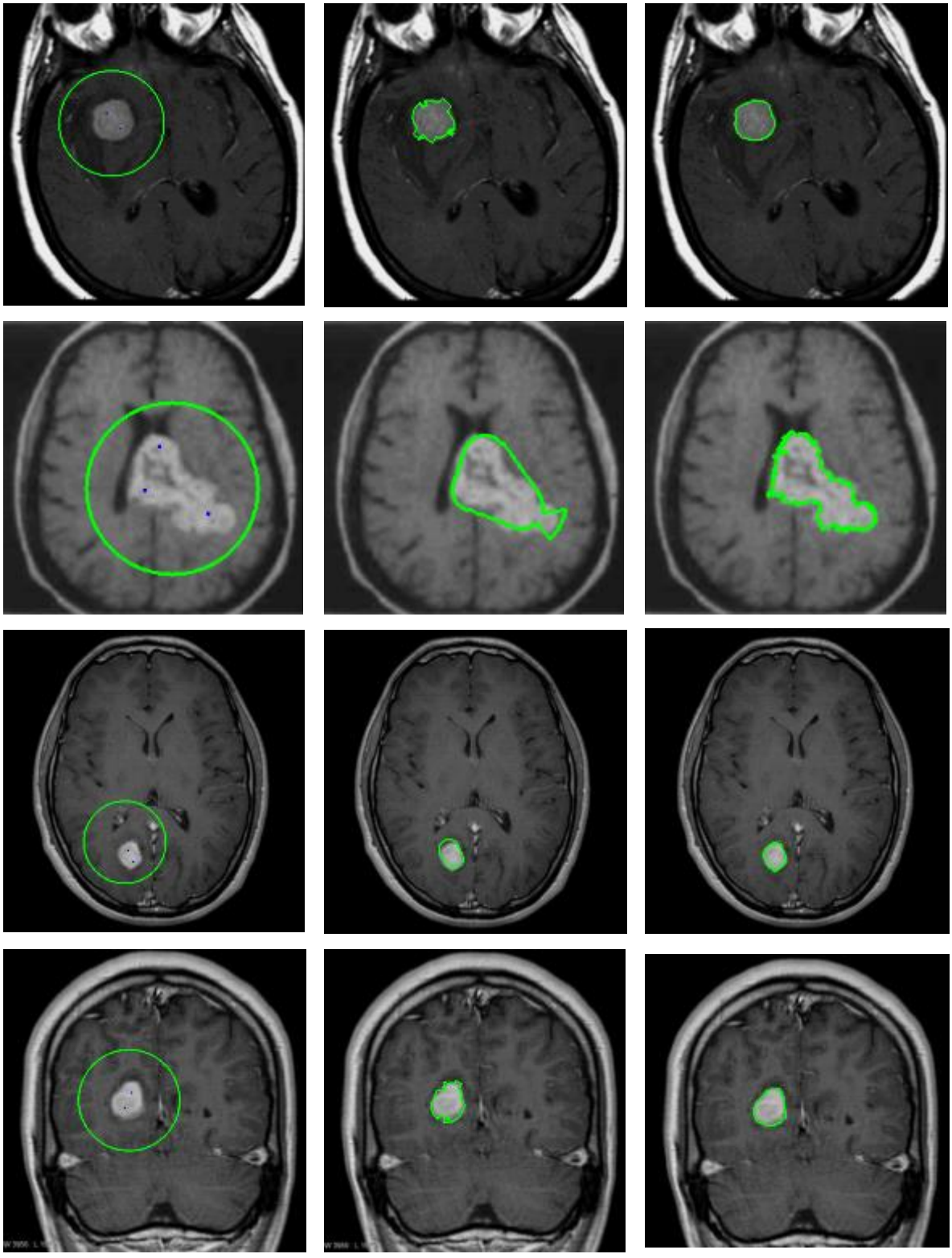
Bu çalışma Intel® Core™ i7-3770 CPU üzerinde 8GB belleğe sahip bilgisayarda çalıştırılmıştır. Ekran kartı olarak NVIDIA GeForge GTX 660 Ti ekran kartı kullanılmıştır. Bu karta 7 adet streaming multiprocessor (SM) bulunmaktadır ve her bir SM'de 192 adet CUDA işlemcisi bulunur. Dolayısıyla ekran kartında toplam 1344 adet CUDA core işlemcisi bulunmaktadır. Her bir SM 65536 kayıtçı içermektedir ve toplam sabit bellek boyutu 64KB'dır. Her bir SM başına paylaşılan bellek miktarı 48KB'dır ve bunlar 32 bank içinde organize olmuşlardır. 2GB kapasitesindeki global belleğe GDDR5 arayüzü aracılığı ile ulaşılabilmektedir ve double sayıları desteklemektedir.

Çalışmada 512x512 boyutundaki MRI beyin tümör resimleri kullanılmıştır. Uygulamaya bakıldığı zaman iki ana kernel fonksiyonu kullanılmıştır. Bunlardan birincisi "onisleme" fonksiyonudur. Bu fonksiyonda gri seviye dönüşümü, Gaussian filtre ve Sobel kenar bulma algoritması gibi önişleme aşamasının işlemleri yapılmaktadır. Burada temel mantık görüntü üzerindeki her bir pikselin bir kanal üzerinde çalıştırılmasıdır. Burada performansa etki eden faktörlerden birisi kanal ve blok sayılarıdır. Bu iki değişkenin doğru bir şekilde belirlenmesi performans açısından çok büyük etkileri olabilmektedir. Görüntünün bir matris olduğu düşünüldüğünde blok ve kanal sayısının da 2 boyutlu olması düşünülebilir. Geliştirilen algoritmada blok sayısı 32x32 şeklinde iki boyutlu belirlenmiştir. Her bir blokta çalışacak kanal sayısı ise 16x16, toplam 256 şeklinde belirlenmiştir. İkinci fonksiyon ise "gelisim" fonksiyonudur. Bu fonksiyonda bir kontur noktası için belirlenen merkez noktaya veya noktalara göre yapılacak işlemler bir kanal üzerinde yapılmaktadır. Toplam kontur noktası, piksel sayısının 256 da biri şeklinde belirlenmiştir. 512x512 resimler için kontur nokta sayısı 1024'tür. Dolayısıyla blok sayısı 4x4 boyutunda seçilirken, her bir bloktaki kanal sayısı 8x8 şeklinde belirlenmiştir. Blok ve kanal sayısı kullanılacak resme göre değişiklik göstermektedir.

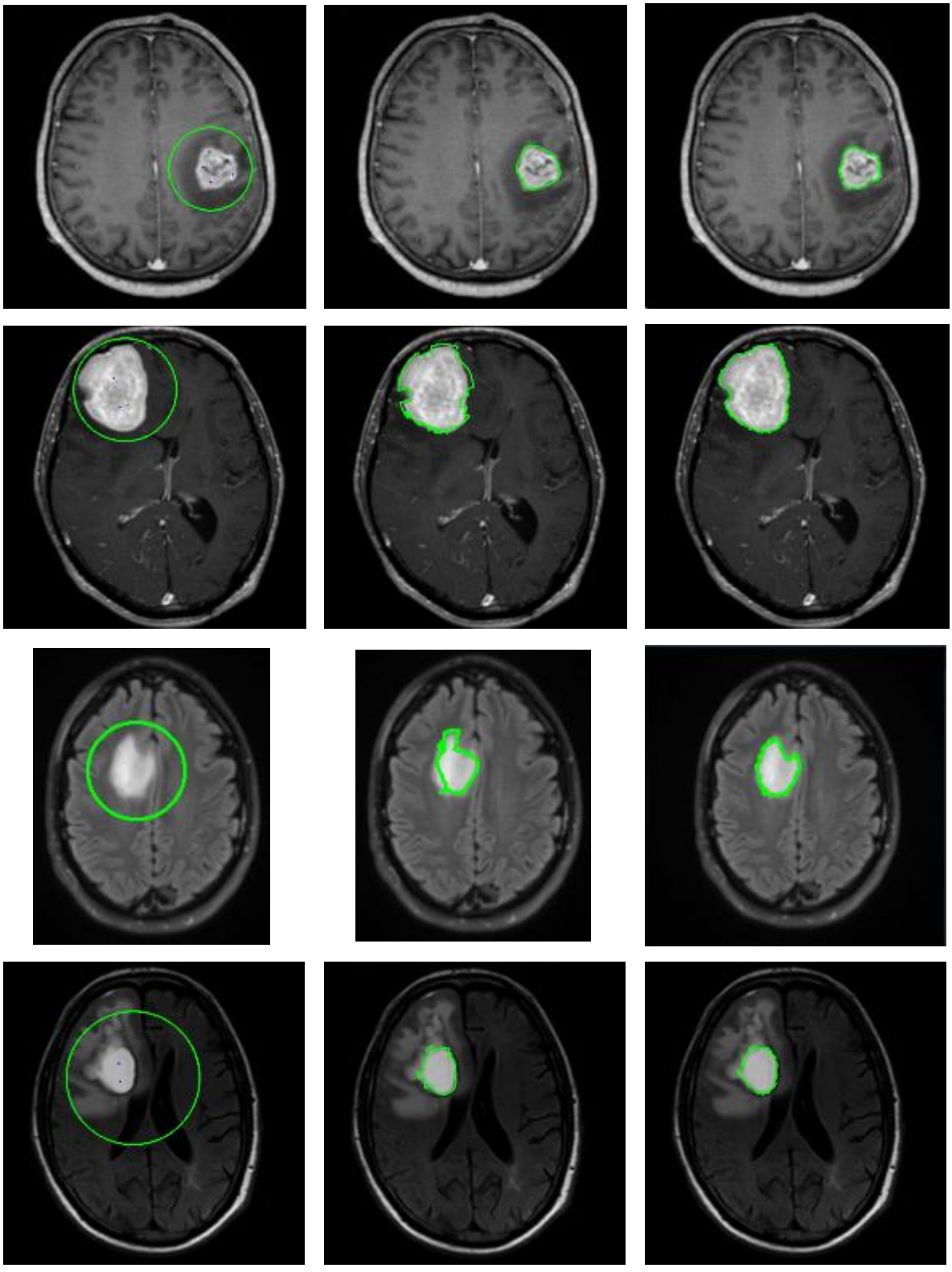
GPU üzerinde çalıştırılan programlarda kesin bir süre ölçüme yapmak zordur. Geliştirilen uygulamada program 10 kez çalıştırılarak yaklaşık sonuçlar elde edilmiştir. Uygulamada önişleme aşaması tüm bellek kopyalama işlemleri dahil yaklaşık 1 milisaniye içerisinde tamamlanmaktadır. Eğri gelişim aşamasının çalışma süresi ise döngüsel bir yapıya sahiptir. Eğri gelişim aşamasında bir döngü yaklaşık 60 mikro saniye içerisinde tamamlanmaktadır. Tümörün bulunması ise tüm bellek kopyalama işlemleri dâhil yaklaşık

olarak 2 milisaniye içerisinde tamamlanmaktadır. Bu sonuçlardan da görüldüğü gibi bellek işlemleri uygulamanın çalışma süresi üzerinde önemli bir etkiye sahiptir.

Uzman tarafından çizilen kontur noktalarının boyutu ise görüntü boyutu ile orantılı olacak şekilde 1024 olarak belirlenmiştir. Burada kontur nokta sayısının fazla olması çözünürlüğü arttıracaktır fakat yapılacak işlemleri arttırdığı için çalışma zamanını artırır. Kontur nokta sayısının az olması ise çözünürlüğü azaltarak nesne bulma oranını azaltacaktır. Geliştirilen uygulamada kontur nokta sayısı toplam piksel sayısının 256'da biri olarak belirlenmiştir. Şekil 4.12 ve Şekil 4.13'te geliştirilen algoritmanın çalışma sonuçları gösterilmiştir. Çalışmada 512x512 boyutundaki MRI beyin tümör resimleri kullanılmıştır. Şekillerde de görüldüğü gibi birçok resimde yüksek başarı oranı ile tümör bulumu sağlanmıştır.



Şekil 4.12. Geliştirilen algoritma ile tümör bulma sonuçları – 1 [36].



Şekil 4.13. Geliştirilen algoritma ile tümör bulma sonuçları – 2 [36].

4.3. Değerlendirme

Geliştirilen uygulama, seviye kümesi yönteminin ölçeklendirme yaklaşımı ile yeniden yorumlanmasıyla elde edilmiştir. Bu farklı yaklaşım bazı avantaj ve dezavantajları beraberinde getirmiştir. Geleneksel seviye kümesi yöntemine bakıldığı zaman PDE çözülmesi gerekmektedir, fakat topolojik değişimler otomatik olarak algılayabilir. Örneğin, başlangıç aşamasında verilen kontur eğrisi, görüntüye bağlı olarak parçalara ayrılabilir veya parçalar birleşebilir. Böylece yüksek oranda nesnenin bulunması sağlanır. Geliştirilen uygulamada ise PDE çözülmemektedir böylece yüksek oranda hız artışı sağlanmıştır. Fakat topolojik değişiklikler otomatik olarak algılanamamaktadır. İleriki çalışmalarda topolojik değişimlerin otomatik bir şekilde yakalanması sağlanmasıyla çok daha başarılı sonuçlar elde edilebilecektir.

Geliştirilen uygulama değişik boyuttaki resimler üzerinde değişik ayarlar ile denenmiştir. Öncelikli olarak önişleme işlemi, değişik boyutlardaki resimler üzerinde çeşitli blok ve kanal sayısı ile çalıştırılmış ve çalışma süreleri Tablo 4.1'de verilmiştir. Tablodan da görüldüğü gibi boyut arttıkça çalışma sürede artmaktadır. Bunun yanında seçilen ayarlar da çalışma süresini büyük oranda etkilemektedir. Tablo 4.1'de de görüldüğü gibi 256x256 boyutundaki görüntüde kanal sayısı 8x8 boyutunda seçildiğinde, çalışma süresi yaklaşık 2 kat artmaktadır. Aynı şekilde 1024x1024 boyutundaki görüntüde de kanal sayısı 8x16 boyutunda seçildiğinde çalışma süresi artmaktadır. Bunun nedeni görüntüdeki piksellerin kanallara eşit oranda bölüştürülememesinden kaynaklanmaktadır. Bu ise ekran kartının kapasitesinin kullanılmadığının göstergesidir. Kanal sayısının diğer ayarlarında ise önemli bir değişiklik görülmemiştir. Tabloda verilen blok sayıları, kanal sayısı ve piksel sayısı ile orantılı şekilde oluşturulmuştur. Her bir pikselin bir kanal üzerinde çalıştırıldığı düşünüldüğünde, kanal sayısı ile blok sayısının çarpımı piksel sayısına eşit olmaktadır.

İkinci olarak, eğri gelişim fonksiyonu değişik boyutlardaki görüntüler, çeşitli blok, kanal ve kontur nokta sayıları ile çalıştırılmış ve Tablo 4.2'deki sonuçlar elde edilmiştir. Eğri gelişim algoritmasının düşünüldüğünde bir döngüsel yapı mevcuttur. Tabloda ise sadece 1 döngü için harcanan zaman verilmiştir. Tabloda çalışma süresi, en yavaş çalışan döngüyü ifade etmektedir ve milisaniye cinsinden verilmiştir.

Tablo 4.1. Önişleme işleminin çalışma süresi

Boyut	Blok Sayısı	Kanal Sayısı	Önişleme Süresi (ms)
256x256	16x16	16x16 (256)	0,36
256x256	32x32	8x8 (64)	0,70
256x256	8x8	32x32 (1024)	0,36
256x256	16x8	16x32 (512)	0,36
512x512	16x16	32x32 (1024)	1,34
512x512	32x32	16x16 (256)	1,33
512x512	16x32	32x16 (512)	1,32
512x512	32x64	16x8 (128)	1,33
1024x1024	64x64	16x16 (256)	5,14
1024x1024	64x32	16x32 (512)	5,19
1024x1024	32x32	32x32 (1024)	5,19
1024x1024	128x64	8x16 (128)	5,36

Tablo 4.2. Eğri gelişim algoritmasının çalışma süresi (1 döngü için)

Boyut	Blok Sayısı	Kanal Sayısı	Kontur Nokta Sayısı	Çalışma süresi (ms)
256x256	4x4	8x8	1024	0,058
256x256	8x8	4x4	1024	0,055
256x256	4x2	4x8	256	0,057
256x256	4x4	4x4	256	0,054
512x512	4x4	8x8	1024	0.058
512x512	8x8	4x4	1024	0.056
512x512	16x16	4x4	4096	0.090
512x512	8x8	8x8	4096	0.059
1024x1024	4x8	16x8	4096	0.060
1024x1024	4x4	16x16	4096	0.060

Tablo 4.2'de de görüldüğü gibi, kontur üzerindeki noktalar üzerinde çalışıldığı için, çalışma süresi görüntü boyutu ile değişmemektedir. Çalışma süresinin kontur üzerindeki nokta sayısı ile değişebileceği düşünülür, fakat nokta sayısı da bir etki göstermemektedir. Bu sonucun 2 nedeni vardır. Birincisi algoritmanım çalışma karmaşıklığı O(1) dir. Her kanal kendi işini yaparak beklemektedir. İkinci ve asıl nedeni ise ekran kartının kapasitesinin kullanılmamasıdır. Görüldüğü gibi blok ve kanal sayısı çok küçük boyutlardadır. Bu ise eş zamanlı çalışan kanal sayısının düşük olmasına neden olur. Burada asıl önemli faktör kanal sayısının yeterli oranda seçilmesi gerekliliğidir. Örneğin ikinci,

dördüncü ve altıncı satırda kanal sayısı 4x4 boyutunda seçilmiş fakat performansa bir etkisi gözlenmemiştir. Fakat kontur nokta sayısı yükseldikçe 4x4 lük kanal sayısı yeterli gelmemektedir. Bu ise performansta ciddi düşüşlere sebep olmaktadır. Örneğin yedinci satırda 4x4 lük kanal sayısı kullanılmıştır fakat kontur sayısı büyük olduğu için bu kanal sayısı yeterli olmamıştır.

GPU üzerinde geliştirilen programların performanslı çalışması için birçok düzenleme yapılması gerekir. CUDA ile geliştirilen programların optimizasyonu için sadece kullanılacak kanal ve blok sayıları sayılarını ideal bir şekilde belirlemek yetmemektedir. Bir diğer önemli faktör ise bellek yönetimidir. Sadece kullanılan bellek türlerinde yapılan değişiklikler ile büyük ölçüde hızlanma sağlanabilmektedir.

5. SONUÇ

Yapılan çalışmada ölçeklendirme yaklaşımlı GPU tabanlı yeni bir seviye kümesi yöntemi sunulmuştur. Yapılan çalışmada heterojen programlama yöntemi izlenmiş bazı adımlar CPU tarafında bazı adımlar GPU tarafında yapılmıştır. Küçük işlemler ve kullanıcı girişleri CPU tarafına bırakılırken kontur hesaplanması gibi yüksek hesaplama gerektiren işlemler GPU tarafında yapılmış ve önemli ölçüde hız artırımı sağlanmıştır.

Geleneksel seviye kümesi yöntemine bakıldığı zaman, seviye kümesi fonksiyonu şeklinde adlandırılan yüksek dereceli bir fonksiyon kullanılarak eğri gelişimi sağlanmıştır. Daha sonra bu eğri gelişim denklemi kısmi diferansiyel denkleme dönüştürülerek çözülmüştür. Kısmi diferansiyel denklemleri çözümü için birçok sayısal yöntem kullanılabilir fakat bu yöntemlerin her biri yüksek hesaplama gücü gerektirmektedir. Bu tez çalışmasında geliştirilen yöntem ile bu seviye kümesi fonksiyonunun bu problemine çözümler aranmıştır.

Geliştirilen yöntem, eğri gelişimini basit geometrik dönüşümleri temel alarak gerçekleştirir, böylece algoritma çok daha hızlı çalışması sağlanmıştır. Ayrıca algoritmanın ön işleme ve kontur gelişimi gibi yüksek hesaplama gerektiren kısımları GPU üzerinde çalıştırılmıştır. Sonuç olarak tüm işlemler yaklaşık 2 milisaniye içinde tamamlanmıştır. Bu ise, geliştirilen yöntem ile gerçek zamanlı çalışma mümkün olduğunu göstermektedir.

6. KAYNAKLAR

[1] **Osher S., and N. Paragios**, 2003. Geometric Level Set Methods in Imaging, Vision and Graphics, Springer-Verlag New York, Secaucus, NJ, USA.

[2] **Shattuck, D. W., Prasad, G., Mirza, M., Narr, K. L., and Toga A. W.**, 2009. Online resource for validation of brain segmentation methods, *NeuroImaging*, **45**, 431-439.

[3] **Osher S., and J. Sethian**, 1998. Fronts propagating with curvature-dependent speed: Algorithims based on Hamilton-Jakobi formulation, *J. Comput. Phys.*, **79-1**, 12-49.

[4] **Fetkiw R.**, 2002. Simulating Natural Phenomena for Computer Graphics, Geometric Level Set in Imaging, Vision and Graphics, 461-479.

[5] **Adalsteinsson D., and Sethian J.**, 1995. A Fast Level Set Method for Propagating Interfaces, *Journal of Computational Physics*, **118-2**, 269-277.

[6] **Whitaker R.**, 1998. A Level-Set Approach to 3D Reconstruction from Range Data, *International Journal of Computer Vision*, **29-3**, 203-232.

[7] **Awate S. P., and Whitaker R. T.**, 2004, An Interactive Parallel Multiprocessor Levet-Set Solver with Dynamic Load Balancing, *Scientific Computing and Imaging Institute Technical Report*, **UUCS-05-002**, University of Utah, Salt Lake City, UT, USA.

[8] **Sanders, J. and Kandrot, E.**, 2011. CUDA By Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley, NVIDIA Cooporation.

[9] B**ilotta G., Rustico E. and Heroult A.** From GPU to GPGPU, http://www.dmi.unict.it/~bilotta/gpgpu/notes/05-gpgpu-history.html, 14 Mayıs 2013.

[10] NVIDIA., CUDA C Programming Guide v5.0, 2012. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 15 Mayıs 2013

[11] **Hwu W.W.**, 2011. GPU Computing Gems: Jade Edition, Morgan Kaufmann.

[12] **Cook S.**, 2012. Cuda Programming: A Developer's Guide to Paralel Computing with GPUs, Elsevier, Mprgan Kaufmann.

[13] Ooster J. Cuda Memory Model, http://3dgep.com/?p=2012, 14 Mayıs 2013.

[14] **Kirk D. B. and Hwu W. W.**, 2010. Programming Massively Parallel Processors, Elsevier, Morgan Kaufmann.

[15] http://www.cis.temple.edu/~giorgio/cis307/readings/cuda.html, GPUs and CUDA, 12 Mayıs 2013.

[16] **Li Y., Zhao K., Chu X. and Liu J.**, 2013. Speeding up k-Means Algorithm by GPUs, *Journal of Computer and System Science*, **79-2**, 216-229.

[17] **Rumpf M., and Strzodka R.**, 2001, Level Set Segmentation in Graphics Hardware, *IEEE International Conference on Image Processing (ICIP'01)*, Thessaloniki, Greece, October 7-10, pp. 1103-1106.

[18] **Lefohn A., and Whitaker R.** 2002. A GPU-Based Three-Dimensional Level Sey Solver with Curvature Flow, *Scientific Computing and Imaging Institute Technical Report*, **UUCS-02-017**, University of Utah, Salt Lake City, UT, USA.

[19] **Lefohn A., Kniss J. M., Hansen C. D., and Whitaker R. T.**, 2004. A Streaming Narrow-Band Algorithm: Interactive Computation and Visualization of Level Sets, *IEEETransactions on Visualization and Computer Graphics*, **10**, 422-433.

[20] http://www.itk.org, The Insight Toolkit (ITK) 2003, 14 Mayıs 2013.

[21] **Klar O.**, 2007. Interactive GPU-based Segmentation of Large Medical Volume Data with Level-Sets, *11th Central European Siminar on Computer Graphics (CESCG'07)*, Budmerice Castle, Slovakia, April 23 - 25.

[22] **Mostofi H., and Colege K.**, 2009. Fast level Set Segmentation of Biomedical Images using Graphics Processing Units, *Final Year Project*, University of Oxford, Department of Engineering Science Oxford.

[23] **Hagan A., and Zhao Y.**, 2009. Parallel 3D Image Segmentation of Large Data Sets on a GPU Cluster, *5th International Symposium on Visual Computing*, Las Vegas, Nevada, USA, November 30 - December 2, pp. 960-969.

[24] **Tornai G.J., and Cserey G.**, 2010. 2D and 3D level-set Algorithm on GPU, 12th International Workshop on Cellular Nanoscale Network and Their Applications (CNNA), Berkeley, CA, USA, February 3-5, pp. 1-5.

[25] **Shi Y., and Karl W.**, 2005. A fast level set method without solving PDEs *International Conference on Acoustics, Speech, and Signal Processing, (ICASSP 2005)*, Philadelphia, Pennsylvania, USA, March 23.

[26] **Roberts M., Packer J., Sousa M. C., and Mitchell J. R.**, 2010. A Work-Efficient GPU Algorithm for Level Set Segmentation, *High Performance Graphics (HGP '10)*, Saarbrucken, Germany, June 25-27, pp. 123-132.

[27] **Jalba, A. C., Van Der Laan, W. J., and Roerdink, J. B. T. M.**, 2013. Fast Sparse Level Sets on Graphics Hardware, *Visualization and Computer Graphics*, **19-1**, 30-44.

[28] **Kass, M., Witkin, A. and Terzopoulos, D.**, 1998. Snakes: Active Contour Model, *International Journal of Computer Vision*, **1-4**, 321-331.

[29] **Terzopoulos, D., and Fleischer, K.** 1988. Deformable Models, *The Visual Computer*, **4-6**, 306-331.

[30] **Yuille, A. L.**, 1991. Deformable Templates for Face Recognition, *Journal of Cognitive Neuroscience*, **3-1**, 59–70.

[31] **Osher, S. and Fedkiw, R.**, 2003. Level Set Methods and Dynamic Implicit Surfaces. Springer.

[32] **Nixon, M. S. and Aguado, A. S.**, 2002. Feature Extraction and Image Processing. Newnes.

[33] **Kale, H. E.**, 2011. Segmentation of Human Facial Muscles on CT and MRI Data Using Level Set and Bayesian Method, *Yükzek Lisans Tezi*, ODTÜ, Enformatik Enstitüsü, Ankara.

[34] **Li C., Xu C., Gui C., and Fox M. D.**, 2010. Distance Regularized Level Set Evolution and Its Application to Image Segmentation, *IEEE Transactions on Image Processing*, **19-12**, 3243-3254.

[35] http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm, Gaussian Smoothing, 14 Mayıs 2013.

[36] http://radiopaedia.org, 9 Mayıs 2013.

ÖZGEÇMİŞ

1987 yılında Tokat'ta doğdu. 2005 yılında Tokat Anadolu Lisesinden mezun olduktan sonra, Anadolu Üniversitesi, Bilgisayar Mühendisliği bölümünde lisans eğitimine başladı. 2010 yılında mezun oldu. Lisans derecesini aldıktan sonra 2010 yılının Ekim ayında, Fırat Üniversitesi Yazılım Mühendisliği bölümünde araştırma görevlisi olarak çalışmaya başladı. 2011 yılının Ekim ayında Fırat Üniversitesi Bilgisayar Mühendisliği dalında, Yazılım bilim dalında yüksek lisansa başladı. Araştırma görevlisi olarak Fırat Üniversitesinde çalışmaya devam etmektedir.