İSTANBUL ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

YÜKSEK LİSANS TEZİ

NVIDIA CUDA İLE YÜKSEK PERFORMANSLI GÖRÜNTÜ İŞLEME

Bilgisayar Müh. Ertan YILDIZ
Bilgisayar Mühendisliği Anabilim Dalı
Bilgisayar Mühendisliği Programı

Danışman
Prof.Dr. Sabri ARIK

Temmuz, 2011

İSTANBUL

Bu çalışma 12/08/2011 tarihinde aşağıdaki jüri tarafından Bilgisayar Mühendisliği Anabilim Dalı Bilgisayar Mühendisliği programında Yüksek Lisans Tezi olarak kabul edilmiştir.

Tez Jürisi

Danışman Adı
(Danışman)
Prof. Dr. Sabri ARIK

İstanbul Üniversitesi
Bilgisayar Mühendisliği

Jüri Adı
Prof. Dr. Ahmet SERTBAŞ

İstanbul Üniversitesi
Bilgisayar Mühendisliği

Jüri Adı
Doç. Dr. Mustafa ONAT

Marmara Üniversitesi
Elektrik-Elektronik Mühendisliği

Jüri Adı
Yrd. Doç. Dr. Olcay KURŞUN

İstanbul Üniversitesi
Bilgisayar Mühendisliği

Jüri Adı
Yrd. Doç. Dr. Oğuzhan ÖZTAŞ

İstanbul Üniversitesi
Bilgisayar Mühendisliği

ÖNSÖZ

Bu çalışma, İstanbul Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı Yüksek Lisans Tezi olarak hazırlanan "NVIDIA CUDA ile Yüksek Performanslı Görüntü İşleme" isimli tezi içermektedir.

Lisans ve yüksek lisans öğrenimim sırasında ve tez çalışmalarım boyunca gösterdiği her türlü destek ve yardımdan dolayı çok değerli hocam Prof.Dr. Sabri ARIK'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.

Temmuz, 2011 **Ertan YILDIZ**

İÇİNDEKİLER

ŞEKİL LİSTESİ

TABLO LİSTESİ

SEMBOL LİSTESİ

API : Uygulama programlama arayüzü
BLASS : Temel lineer cebir fonksiyonları
CPU : Merkezi işlem ünitesi
CUBLAS : CUDA temel lineer cebir fonksiyonları
CUFFT : CUDA hızlı Fourier dönüşümü
FBO : Görüntü karesi tampon nesnesi
FPS : Saniyedeki görüntü karesi sayısı
FFT : Hızlı Fourier dönüşümü
GFLOPS : Saniyede kayan noktalı işlem sayısının bir milyarda biri
GPGPU : Grafik işleme ünitesi üzerinde genel amaçlı hesaplama
GPU : Grafik işleme ünitesi
GPUCV : GPU hızlandırmalı bilgisayarla görme
NPP : NVIDIA performans temel öğeleri
OPENCL : Açık hesaplama dili
OPENCV : Açık kaynak bilgisayarla görme
OPENGL : Açık grafik kütüphanesi
PBO : Piksel tampon nesnesi
RGB : Kırmızı-yeşil-mavi
SDK : Yazılım geliştirme kiti

ÖZET

NVIDIA CUDA İLE YÜKSEK PERFORMANSLI GÖRÜNTÜ İŞLEME

Son yıllarda hızla gelişen GPU teknolojisi araştırmacıların ve yazılım geliştiricilerin dikkatini çekmiştir. Özellikle GPU'ları genel amaçlı hesaplamalar için kullanabilen programlama arayüzlerinin geliştirilmesi ile GPU dünyasında yeni bir çağ başlamıştır. Yaşanan bu gelişmeler, tıbbi görüntü işleme ve 3D modelleme gibi birçok alanda yeni fikirleri tetiklemiştir. Görüntü işleme gibi yüksek derecede veri paralelliği gerektiren bazı alanlarda köklü değişiklikler yaşanmıştır.

Bu çalışmada, NVIDIA firmasının CUDA destekli platformlarında, paralel programlama ile yüksek performanslı uygulamalar geliştirmeyi sağlayan CUDA C dili incelenmiştir. Bu dil ile örnek bir tekil nesne takibi uygulaması geliştirilmiştir. Uygulama geliştirme aşamasında CUDA C dilinin bazı temel özellikleri test edilmiş ve sonuçlar ayrıntılı istatistikler ile birlikte sunulmuştur.

SUMMARY

HIGH PERFORMANCE IMAGE PROCESSING WITH NVIDIA CUDA

GPU technology, which is developing rapidly in recent years, has attracted the attention of researchers and software developers. Especially the development of programming interfaces that can use GPUs for general-purpose computations, started a new age ın GPU world. These developments triggered new ideas in so many fields such as medical image processing and 3D modeling. Fundamental changes have occurred in some fields such as image processing which require high level data parallelism.

This study examines the CUDA C language which provides development of high-performance applications by parallel programming on NVIDIA CUDA-supported platforms. A sample single object tracking application is developed via using CUDA C. Some of the basic features of the CUDA C language has been tested during application development stage and the results have been presented along with detailed statistics.

1. GİRİŞ

Son yıllarda, gerçek zamanlı ve yüksek çözünürlüklü grafiklere olan talep, programlanabilir grafik işlemcilerinin yüksek hesaplama gücüne ulaşmasına yönelik çalışmaları hızlandırdı. Bu da GPU programlama kütüphanelerinin iyileştirilmesi, OpenGL (Open Graphics Library) [1] grafik kütüphanesi entegrasyonu ve OpenCL (Open Computing Language) [2] gibi farklı mimariler üzerinde çalışabilen dillerin ortaya çıkması gelişmelerine yol açmıştır.

2006 yılının sonlarına doğru NVIDIA[3] firması piyasadaki ilk DirectX 10 GPU temelli ürünü olan GeForce 8800 GTX ürününü tanıttı[4]. Bu ürün aynı zamanda CUDA (Compute Unified Device Architecture) mimarisi ile üretilen ilk grafik kartıydı. CUDA mimarisi, GPU'nun genel amaçlı kullanımını sağlayacak ve önceki grafik işlemcilerin limitlerini ortadan kaldıracak şekilde tasarlanmış birçok yeni elaman içeriyordu.

Yaşanan bu gelişmeler görüntü işleme gibi performans değerlerinin donanımdan çok fazla etkilendiği çalışmalarda, bu yeni ve aynı zamanda düşük fiyatlı donanımın kullanılması konusundaki araştırmaların ilham kaynağı olmuştur. Ortaya çıkan sonuçlar daha önce yüksek maliyetli donanımlarla erişilebilen performans değerlerine çok daha az maliyetlerle ulaşılabileceğini ispatlamıştır.

1.1. TEZİN AMACI

Bu tezde öncelikli amaç, resim işlemedeki bazı konvolüsyon ve filtreleme örneklerinin GPU üzerinde CUDA C dili ile CPU üzerindeki örneklerine göre daha yüksek performans gösteren türevlerini oluşturmaktır. Diğer hedefler ise:

1. NVIDIA firmasının CUDA mimarisi ile üretilmiş ekran kartlarında program geliştirmeyi mümkün kılan CUDA C dilinin tanıtımı ve pratik örneklerinin değerlendirilmesi.
2. CUDA/C++/OpenGL entegrasyonun kullanıldığı yüksek performanslı bir nesne takip uygulamasının hazırlanması.

İşlenecek görüntülerin uygulama tarafından okunması ve bazı görüntü işleme algoritmalarının CPU-GPU karşılaştırmalarında OpenCV (Open Source Computer Vision) [7] kütüphanesinden faydalanılmıştır.

1.2. TEZİN YAPISI

Bu çalışmada NVIDIA CUDA C ve C++ dilleri kullanılarak, OpenGL entegrasyonlu çoklu nesne belirleme ve tekil nesne takibi uygulamaları gerçekleştirilmiştir. Yazılım Visual Studio 2008 ortamında hazırlanmıştır. CUDA C ile Visual Studio ortamında debug işlemi gerçekleştirilemediği için NVIDIA Parallel Nsight eklentisinden yararlanılmıştır. Video okuma işlemleri OpenCV 2.2 versiyonu kullanılarak gerçekleştirilmiştir.

Tezin ikinci bölümünde NVIDIA CUDA mimarisi ve CUDA C diline ait tarihsel süreçle ilgili bilgiler yer almaktadır.

Üçüncü bölüm CUDA mimarisinin detaylarının anlatıldığı ve CPU ile temel farklarının sunulduğu kısımdır. Ayrıca bu bölümde; CUDA mimarisinin temel avantajları ve dezavantajlarına kısaca değinilmiştir.

Dördüncü bölümün birinci kısmı NVIDIA CUDA C diline giriş, dilin temel kavramları ve gerekli geliştirme ortamları hakkında ayrıntılı bilgi içerir. Bu bölümün ikinci kısmında ise NVIDIA CUDA C ile C++ dilinin entegrasyonu konusunda detaylar yer almaktadır. Üçüncü kısımda CUDA C / OpenGL birlikte çalışabilirliği konusuna yer verilmiştir. Bu bölümün dördüncü ve son kısmında ise Visiual Studio 2008-Cuda entegrasyonu ve debug işleminin detaylarına değinilmiştir.

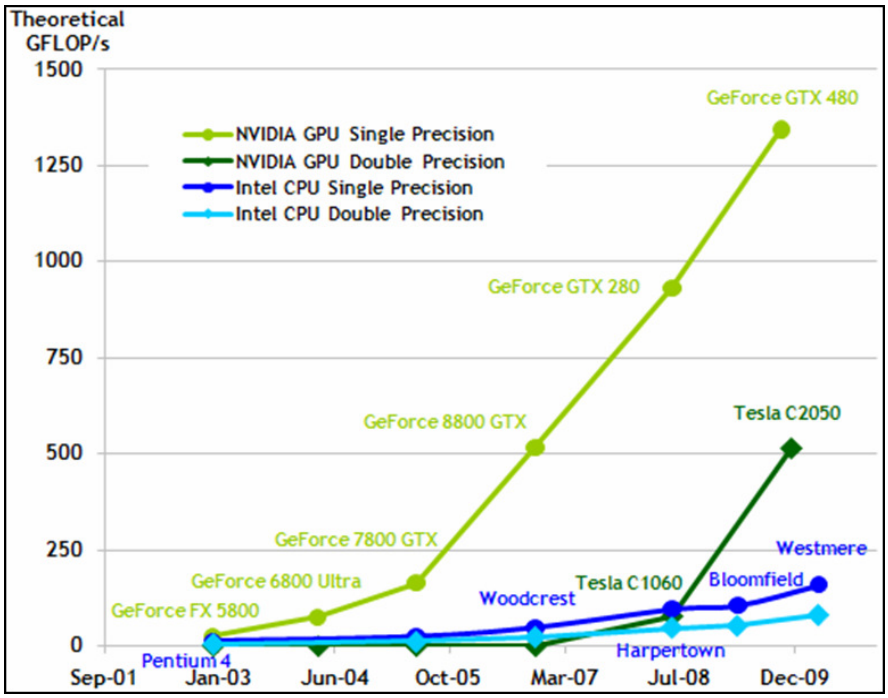
Beşinci bölümde CUDA C ile geliştirilecek örnek uygulamaların detaylarını, akış şemalarını ve örnek ekranları içerir. Ayrıca bu bölümde zaman zaman bazı işlemler için CPU-GPU kıyaslamalarına yer verilmiştir.

Altıncı bölüm tartışmalar, sonuçlar ve gelecek için önerilen çalışmalardan oluşmaktadır.

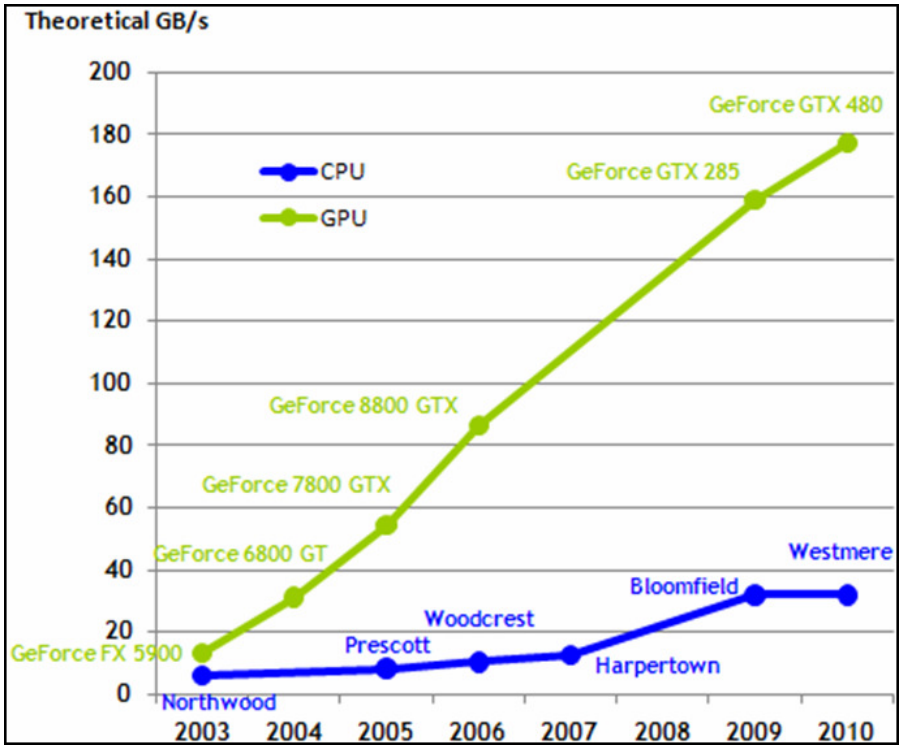
2. GENEL KISIMLAR

1980'li yıllarda IBM'in ve Intel'in hakimiyetindeki GPU geliştirme çalışmaları, 1990'lı yılların başından itibaren sadece grafik kartı geliştirmeye yönelmiş olan S3 Graphics, NVIDIA ve ATI gibi firmaların kontrolünde devam etti. Bu dönemde 2D donanımsal hızlandırma yaygınlaştı. Yine aynı dönemde OpenGL grafik programlama kütüphanesinin kullanıma sunulmasıyla GPU dünyasında yeni bir dönem başladı.

2000'lı yıllarda ise GPU geliştirme çalışmaları, GPU hesaplama gücünün kat ve kat artmasıyla sonuçlanmıştır. Şekil 1.1 de yıllara göre GFLOPS (Giga Floating-Point Operations Per Second) bazında, şekil 1.2 de ise yıllara göre bellek bant genişliği bazında CPU-GPU kıyaslaması yer almaktadır.



Şekil 2.1: Yıllara göre CPU-GPU FPLOPS karşılaştırması[5].



Şekil 2.2: Yıllara gore CPU-GPU bellek bant genişliği karşılaştırılması[5].

GPU'ların artan hesaplama kabiliyetleri GPGPU (General Purpose Computing On GPU) [8] çalışmalarının görüntü işleme, lineer cebir, istatistik ve 3D modelleme gibi alanlarda uygulanmasının tetikleyicisi olmuştur.

2.1. NVIDIA CUDA İLE GÖRÜNTÜ İŞLEME: TARİHSEL GELİŞİMİ

OpenGL ve DirectX [6] gibi kütüphanelerin genel amaçlı GPU programlama ihtiyacını karşılayamaması bu alanda herkesin yararlanabileceği programlama arayüzlerinin geliştirilmesi yönünde çalışmaların başlamasını sağlamıştır. Bunların en önemli iki tanesi Lib Sh ve BrookGPU'dur[9, 10, 11].

BrookGPU, Stanford Universitesi Grafik Grubu'nun Brook veri dizisi programlama dilinin bir uygulaması niteliğindedir. Bu proje ayrıca CUDA C dilinin de başlangıcı olmuştur.

NVIDIA firması Şubat 2007'de CUDA SDK (Software Development Kit)'yı genel kullanıma açtı.

2007 Haziran'ında Victor Podlozhnyuk [12, 13] isimli araştırmacı CUDA ile basit konvolüsyon ve FFT (Fast Fourier Transform) temelli konvolüsyon işlemlerini CUFFT kütüphanesinden de faydalanarak uygulamış ve analizlerini sunmuştur.

Aynı zaman diliminde Alexander Kharlamov ve Victor Podlozhnyuk [14] isimli araştırmacılar CUDA ile görüntü üzerinde gürültü azaltma üzerine bir çalışma ortaya koymuşlardır ve bilinen bazı metotları CUDA'ya uyarlamışlardır. Bu çalışma sonucunda GeForce8800 GTX graifk kartı üzerinde, 320x408 ebatlarındaki örnek bir resim için 500 fps (Frame Per Second) değerine ulaşmıştır.

2008 yılında Shuai Che ve diğ. [15] bir görüntü işleme uygulamasını GeForce GTX 260 üzerinde CUDA C ile ve Intel Xeon üzerinde OpenMP (Open Multi-Processing) [16] ile gerçeklemişlerdir. Yapılan çalışma CUDA C'nin optimizasyona gerek duymadan daha başarılı olduğunu ortaya koymuştur. Araştırmacılar, CUDA C ile yazılım geliştirmenin OpenMP ile geliştirmeye göre biraz daha zor olduğunu vurgulamışlardır.

2008 yılında Michael Garland ve diğ. [17] yaptıkları çalışmada çeşitli işlemlerde CUDA kullanımı üzerine tecrübelerini paylaşmışlardır. Medikal alanda görüntü işleme çalışmalarında CUDA ile ulaşılan performans değerlerini, aynı çalışmaların CPU eşlenekleri ile kıyaslamışlardır.

2008 yılında Zhiyi Yang ve diğ. [18] tarafından yapılan çalışmada, yaygın olarak kullanılan kenar bulma, histogram alma gibi görüntü işleme adımlarının GPU uygulaması yapılmış ve CPU eşlenekleri ile çalışma zamanı bakımından kıyaslanmıştır.

2008 yılında Yannick Allusse ve diğ. [19] tarafından sunulan çalışmada GpuCV [20] resim işleme kütüphanesinin tanıtımı yapılmıştır. Bu çalışma ile görüntü işlemede yaygın olarak kullanılan OpenCV kütüphanesinde bulunan fonksiyonların mümkün olan noktalarda CUDA eşlenekleri hazırlanmıştır.

2008 yılında Jing Huang ve diğ. [21] tarafından yapılan çalışmada CUDA C ile hareket takibi uygulaması yapılmıştır. Uygulama farklı GPU'lar üzerinde de denenmiş ve GPU'lar arasındaki küçük farkların dahi uygulamalar arasında büyük performans farklarına neden olduğunu gözlemişlerdir.

2008 yılında Lei Pan ve diğ. [22] tarafından yapılan çalışmada tibbi görüntü işlemede önemli bir konu olan görüntü bölütleme konusunda bazı yaygın kullanıma sahip algoritmaların GPU uygulamaları gerçekleştirilmiştir. Ayrıca CUDA ve önceki GPU teknolojileri arasında yapılan kıyaslamalara da yer verilmiştir.

2008 yılında James Fung ve Steve Mann [23] tarafından yapılan çalışmada, GPU'nun görüntü işlemede daha aktif nasıl kullanılabileceği konusunda bilgilere yer verilmiştir. Ayrıca GPU bellek erişim teknikleri arasında kıyaslamalar yapılmıştır.

2009 yılında Jun-Sik Kim ve diğ. [24] tarafından yapılan çalışmada işlem yoğunluğuyla bilinen KLT (Kanade-Lucas-Tomasi) algoritmasının GPU uygulamasına yer verilmiştir. Ayrıca bu uygulama farklı CPU ve GPU modelleri üzerinde çalıştırılarak sonuçlar karşılaştırılmıştır. İzleme, nitelik kaydı gibi adımlar ayrı ayrı teste tabi tutulmuştur. Kamera açısından bağımsız olarak izleme adımında yaklaşık 50 fps değerini yakalamışlardır.

2009 yılında Raúl Cabido ve diğ. [25] tarafından sunulan çalışmada, GPU üzerinde parçacık süzme algoritması kullanarak nesne takibi uygulaması yapılmıştır. Konu üzerinde daha önceki çalışmalarda 20 fps civarında sonuçlar alınırken, CUDA teknolojisinin kullanıldığı bu çalışmada, 6 durum değişkeni ve 256 parçacıklı bir parçacık süzgeci uygulamasında ve 320x240 çözünürlüklü video için 700 fps değerine ulaşılmıştır.

2009 yılında Michael Boyer ve diğ. [26] tarafından yapılan çalışma ile lökosit tespit ve takip etmeye yönelik bir medikal çalışmada 200x'e varan performans artışı sağlanabileceği düşüncesi uygulamaya dökülmüştür. Nitekim araştırma sonuçları lökosit takibi işlemlerinde araştırmacıların iddaalarında haklı olduklarını göstermiştir. 30 fps olarak kaydedilmiş örnek bir videoda MATLAB [27] ile 0.11 fps değeri ile

izleme gerçekleştirilirken, bu değer CUDA ile 21.6 fps olmuştur. Denemelerde kullanılan 1 dakika uzunluğundaki videonun işlenmesi MATLAB ile 4.5 saat sürerken, CUDA ile 1.5 dakika sürmüştür.

2009 yılında V.A.Prisacariu ve Ian Reid [28] tarafında yapılan çalışmada HOG algoritması kullanılarak görüntü üzerinde nesne takip uygulaması gerçekleştirilmiştir. Paralel dizayna sahip bu uygulamanın, sıralı dizayndaki eşleneklerine göre 67 kat daha hızlı olduğu saptanmıştır.

2011 yılında Richard Membarth ve diğ. [29] tarafından yapılan çalışmada CUDA programlama modeli kullanılarak çoklu çözünürlük algroritmalarının bir çeşidi uygulanmıştır. Çalışmada ayrıca bellek transfer sürelerini azaltmak için bazı teknikler kullanılmıştır. Sonuç olarak uygulama NVIDIA Tesla C1060 [30] üzerinde, Xeon Quad Core üzerindekinden 145 kat daha hızlı çalışmıştır.

3. MALZEME VE YÖNTEM

3.1. NVIDIA CUDA MİMARİSİ

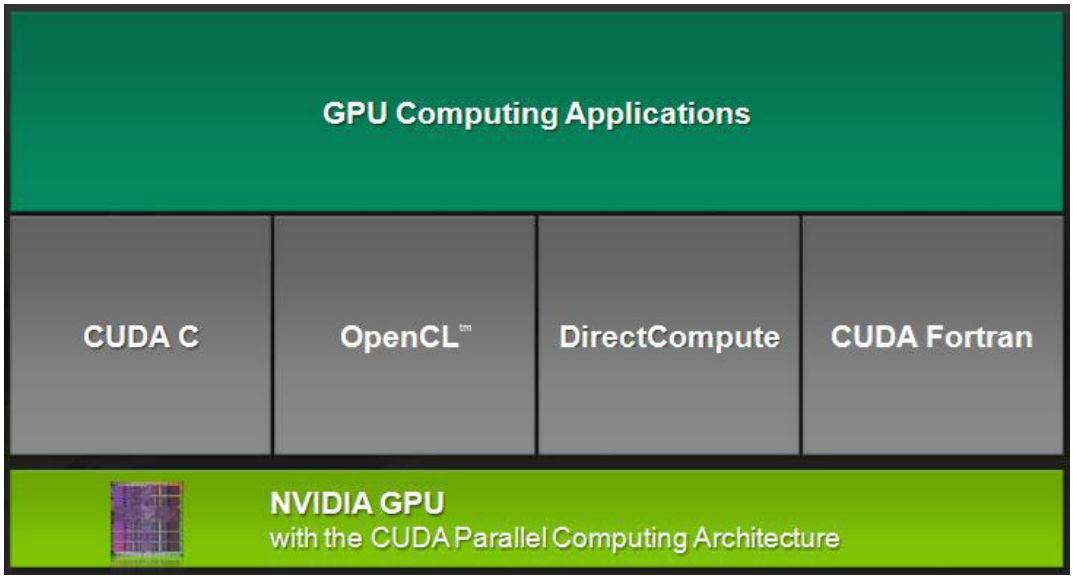
CUDA mimarisi üzerinde yapılan örnek çalışmalarla performansın CPU üzerindeki uygulamalara göre daha yüksek olduğu netleşmiştir[15, 17, 18, 24, 26, 28, 29]. Aslında bu fark temel olarak GPU mimarisinin, grafik işleme gibi işlem yoğunluğu olan ve yüksek derecelerde paralellik gerektiren işlemler için geliştirilmiş olmasından kaynaklanmaktadır.

Şekil 3.1: GPU veri işleme için daha fazla transistor tahsis eder[5].

CPU'daki gibi akış kontrolü gerektiren işlemler yerine, aritmetiğin yoğun olduğu ve her bir piksel için tekrar eden işlemlerden oluşan resim/görüntü işleme, 3D derleme ve sinyal işleme türündeki uygulamaların GPU'nun hedef uygulamaları olması, önbellek ve akış kontrol mekanizmalarının yerine veri işlemeye yönelik transistörlerin ağırlıkta olduğu bir mimarinin ortaya çıkmasını sağlamıştır.

3.1.1. Programlama modeli

CUDA, yazılım geliştiricilerin yüksek seviyeli bir dil olan C ile yazılım geliştirmelerine izin veren bir ortamla birlikte gelir. Şekil 3.2'de CUDA platformu tarafından desteklenen diller görülmektedir.



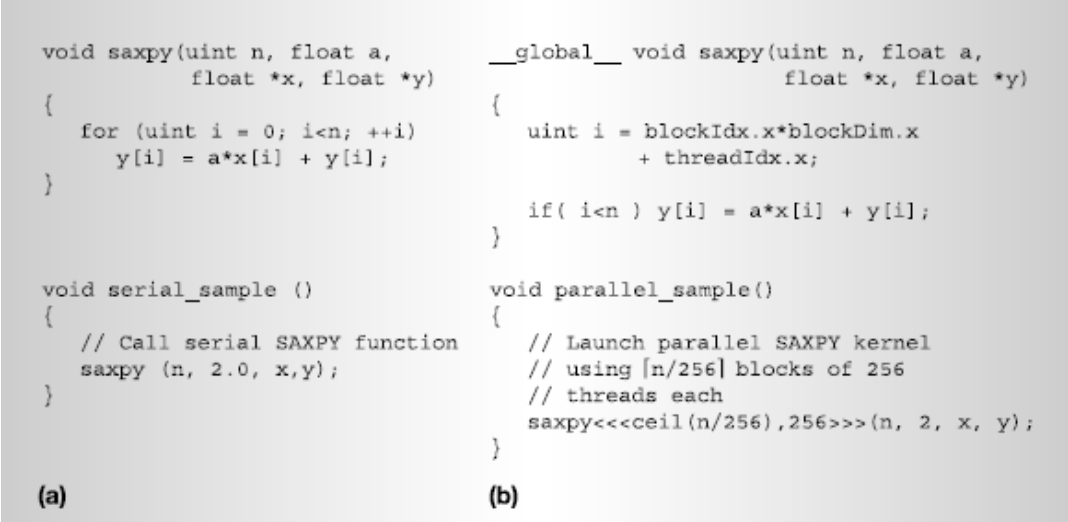
Şekil 3.2: CUDA çeşitli dilleri ve uygulama arayüzlerini destekleyecek şekilde tasarlanmıştır.

Tasarlanışı bakımından CUDA mimarisi ölçeklenebilir programlama imkanı sağlar. Yazılımcıyı GPU çekirdekleri ile uğraşmak zorunda bırakmaz. Bunu binlerce kanalı aynı işlem için çalıştırarak sağlar. CPU'da ise çok az miktardaki çekirdek için yine az miktarda kanal çalıştırılır.

3.1.1.1. Kerneller

CUDA C, standart C dilini genişletir ve kullanıcılara kernel denilen C fonksiyonlarını tanımlama imkanı sağlar. Kernel fonksiyonları, normal C fonksiyonlarından farklı olarak N kere çalıştırıldıklarında, N tane ayrı kanalda paralel olarak çalışırlar.

Kernel fonksiyonları __global__ ifadesi ile tanımlanır. Kerneli çalıştıracak kanalların sayısı <<<...>>> ifadesi ile belirtilir. Kerneli çalıştıran her bir kanala özel bir anahtar değer (ID) verilir. Bu değere "threadIDx" değişkeni üzerinden ulaşılır.



Şekil 3.3: x ve y vektörleri için αx+y işleminin sıralı hali (a), CUDA destekli paralel hali(b)[17].

Şekil 3.3 b'deki gösterimde saxpy kerneli <<<B,T>>> türünde bir konfigürasyon ile çağırılmıştır. Burda B blok sayısını ve T de her bir blok içindeki kanal sayısını ifade eder[17].

3.1.1.2. Kanal Hiyerarşisi

Kanalların ID değerlerine ulaşmamızı sağlayan threadIDx değişkeni 3 elementli bir elamandır. Bu yapısı ile vektör, matris veya 3D data kümeleri üzerinde işlemlerin kolayca yapılmasını sağlar.

Kanal indeks değeri ve kanal ID değerleri arasındaki ilişki, data bloğunun boyutlarına göre değişir. Tek boyutlu bir data bloğu için kanal index ve ID değerleri bir birine eşittir. İki boyutlu, boyutları D_x ve D_y olan bir data bloğu için, x ve y koordinatlarındaki kanal için indeks (K_{in}) değeri

$$K_{in}=x+y\ D_y \qquad (3.1)$$

x: Kanalın, kanallardan oluşan matris içerisindeki sütun numarası

y: Kanalın, kanallardan oluşan matris içerisindeki satır numarası

D_x: Kanal matrisinin sütun sayısı

D_y: Kanal matrisinin satır sayısı

Olarak hesaplanır. Üç boyutlu veri kümelerinde ise

$$K_{in}=x+y\ D_x+z\ D_x\ D_y \quad (3.2)$$

x: Kanalın, kanal kümesindeki koordinatının x değeri

y: Kanalın, kanal kümesindeki koordinatının y değeri

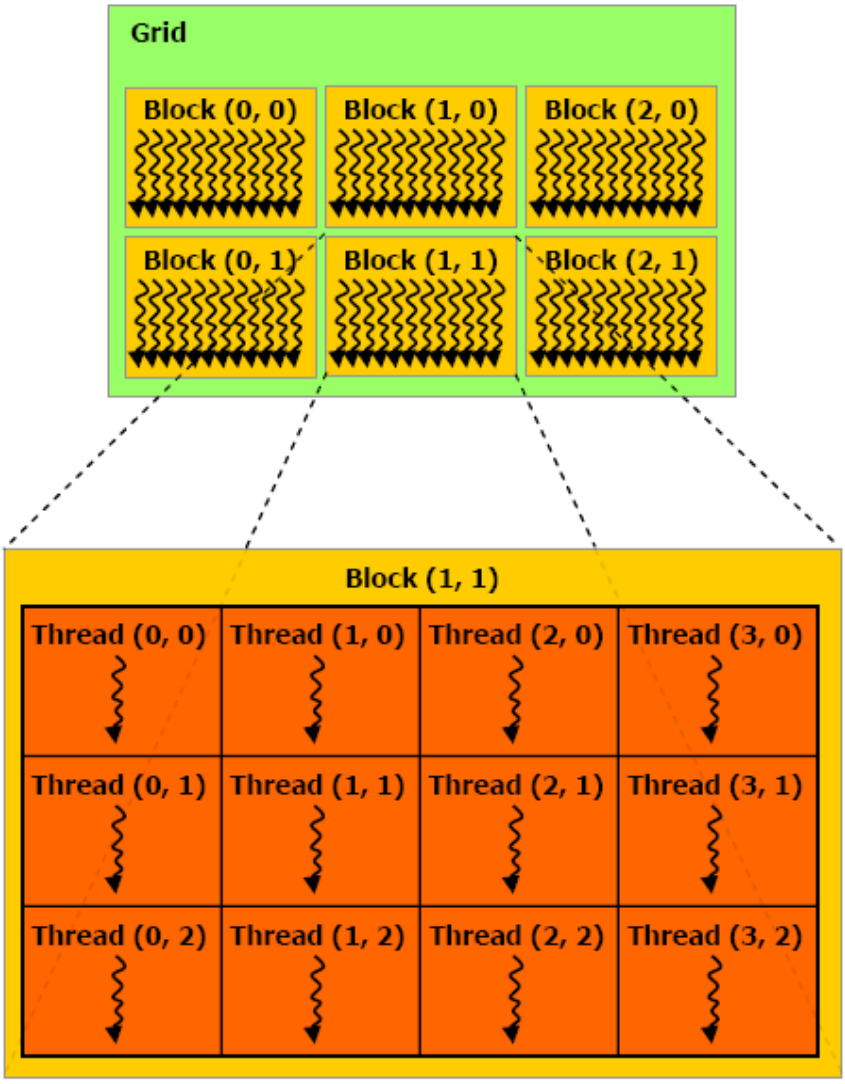
z: Kanalın, kanal kümesindeki koordinatının z değeri

D_x: Kanal kümesinin x ekseni değeri

D_y: Kanal kümesinin y ekseni değeri

D_z: Kanal kümesinin z ekseni değeri

formülü geçirlidir.



Şekil 3.4: CUDA grid, blok ve kanal yapısı örneği[5].

Bir blokta açılabilecek kanal sayısı donanıma göre değişiklik göstermektedir. Mevcut ürünlerde bu sayı 1024'e kadar çıkabilir. İşlenecek datanın tam kapsanması için eşit

şekillendirilmiş bloklardan yararlanılabilir. Böylece toplam kanal sayısı, bir bloktaki kanal sayısının toplam blok sayısı ile çarpılmasıyla elde edilir.

Bloklar bir boyutlu veya iki boyutlu kümeler halinde olabilir. Bu yapılara grid denir. Şekil 3.4'de 6 bloktan oluşan bir kanal yapısı görülmektedir. Tablo 3.1'de hesaplama kapasitesine göre blok, grid ve kanal özellikleri görülmektedir.

Grid içerisindeki herhangi bir blok, bir boyutlu veya iki boyutlu bir indeks ile ifade edilebilir. Bu indeks değeri blockIdx olarak adlandırılır. "blockDim" elamanı üzerinden söz konusu bloğun boyutları elde edilebilir.

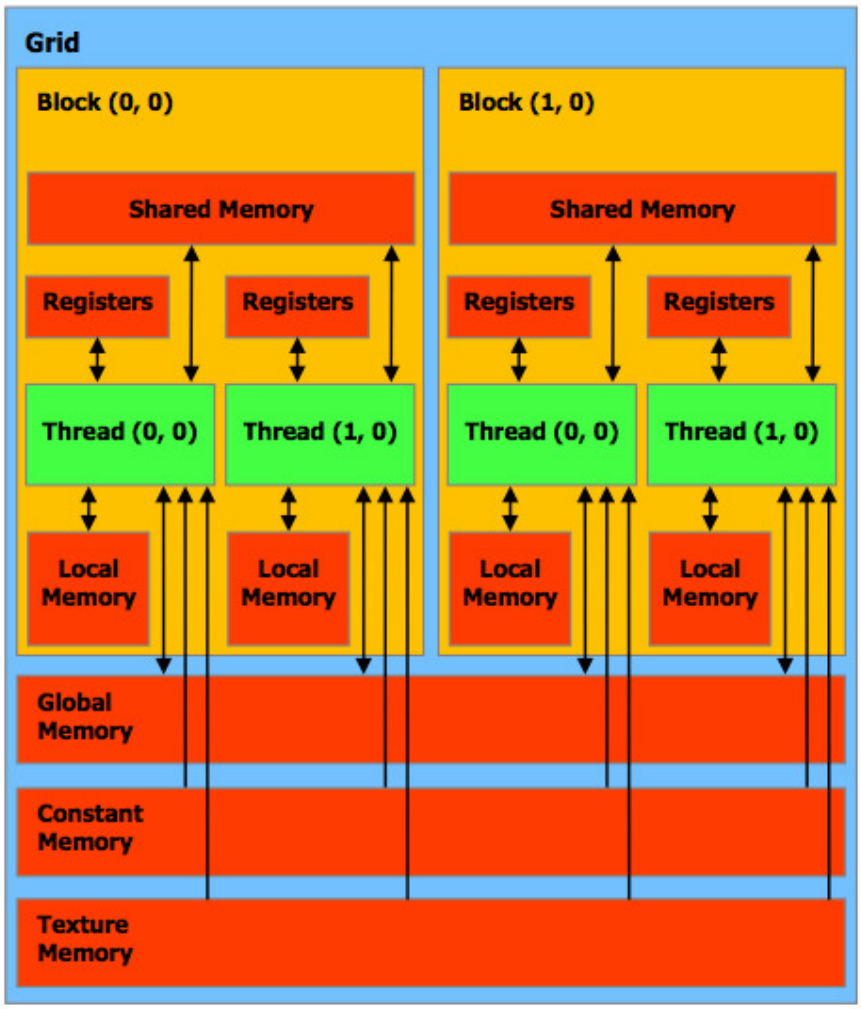
Blok içerisindeki kanallar etkileşimli olarak çalışabilirler. Ayrıca erişim hızının çok yüksek olduğu ortak paylaşımlı bellek yapısı mevcuttur. Bu yapıya erişimleri düzenleyen ve blok içerisindeki kanalların senkronizasyonunu sağlayan mekanizmalar da mevcuttur.

3.1.2. Bellek Modeli

CUDA kanallarının birçok bellek türüne erişimi vardır. Şekil 3.5'te kanallar ve erişebildikleri bellek türleri gösterilmektedir. Her bir kanal özel erişimi bulunan bellek alanına sahiptir. Her bir blok ise içerisindeki bütü kanalların erişimine açık olan ortak bellek alanına sahiptir. Bu bellek türünün ömrü, ilgili bloğun ömrü ile sınırlıdır. Bütün kanallar aynı global belleğe erişim hakkına sahiptirler.

Bunların dışında iki ayrı bellek türü daha vardır. Bunlar sırasıyla constant (değişmez) ve texture (doku) belleklerdir.

Değişmez bellek alanı, kullanılan grafik kartının hesaplama kapasitesi (compute capability) değerine göre değişir. Tablo 3.2'de hesaplama kapasitesine göre değişmez bellek toplam alanına yer verilmiştir.



Şekil 3.5: CUDA grid, blok ve kanal yapısı örneği [31].

Değişmez bellek alanı mevcut NVIDIA ürünlerinde 64KB ve bunun altındadır. Bu nedenle efektif kullanımı zordur.

Doku bellek, GPU tarafından donanım olarak desteklenen ve global bellek üzerindeki işlemlere göre daha yüksek performans değerlerini destekleyen bir teknolojidir. Uygulama arayüzü için sadece okuma işlemlerine izin verir. Kullanımında ise temel olarak önce bir veri kaynağı ile bağlantı yapılır. Daha sonra CUDA tarafından sağlanan fonksiyonlar ile veri yönetilir.

Global, değişmez ve doku bellek alanları bütün kerneller tarafından erişilebilir olarak tasarlanmışlardır.

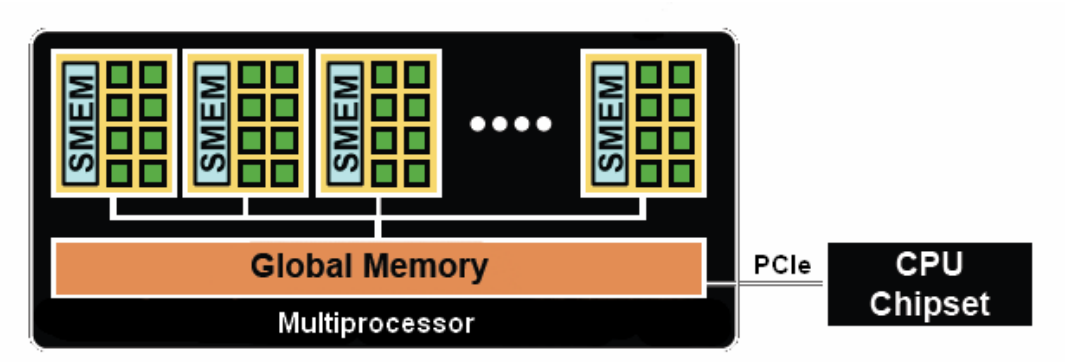
3.1.3. Çalıştırma Modeli

CUDA çalıştırma modeli ile ilgili bazı temel bilgiler aşağıda listelenmiştir.

1. CUDA çalışma modelinde kerneller gridler üzerinde çalışır.
    a. Aynı anda sadece bir tane kernel çalışır.
2. Bir kanal bloğu bir çoklu-işlemci (multiprocessor) çalışır.
    a. İşlemciler arasında aynı bloğa ait kanallar paylaşılmaz.
3. Bir çoklu-işlemci eş zamanlı olarak birden fazla bloğu çalıştırabilir.
    a. Burada çalıştırılabilecek kanal sayısı işlemcinin kaynakları ile sınırlıdır.
    b. Ortak (shared) bellek alanı bloklar arasında paylaştırılır.
    c. Yerel bellek (register) kanallar arasında paylaştırılır.

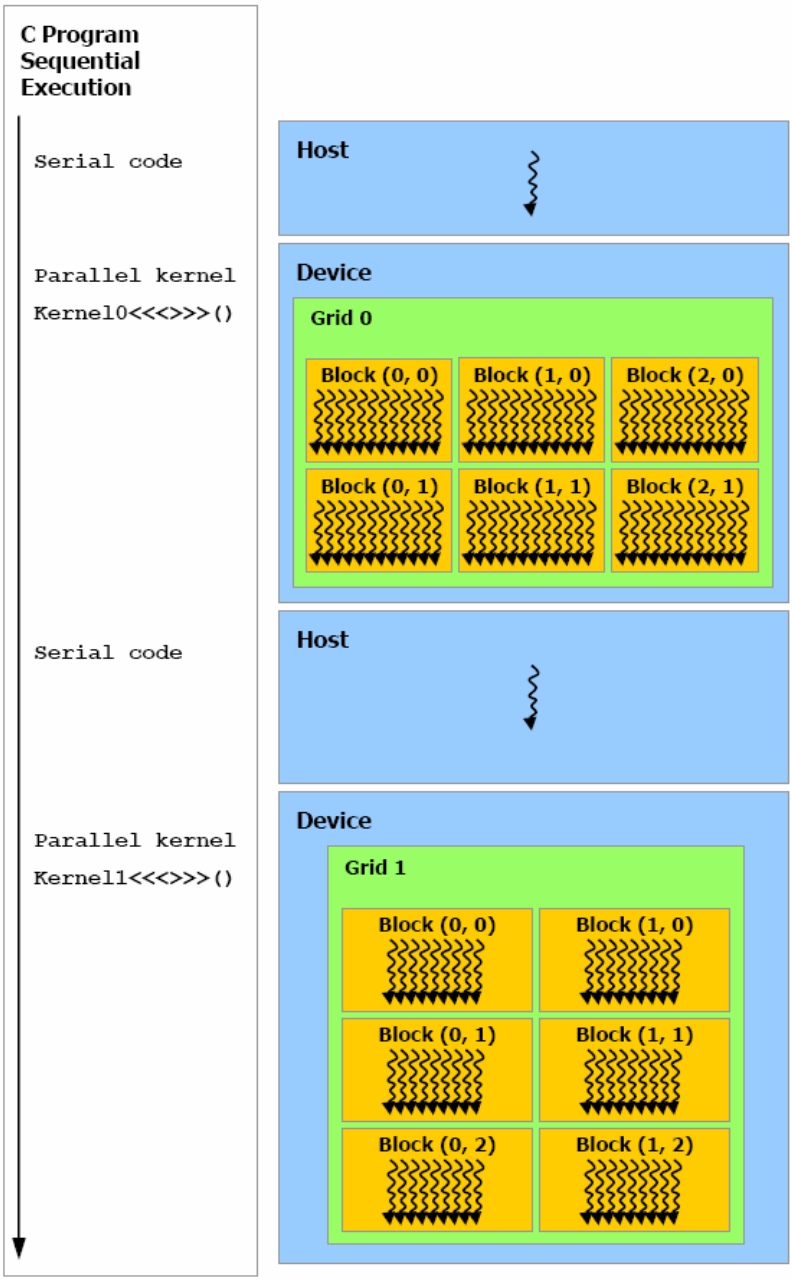
3.1.4. Hetorejen Programlama

CUDA programlama modeli temel olarak GPU'yu CPU'yardımcı bir işlemci olarak kabul eder. Şekil 3.6'da bu basit model görülmektedir.



Şekil 3.6: GPU-CPU basit modeli.

Bu modelde, çalıştırılan uygulamanın C programlama dili ile yazılmış sıralı düzendeki kısımlarını CPU, geriye kalan ve GPU üzerinde çalışan paralel kısımları ise CUDA yönetir. Şekil 3.7'de örnek bir GPU uygulaması için program akışı görülmektedir.

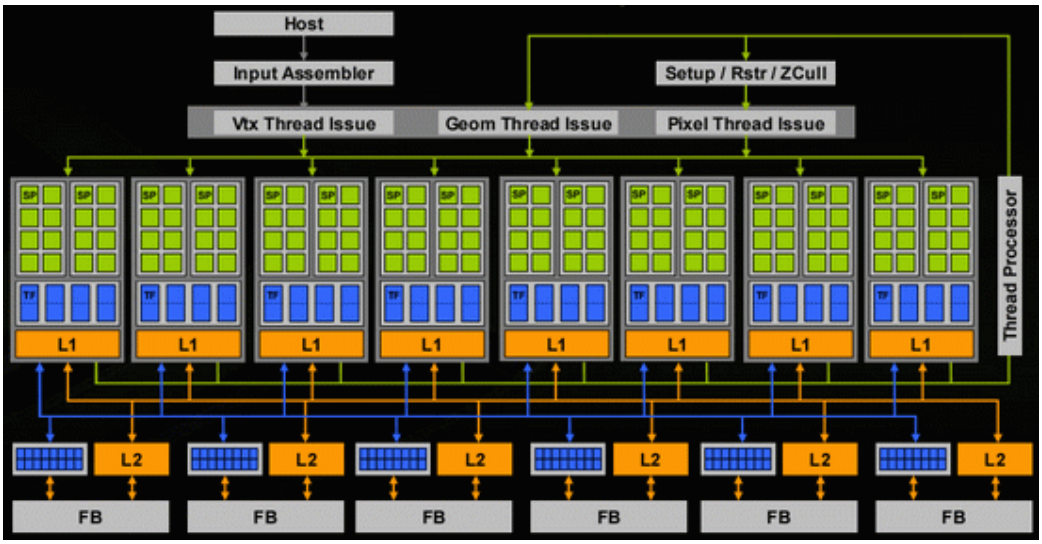


Şekil 3.7: CUDA C için örnek program akışı[5].

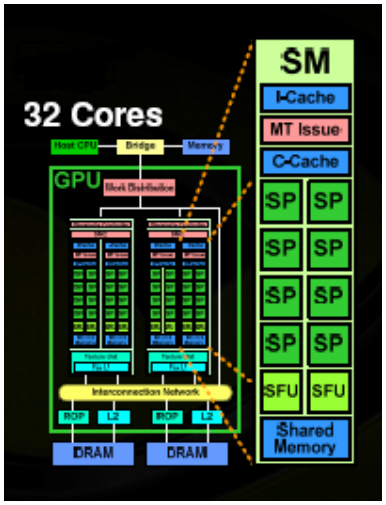
CUDA, CPU ve GPU mekanizmalarının kendileri için ayrı bellek alanları olduğunu kabul eder. Bu nedenle programlar global, değişmez ve texture bellekleri CUDA uygulama arayüzünü kullanarak yönetirler. Bu yönetim ayrıca veri alanı tahsis etme, bu alanların silinmesi, CPU-GPU platformları arasında veri transferlerinin kontrolünü de içerir.

3.1.5. Ölçeklenebilir Programlama

NVIDIA firması, CUDA dili ile hazırlanan uygulamaların Geforce 8800 ve sonrasındaki CUDA mimarisine sahip ürünlerde çalışacağını vurgulamıştır. Aslında bu ürünler arasında ciddi donanım farkları bulunmaktadır. Şekil 3.8'de NVIDIA GeForce 8800, şekil 3.9'da ise NVIDIA GeForce 9500 ürünlerine ait mimari gösterilmektedir.



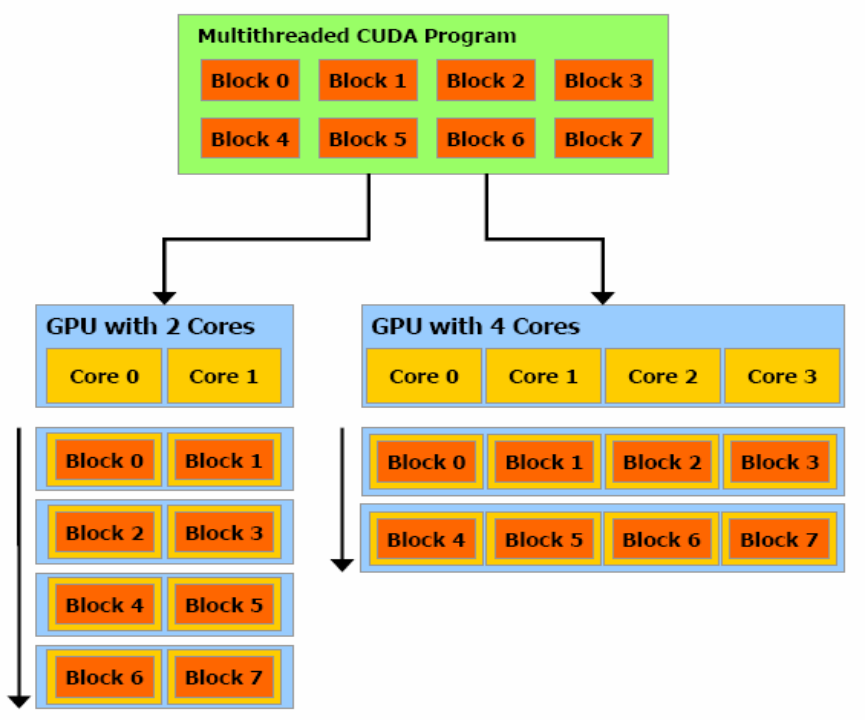
Şekil 3.8: NVIDIA GeForce 8800 mimarisi[33].



Şekil 3.9: NVIDIA GeForce 9500 mimarisi.

Şekil 3.8 ve şekil 3.9'da görüldüğü üzere iki mimarinin çekirdek sayısı oldukça farklıdır. Ancak CUDA C ile yazılmış bir uygulama, bir kerneli çağırdıktan sonra bu

uygulamanın çekirdekler arasında nasıl yönetileceği konusunda bir işlem yapmak zorunda değildir. CUDA mimarisi bunu kullanıcıdan soyutlar ve yönetimini gerçekleştirir. Şekil 3.10'da 8 bloktan oluşan bir kernelin farklı sayıda çekirdekleri olan iki mimarideki paralel çalışma şekli görülmektedir.



Şekil 3.10: Değişen çekirdek sayısına göre blokların paylaşımı[5].

3.1.6. Hesaplama Kapasitesi

Hesaplama kapasitesi büyük versiyon ve küçük versiyon numaralarıyla ifade edilir. Büyük versiyon numaraları aynı olan ürünler aynı çekirdek mimarisine sahiptirler. Küçük versiyon numarası ise mimari üzerinde yapılan güncellemelere göre değişir. Örneğin Fermi [34] mimarisine sahip ürünlerin hesaplama kapasitesi 2.x ile ifade edilir. Fermi mimarisinden önceki CUDA mimarili ürünlerin hesaplama kapasiteleri ise 1.x ile ifade edilir.

Hesaplama kapasitesine göre çoklu-işlemcilerin yetenek ve özellikleri farklıdır. Tablo 3.1'de blok, grid ve kanalların, tablo 3.2 ise bellek alanlarının hesaplama kapasitesine göre değişen özellikleri yer almaktadır.

Tablo 3.1: Hesaplama kat sayısına göre programlama modeli özellikleri.

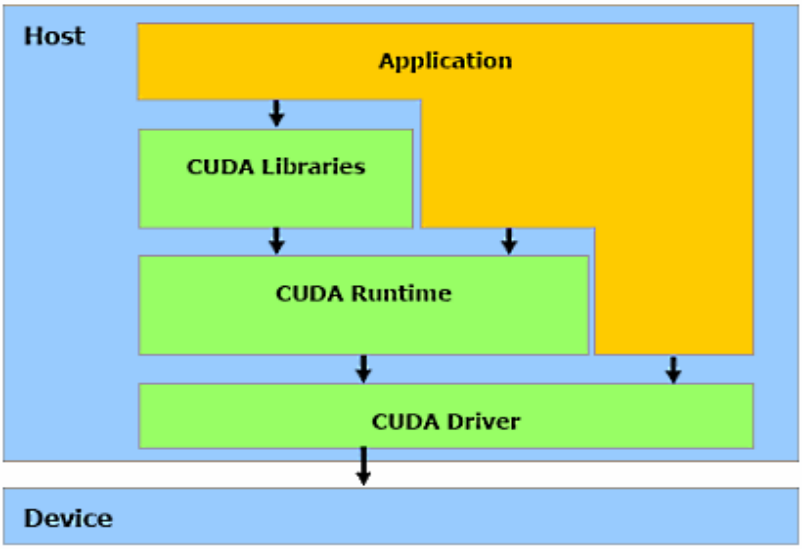
Teknik Özellikler	**Hesaplama Kapasitesi (versiyon)**				
	1.0	**1.1**	**1.2**	**1.3**	**2.x**
En az grid boyutu	2				3
En fazla grid boyutu	65535				
Bloklar için en fazla boyut sayısı	3				
Blok için x-y ekseninde en fazla kanal sayısı	512				1024
Blok için z ekseninde en fazla kanal sayısı	64				
Blok başına en fazla kanal sayısı	512				1024
Warp (planlanan kanal) sayısı	32				
Çoklu-işlemci başına sabit blok sayısı	8				
Çoklu-işlemci başına en fazla sabit kanal sayısı	24		32		48
Çoklu-işlemci başına sabit kanal sayısı	768		1024		1536

Tablo 3.2: Hesaplama kat sayısına göre bellek modeli özellikleri.

Teknik Özellikler	**Hesaplama Kapasitesi (versiyon)**				
	1.0	**1.1**	**1.2**	**1.3**	**2.x**
Çoklu-işlemci başına 32-bit yazmaç sayısı	8 K		16 K		32 K
Çoklu-işlemci başına en fazla ortak bellek (shared memory) alanı miktarı	16 KB				48 KB
Ortak bellek öbeği(shared memory bank) sayısı	16				32
Kanal başına yerel bellek alanı miktarı	16 KB				512 KB
Değişmez bellek alanı miktarı	64 KB				
Değişmez bellek için çalışma önbelleği	8 KB				
Çoklu-işlemci başına texture bellek için çalışma	Donanımdan bağımsız, 6-8 KB				

önbelleği	arasında	
Bir CUDA dizisine bağlanmış tek boyutlu bir texture alanın en fazla genişliği	8192	32768
Doğrusal (linear) bellekle ilişkilendirilmiş tek boyutlu bir texture alanın en fazla genişliği	2^{27}	
Tek boyutlu ve katmanlı texture alanlar için en büyük genişlik ve en fazla katman sayısı	8192 x 512	16384 x 2048
Bir doğrusal veya CUDA dizisine bağlanmış iki boyutlu texture bellek alanı için en büyük genişlik ve yükseklik değerleri	65536 x 32768	65536 x 65535
İki boyutlu katmanlı texture referansı için en büyük genişlik, yükseklik ve katman sayısı değerleri	8192 x 8192 x 512	16384 x 16384 x 2048
Bir doğrusal veya CUDA dizisine bağlanmış üç boyutlu texture bellek alanı için en büyük genişlik, yükseklik ve derinlik değerleri	2048 x 2048 x 2048	
Bir kernel tarafından kullanılabilecek en fazla texture alan sayısı	128	

3.1.7. Yazılım Yığını



Şekil 3.11: CUDA yazılım yığını ve elemanları[35]

CUDA yazılım yığını bir aygıt sürücüsü, bir uygulama programlama arayüzü ve yürütüm yazılımı,iki adet genel kullanım için yüksek seviye matematik kütüphanesinden oluşmaktadır. Şekil 3.11'de CUDA yazılım yığını ve elemanları görülmektedir.

3.2. OPENCV VE OPENGL KÜTÜPHANELERİ

CUDA uygulamaları, görüntü işleme konusunda her ne kadar iyi de olsa, görüntünün alınması ve render edilmesi gibi standart görüntü işleme adımlarında yetersizdir. Bu nedenle bazı ek kütüphanelerden faydalanılabilir. OpenCV ve OpengGL bu alanda en çok kullanılan kütüphanelerdendir.

3.2.1. OpenCV

OpenCV açık kaynak bir görüntü işleme kütüphanesidir. Kütüphane C ve C++ programlama dilleri ile yazılmıştır ve Linux, Windows ve Mac OS X işletim sistemlerinde çalışır. Halihazırda Python, Ruby, Matlab ve diğer dillerde geliştirmeler devam etmektedir[7, 37].

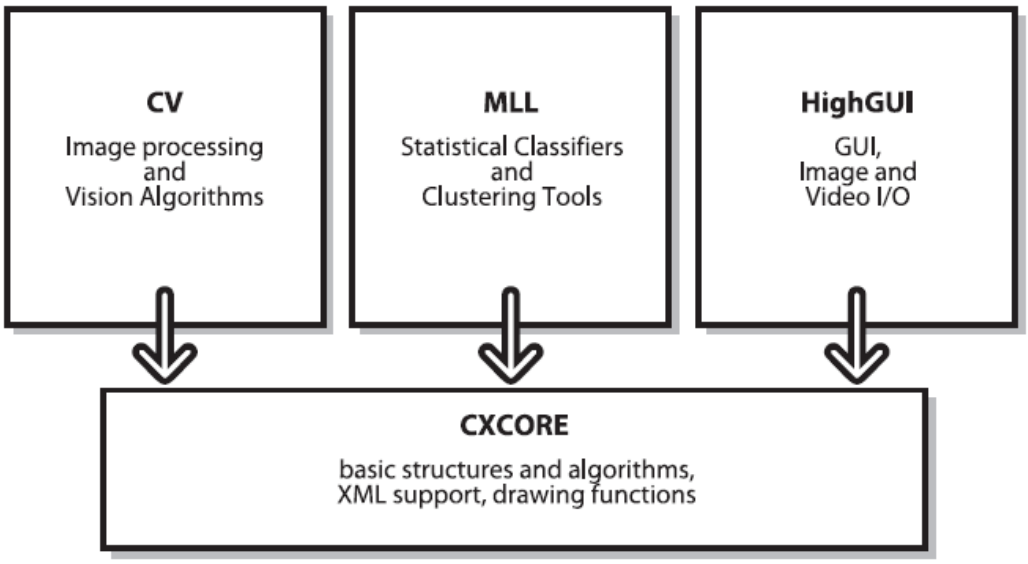
OpenCV gerçek zamanlı uygulamaları esas alarak, verimli hesaplamalar için tasarlanmıştır. OpenCV optimize C ile ile yazılmış ve çok çekirdekli işlemcileri kullanabilir olarak tasarlanmıştır[37].

OpenCV'nin amaçlarından bir tanesi gelişmiş görüntü işleme uygulamalarını çok basit ve hızlı bir şekilde oluşturmayı sağlayacak bir altyapı oluşturmaktır. Kütüphane içerisinde tıbbi görüntü işleme, kamera kalibrasyonu, güvenlik ve robotik gibi alanlarda 500'ün üzerinde tanımlanmış fonksiyon bulunmaktadır.

Bu çalışmada OpenCV kütüphanesi işlenecek olan görüntülerin dosyadan okunması işlemi ve bazı CPU-GPU performans karşılaştırma çalışmalarında kullanılmıştır.

3.2.1.1. OpenCV Yapısı ve İçeriği

OpenCV temel olarak 5 ana elemandan meydana gelir. Şekil 3.12'de bu elemanların 4 tanesi görülmektedir.



Şekil 3.12: OpenCV yapısı [37].

CV elemanı temel görüntü işleme ve yüksek seviyeli bilgisayarlı görü (computer vision) algoritmalarını içeririr. MLL elemanı özdevinimli öğrenme (machine learning) ile ilgili bazı istatiksel sınıflandırma, kümeleme gibi işlemleri gerçekleştiren araçları içerir. CXCORE temel veri yapılarını ve içeriklerini bulundurur. HighGUI elemanı görüntü ve resimler için girdi-çıktı işlemlerini gerçekleştiren fonksiyonları kapsar. Bu projedeki CUDA uygulamasında sadece HighGUI kütüphanesindeki video fonksiyonlarından yararlanılmıştır.

Yukarıda bildirilen dört elamanın dışında bir de CvAux elemanı mevcuttur. Bu konudaki dokümantasyon yeterli değildir. Ancak temel olarak bu kütüphanede yüz tanıma ve arka plan-ön plan bölümleme gibi işlemleri gerçekleştiren fonksiyonlara sahiptir[7].

3.2.1.2. Video Okuma İşlemleri

Değişen video formatları, bir görüntü işleme uygulaması için en önemli sorunlardan biridir. OpenCV, HighGUI sınıfı altında sağladığı yapılar ile bu işlemi en basit şekilde gerçeklemeyi başarmıştır. Şekil 3.13'te kullanıcıdan aldığı girdi ile dosyadan görüntü okuması yapan örnek kod görülmektedir.

```
#include "highgui.h"

int main( int argc, char** argv ) {
    cvNamedWindow( "Example2", CV_WINDOW_AUTOSIZE );
    CvCapture* capture = cvCreateFileCapture( argv[1] );
    IplImage* frame;
    while(1) {
        frame = cvQueryFrame( capture );
        if( !frame ) break;
        cvShowImage( "Example2", frame );
        char c = cvWaitKey(33);
        if( c == 27 ) break;
    }
    cvReleaseCapture( &capture );
    cvDestroyWindow( "Example2" );
}
```

Şekil 3.13: OpenCV video okuma[37].

Yukarıdaki örnekte "cvCreateFileCapture" fonksiyonu kullanıcıdan aldığı dosya adresi ile ilgili dosyanın yüklenmesini sağlar ve geri dönüş değeri olarak dosyayı kapsayan "CvCapture" yapısının referansını sağlar. "cvQueryFrame" fonksiyonu videoda sıradaki görüntünün içinde bulunduğu "IplImage" yapısının referansını döndürür. Şekil 3.14'te IplImage yapısı ve elemanları görülmektedir.

IplImage yapısı her ne kadar OpenCV fonskiyonları için uygun olsa da, OpenCV'den bağımsız olarak görüntü işleme gerçekleştiren uygulamalarda kullanılması oldukça zordur. Bu formatta temel olarak renk dizilimi BGR (blue-green-red/mavi-yeşil-kırmızı) şeklindedir. Görüntü işlemede ise genel olarak RGB (red-green-blue/kırmızı-yeşil-mavi) düzeni kullanılır. Eğer OpenCV ile görüntüleri okumak ve programa aktarmak isteniliyorsa BGR'den RGB'ye çevrim gereklidir. Bu çevrim uygulamalar için performansın düşmesi demektir.

IplImage verilerin device (GPU) belleğinde saklanabilmesi için CUDA uygulama arayüzü ile device belleğinde gerekli alanın açılması ve ilgili verinin host (CPU) belleğinden device üzerinde açılan alana kopyalanması gerekmektedir. Bu da uygulamalar için bir başka performans sorunu demektir.

```
typedef struct _IplImage {
  int                  nSize;
  int                  ID;
  int                  nChannels;
  int                  alphaChannel;
  int                  depth;
  char                 colorModel[4];
  char                 channelSeq[4];
  int                  dataOrder;
  int                  origin;
  int                  align;
  int                  width;
  int                  height;
  struct _IplROI*      roi;
  struct _IplImage*    maskROI;
  void*                imageId;
  struct _IplTileInfo* tileInfo;
  int                  imageSize;
  char*                imageData;
  int                  widthStep;
  int                  BorderMode[4];
  int                  BorderConst[4];
  char*                imageDataOrigin;
} IplImage;
```

Şekil 3.14: IplImage yapısı ve elemanları[37].

Bu çalışma için geliştirilen CUDA uygulamalarında veri yapıları ve veri alanları arasındaki çevrim ve transferlerin performansa olan etkilerini en aza indirmek amacıyla CUDA teknolojisinin OpenGL Interoperability (OpenGL birlikte çalışabilirlik) özelliğinden faydalanılmıştır. Bölüm 3.3'te konuya ilişkin detaylar yer almaktadır.

3.2.2. OpenGL

OpenGL (Open Graphics Library-Açık Grafik Kütüphanesi), gelişmiş donanım desteğini kullanarak hem iki hem de üç boyutlu grafikleri ekrana çizmek için kullanılan ücretsiz bir grafik arabirimidir.Windows, Linux, MacOS ve Solaris gibi birçok işletim sisteminde yaygın olarak ve Playstation başta olmak üzere bazı oyun konsollarınca desteklenir. Donanım tarafında ise SGI, ATI, NVIDIA veya Intel gibi büyük üreticiler her grafik kartında OpenGL desteği sunar.

1992 yılında ilk taslağı yaratılmış olan bu standart, günümüzde 4.1 sürümüne ulaşmıştır ve 250 nin üzerinde fonksiyona sahiptir. Çoklu platform desteği içeren uygulamalar ve özellikle de deneysel ve bilimsel araçlarda açık arayla önde ve standart olarak kullanılmakta olan platform OpenGL'dir.

OpenGL taşınabilirdir. Bu kitaplık işletim sisteminden ve işletim sisteminin çalıştığı platformdan bağımsızdır. Nasıl ki ekrana yazı yazmak kullanıcıdan veri almak ANSI C'de C dilinden bağımsız olarak kütüphane tarafından printf() ve scanf() gibi işlevlerle standartlaştırılmış ve hangi işletim sistemiyle çalışırsanız çalışın bu iki işlev aynı işi yapıyorsa, OpenGL kitaplığıda ekrana grafik çizmeyi standartlaştırmıştır. OpenGL sayesinde grafik kartının modeli veya işlemcinin mimarisi gibi donanımsal etkenlerden bağımsız programlama yapılır. Aynı zamanda işletim sisteminden de bağımsız programlama yapılır. Kolay kullanım ve bu "taşınabilirlik" özellikleri nedeniyle OpenGL popüler bir araç olmuştur.

İşletim sisteminden bağımsızdır OpenGL kullanan bir programı işletim sisteminizde çalıştırmanız için öncelikle işletim sisteminizde programın çalışırken kullanacağı işlevleri içeren kitaplığın bulunması gerekir, bu kitaplıkların genel adı runtime-library (çalışma anı kitaplığı)'dir.

OpenGL çalışma anı kitaplığı Linux, Unix, Mac OS, OS/2, Windows 95/98/NT/2000, OPENStep ve BeOS işletim sistemlerinde hali hazırda mevcuttur. Windows işletim sistemi ailesinde standart olarak gelir.

Pencere yöneticisinden bağımsızdır. OpenGL kullanılarak yazılmış programlar, Win32, MacOS ve X-Window pencere yöneticilerinde sorunsuz çalışırlar. Birçok programlama dilinden kullanılabilir Ada, C, C++, C# (SharpGL adı verilen sınıflar sayesinde), Fortran, Python,Perl ve Java programlama dilleri kullanılarak OpenGL kitaplığından faydalanılabilir[36, 38].

3.3. CUDA PROGRAMLAMA

CUDA programlama için mevcutta iki adet arayüz bulunmaktadır. Bunlar CUDA C ve CUDA sürücü uygulama arayüzüdür. Genel olarak uygulamalar bu arayüzlerden sadece birini tercih ederler ancak teknik olarak ikisininde aynı program içerisinde kullanılması mümkündür.

CUDA C dili, standart C diline yapılan en az düzeyde eklentilerle oluşturulmuş bir programlama dilidir. İçerisinde bu dile ait herhangi bir komut bulunduran bir kaynak dosyası nvcc (NVIDIA CUDA Compiler) ile derlenmelidir. Gelen eklentilerle birlikte yazılımcılar kernel tanımlarını bir C fonksiyonu gibi gerçekleştirebilirler. Yeni söz dizilimleri ile her fonksiyon çağrısından önce grid ve blok boyutları kolaylıkla belirlenebilir.

CUDA sürücü API(application programming interface-uygulama programlama arayüzü), kernelleri ikili veya assembly kod modulleri şeklinde yüklemeyi sağlayan fonksiyonlara sahip, düşük seviyeli bir C API'sidir. Söz konusu assembly kodlar, C dilinde yazılmış kernellerin derlenmesi ile elde edilir.

CUDA C, runtime API'si ile birlikte gelir. Hem CUDA C runtime API hem de sürücü API'si GPU belleğinde alan tahsisi ve belleğin silinmesi, CPU platformu ve GPU platformu arasında veri transferi, sistemlerin yönetilmesi ve birden fazla GPU'nun kontrolünü sağlayacak fonskiyonlara sahiptir.

Temel olarak CUDA runtime API'si sürücü API'sinin üzerine inşa edilmiştir. İçerik yönetimi, modül yönetimi ve hata mesajları aynı veya örtüşecek derecede benzerdir. CUDA sürücü API'si programlaması ve hata ayıklaması zor bir seçenek de olsa, assembly ve ikili kodları ele alması ve dillerden bağımsız olmasıyla daha iyi kontrol sağlar[5, 39].

3.3.1. NVCC ile Derleme

NVCC, C kodunun derlenmesini kolaylaştıran bir derleme sürücüsüdür. Sağladığı basit ve bilindik komut satırı argümanları ile derleme işleminin çeşitli basamaklarını gerçekleştiren araçların uyarılması ve çalışmasını sağlar.

3.3.1.1. Derleme Akışı

Derlenecek olan kaynak kod dosyaları içerisinde host kodu ve device kodu birlikte bulundurabilir. NVCC'nin temel akışı device ve host kodunun birbirinden ayrılmasını ve device kodunun assembly (PTX) veya ikili (cubin) formunda, host kodunun ise başka bir derleme aracı ile derlenecek şekilde C kodu veya host derleyicisi tarafından derlenmek üzere nesne kodu biçiminde üretilmesini içerir.

Uygulamalar bu aşamadan sonra:

1. CUDA sürücü API'sini kullanarak PTX kodunu veya cubin nesnelerini yükleyebilir veya çalıştırabilir, eğer mevcutsa host kodunu dikkate almayabilirler.
2. Host kodunu, içerisindeki CUDA nesne ve kernellerinin derlenmesi ile oluşan nesne ve çalıştırabilir dosyalarla ilişkilendirebilir.

Herhangi bir uygulama tarafından runtime'da yüklenen PTX kodu daha sonra aygıt sürücüsü tarafından ikili koda dönüştürülür. Buna just-in-time (anında) derleme denir. Bu özellik uygulamanın yükleme zamanını artırır, fakat uygulamaların derleyicilerdeki son güncellemelerden faydalanmalarını sağlar. Ayrıca bu özellik sayesinde sonradan gelişen aygıtlarda, önceden derlenmiş programların çalışması mümkün olur[5, 31, 39].

3.3.1.2. İkili Uyumluluk

İkili kod, mimariden mimariye değişen bir kod türüdür. Bir cubin nesnesi derleyicide –code seçeneği ile hedef mimari belirtilerek üretilir. Örneğin –code=sm_13 seçeneği ile derlemek hesaplama kapasitesi 1.3 olan aygıtlar için ikili kodun üretilmesini sağlar. NVIDIA ürünlerinde ikili uyumluluk bir sonraki küçük versiyon değişimlerinde garanti edilir ancak mevcut versiyon için üretilen ikili kodun bir önceki versiyonda çalışması garanti edilmez.

3.3.1.3. PTX Uyumluluk

Bazı PTX komutları sadece yüksek hesaplama kapasitesine sahip aygıtlarda çalışır. Örneğin global bellek üzerinde atomik işlemleri gerçekleştirebilmek için hesaplama kapasitesi 1.1 veya üzeri olan bir GPU gereklidir. Çift hassasiyetli işlemleri ancak hesaplama kapasitesi 1.3 ve üzeri olan cihazlarda çalışır. Bu işlemi gerçekleştirmek için –code=sm_13 veya daha ileri bir mimariyi ifade eden komut satırı argümanını kullanmak gerekir.

Herhangi bir hesaplama kapasitesine sahip donanım için üretilmiş PTX kodu, aynı veya daha yüksek hesaplama kapasitesine sahip bir başka aygıt tarafından kullanılabilir.

3.3.1.4. Uygulama Uyumluluğu

Bir kodu belirli bir hesaplama kapasitesi sınıfındaki cihazlarda çalıştırabilmek için ilgili hesaplama kapasitesi ile uyumlu PTX kodunu uygulama tarafında yüklenmesi gereklidir. Mevcutta olmayan ileri mimarilere yönelik geliştirilen kodların, ilgili mimaride çalışabilmesi için just-in-time derleyici ile derlenmesi gerekir.

3.3.1.5. C/C++ Uyumluluğu

Derleyicinin ön yüzü CUDA kaynak dosyalarını C++ sentaksına göre derler. Host kodunda tam C++ desteği vardır ancak device kodunda kısıtlı C++ desteği mevcuttur. Son olarak “void” türdeki işaretçiler void olmayan türdeki işaretçilere atanamaz. Bu atama için tür çervrimi gereklidir.

3.3.2. CUDA C

CUDA C, C programlama diline aşina geliştiriciler için basit bir çözüm sunar. Bu kolaylık CUDA C dilinin standart C diline çok az miktar eklenti ile oluşturulmasından kaynaklanmaktadır.

CUDA runtime, “cudart” dinamik kütüphanesi ile oluşturulmuştur. Fonksiyonlara “cuda” ön eki getirilerek yazılımcılara sunulmuştur. CUDA C dilinde runtime başlangıç fonksiyonu bulunmaz, bunun yerine runtime fonksiyonlarından birine yapılan ilk çağrıda runtime başlatılır. Runtime tanımla işlemi host üzerindeki bir kanal tarafından

tetiklendikten sonra, bu kanal tarafından yönetilen kaynaklar, sadece söz konusu kanal tarafından yönetilebilir.

3.3.2.1. GPU Belleği

GPU belleği, doğrusal veya CUDA dizisi olarak tahsis edilebilir. CUDA dizileri texture belleği okumayı sağlayan opak bellek düzenleridir. Doğrusal bellek ise 1.x hesaplama kapasitesine sahip aygıtlarda 32-bit, 2.x hesaplama kapasitesine sahip aygıtlarda ise 40-bit adres alanında bulunur. Bu bellek türünde oluşturulan nesneler arasındaki bağlantılar, ikili ağaçlardaki gibi işaretçiler aracılığıyla sağlanır.

Doğrusal bellekte alan tahsisi genellikle cudaMalloc fonksiyonu aracılığıyla yapılır. Tahsis edilen alanın temizlenmesi ise cudaFree fonksiyonu aracılığıyla gerçekleştirilir. Host ve device arasındaki veri transferleri ise cudaMemcpy fonksiyonu ile gerçekleştirilir. Şekil 3.15'te ve 3.16'da bu fonksiyonların kullanımı gösterilmiştir.

```
int N = ...;
size_t size = N * sizeof(float);

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

Şekil 3.15: cudaMalloc fonksiyonun kullanımı.

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

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

Şekil 3.16: cudaMemcpy ve cudaFree fonksiyonlarının kullanımı.

3.3.2.2. Texture Bellek

Texture bellek keneller taradından API'de tanımlanan fonksiyonlar aracılığıyla okunur. Alınan verinin ilk parametresi texture referansı olarak bilinir. Bir birinden farklı texture referansları aynı texture ile veya bellekte kesişen texture alanlar ile ilişkilendirilebilir.

Texture referansı birçok elemandan meydana gelir. Bunlardan birtanesi texture alanın boyut bilgisini koordinatlar halinde bulundurur. Bir elaman, tek boyutlu texture alanlar için genişlik bilgisini, iki eleman ise yüzey bilgilerini, üç eleman ise yüzey ve derinlik bilgilerini bulundurur. Referans içerisindeki her bir elemana "texel" denir. Diğer özellikler ise verilen koordinat değerlerinin nasıl yorumlanacağının ve işleneceğinin bilgisini içerir.

3.3.2.3. Çoklu GPU

Bir host birden fazla GPU'ya sahip olabilir. Bu GPU'lar numaralandırılabilir, özellikleri sorgulanabilir ve biri seçilerek üzerinde kerneller çalıştırılabilir. Birden fazla host kanalı seçilen cihazlardan biri üzerinde aynı anda çalışabilir. Ancak bir host kanalı aynı anda birden fazla GPU'ya erişemez. Ayrıca aynı GPU'üzerinde çalışan farklı host kanalları birbirlerinin kaynaklarına erişemezler. Şekil 3.17'de host sistemindeki CUDA desteği mevcut olan aygıtları numaralandıran ve özelliklerini okuyan kod bloğu görülmektedir.

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);
    if (dev == 0) {
        if (deviceProp.major == 9999 && deviceProp.minor == 9999)
            printf("There is no device supporting CUDA.\n");
        else if (deviceCount == 1)
            printf("There is 1 device supporting CUDA\n");
        else
            printf("There are %d devices supporting CUDA\n",
                   deviceCount);
    }
}
```

Şekil 3.17: CUDA aygıtlarının numaralandırılması ve özelliklerine erişim[39].

CUDA GPU yönetim fonksiyonlarından herhangi biri çağırılmadıkça sıfır numaralı cihaz varsayılan cihaz olarak kullanılır. Diğer cihazlar arasında seçim cudaSetDevice fonksiyonu ile yapılır. Bundan sonraki cudaSetDevice çağrıları cudaThreadExit çağırılmadıkça hata verir. "cudaThreadExit" fonksiyonu çağırıldıktan sonra ilgili host

kanalı için atanan kaynaklar serbest kalır. Bundan sonraki ilk API kullanımı runtime'ın yeniden oluşturulmasını sağlar[5, 31, 35, 39].

3.3.2.4. Asenkron Eş Zamanlı Çalışma

CPU ve GPU arasındaki eş zamanlı çalışmayı kolaylaştırmak amacıyla bazı fonksiyon çağrıları eş zamanlı olarak gerçekleştirilir. GPU isteği tamamlamadan, kontrol istekte bulunan CPU kanalına geçer. Asenkron olarak yapılan bu işlemler:

- Kernel çağırımı
- GPU-GPU veri transferleri
- CPU-GPU arasındaki 64KB ve daha az veri transferleri
- "Async" ön eki ile başlayan fonksiyonlar aracılığıyla yapılan veri transferleri
- Bellek atama işlemleridir.

Geliştiriciler, uygulamalar için global olarak asenkron çalışmayı engelleyebilirler. Bunun için CUDA_LAUNCH_BLOCKING ortam değişkeninin 1 olarak atanması yeterlidir. Ancak bu güvenilir bir uygulama sağlamak için değil, hata ayıklama modunda çalışmak için önerilir.

Hesaplama kapasitesi 2.x düzeyinde olan bazı aygıtlar eş zamanlı kernel çalıştırabilir. Bunun için cudaGetDeviceProperties fonksiyonu aracılığıyla aygıtın özelliklerini okunur ve concurrentKernels özelliği kontrol edilir. Aynı aygıtta en fazla 16 kernel aynı anda çalışabilir. Bir CUDA kaynağında çalışan kernel, başka bir kaynakta çalışan CUDA kerneliyle eş zamanlı olarak çalışamaz. Bellek işlemlerini çokça gerçekleştiren kernellerin eş çalışması az tercih edilen bir seçenektir.

3.3.2.5. Senkon Fonksiyon Çağrıları

Senkron bir fonksiyona çağrıda bulunulduğunda, GPU işlemleri bitirmeden host kanalına kontrolü döndürmez. Bu durumda nasıl davranılacağını "cudaSetDeviceFlags" fonskiyonu ile belirlemek mümkündür. Bu fonskiyon ile 4 farklı mod seçilebilir:

- cudaDeviceScheduleAuto değeri ile varsayılan mod seçilebilir. Bu durumda CUDA aktif CUDA içeriklerinin sayısı(c)'nı mantıksal işlemci sayısı(p) ile karşılaştırır. Eğer c>p ise, CUDA diğer işletim sistemi kanallarına öncelik veririr. Aksi takdirde aktif döngü durumuna geçer.
- cudaDeviceScheduleSpin değeri CUDA'nın GPU işlemleri bitene kadar dögü durumuna geçmesini sağlar. Bu GPU üzerindeki zaman kaybını azaltabilir ancak paralel çalışmakta olan CPU kanalları için performans sorunu doğurur.
- cudaDeviceScheduleYield değeri CUDA'nın GPU işlemlerini beklerken öncelik durumunu diğer kanallara vermesini sağlar. Bu değer ile GPU üzerinde zaman kaybı artar ancak, paralel çalışan CPU kanalları için performans artışı sağlanır.
- cudaDeviceBlockingSync değeri ise CUDA'nın CPU kanalını GPU işlemleri bitene kadar durdurmasını sağlar.

3.3.2.6. CUDA-OpenGL İşbirliği

CUDA hesaplama yeteneklerinden faydalanarak görüntüleri oluşturabilmek için CUDA'nın ek teknolojilere ihtiyacı vardır. Performans açısından CUDA ile oluşturulmuş görüntüleri en iyi sunacak olan teknolojilerden biri ise OpenGL'dir.

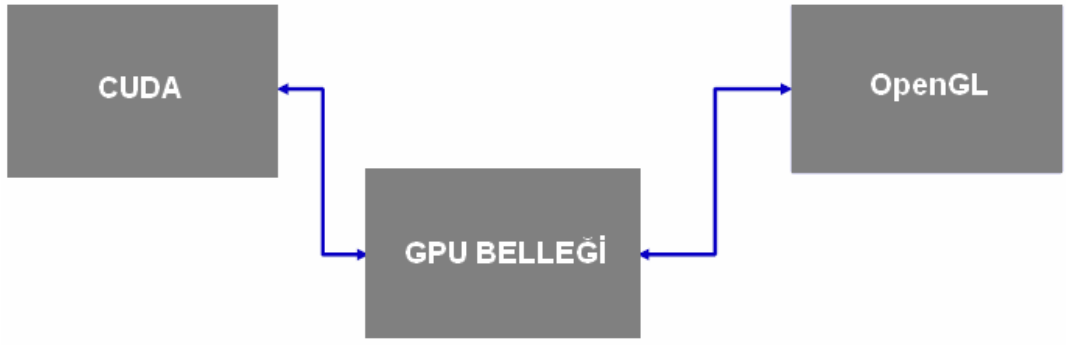
Grafik işbirliği kısaca ilgili CUDA kaynaklarının ortak kullanım amacıyla OpenGL çağrılarının erişimine açılması, aynı zamanda ilgili OpenGL kaynaklarının CUDA adresleriyle eşleştirilmesinden ibarettir. Böylece önemli performans sıkıntılarına yol açan CPU-GPU ve GPU-CPU veri transfer işlemlerinin etkileri ortadan kaldırılır.

İşbirliği için öncelikle kullanılacak olan kaynak CUDA'ya kayıt edilmelidir. Aksi takdirde bu kaynaklar ilgili fonksiyonlar tarafından kullanılamaz. Fonksiyonlar kullanımları ile birlikte CUDA grafik kaynaklarına "cudaGraphicsResource" türünde bir yapıyı gösteren işaretçiyi geri dönüş değeri olarak verir. Kaynakların CUDA'ya

kaydedilmesi masraflı bir işlem olduğu için kaynaklar için sadece bir kere gerçekleştirilir. “cudaGraphicsUnregisterResource” ile kaynakların kayıtlarının silinmesi işlemi gerçekleştirilir.

Bir kaynak CUDA’ya bir kere kaydedildikten sonra, “cudaGraphicsMapResources” ve “cudaGraphicsUnmapResources” fonksiyonları kullanılarak istenilen sayıda eşleştirme ve eşleşme kaldırma işlemi yapılabilir. “cudaGraphicsResourcesSetMapFlags” fonksiyonu ile bu eşleştirme sonrasında kaynağın erişim izinlerinin nasıl olacağı belirlenir. Kernellerin eşlenmiş veri tamponu içerisindeki verilere erişimi “cudaGraphicsResourcesGetMappedPointer” fonksiyonu ile elde edilen işaretçi üzerinden gerçekleştirilir. “cudaGraphicsSubResourcesGetMappedArray” fonksiyonu ile de eşlenmiş CUDA dizilerine kerneller tarafından erişim sağlanabilir.

Örnek bir GPU-OpenGL işbirliği bir birini takip birkaç adımdan oluşur. Şekil 3.18’de CUDA-OpenGL işbirlinin temel yapısı görülmektedir.

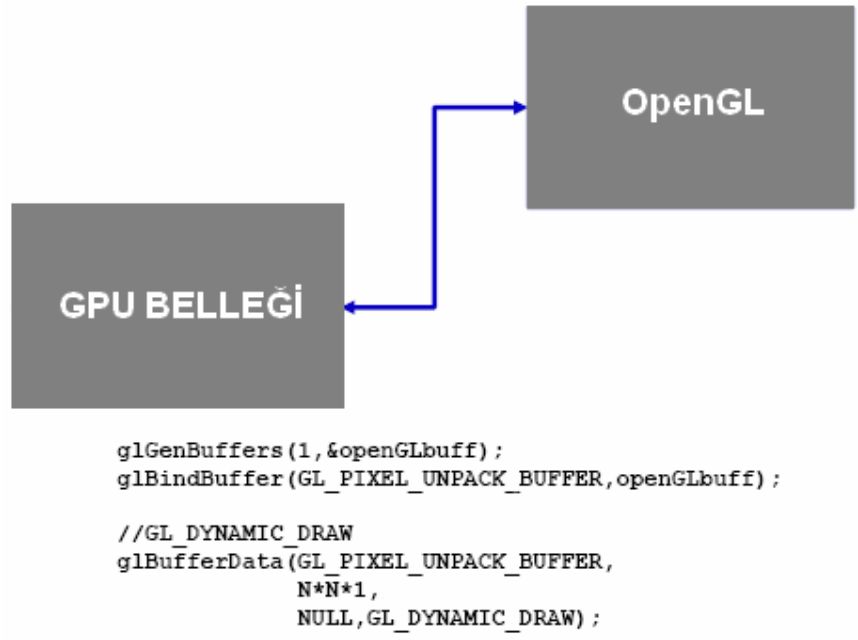


Şekil 3.18: CUDA-OpengGL işbirliğinin temel yapısı.

İlk adım OpenGL ile GPU belleği arasında erişim izinlerinin sağlanması adımıdır. Bu adımdaki işlemin görseli ve gereken örnek OpenGL kodu şekil 3.19’da görülmektedir.

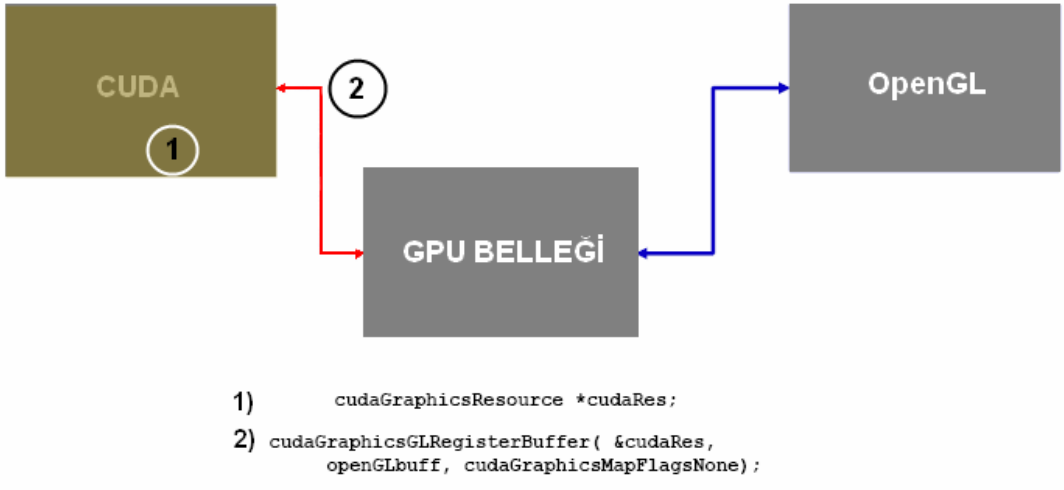
İlk önce piksel tampon nesnesi oluşturulur. Sonraki adımda bu nesne ile hedef arasında ilişki kurulur. “glBindBuffer” OpenGL tarafında bu işlemi gerçekleştiren fonksiyondur. Bu işlem ile hedefe daha önceden yapılan bağlantılar silinir. Piksel veri tamponu için OpenGL’de veri alanı oluşturmak birinci adımın son işlemidir. Bu da “glBufferData” fonksiyonu ile gerçekleştirilir. Bu fonksiyon ile piksel veri tamponu için ne kadar yer

ayrılacağı belirtilir. Eğer mevcut ise veri kaynağına ait bir işaretçi ile OpenGL tarafında oluşturulacak alana kopyalanacak verinin de kaynağı bildirilmiş olur. Şekil 3.19'da NxN boyutunda bir alan örnek olarak açılmıştır.



Şekil 3.19: OpenGL-GPU Bellek ilişkisi.

CUDA-OpenGL işbirliğinin ikinci adımı ise CUDA kaynağının belirlenmesi ve bu kaynağın eşleştirilmesidir. Şekil 3.20'de bir numaralı görsel CUDA kaynağının oluşturulması, iki numaralı görsel ise CUDA kaynağının eşlenmesini göstermektedir.



Şekil 3.20: CUDA kaynak eşleştirmesi.

3.3.3. CUDA C GELİŞTİRME ARAÇLARI

CUDA C projesi geliştirebilmek için uygulama geliştirme ortamında mutlaka olması gereken araçların yanında bir de projenin ilerleyişini hızlandıracak, hata ayıklamayı ve kodlamayı bir sorun olmaktan çıkaracak ek araçlara ihtiyaç duyulur. Bu bölümde bu araçların tanıtımına yer verilmiştir.

3.3.3.1. CUDA Araç Kiti

Bir CUDA projesi geliştirebilmek için öncelikle CUDA araç kiti kurulumu yapılır. Derleyici ve sürücüler başta olmak üzere, CUDA destekli uygulama geliştirmek için zorunlu olan bütün parçalar CUDA araç kiti ile birlikte gelir.

3.3.3.2. CUFFT Kütüphanesi

CUDA araç kiti iki önemli destek kütüphanesi ile birlikte gelir. Bunlardan ilki hızlı Fourier dönüşüm kütüphanesi olarak bilinen CUFFT'tur. Bu kütüphane birçok faydalı özellik sunmaktadır. Bunlardan bazıları:

- Gerçek ve karmaşık sayılardan oluşan girdi verileri için 1D, 2D ve 3D dönüşümler.
- Çok sayıda bir boyutlu dönüşümü paralel olarak yapabilen toplu çalıştırma özelliği.
- Her bir boyutta 2'den 16384'e kadar değişen 2D ve 3D dönüşümler.
- 8 milyona kadar elemana sahip girdilerde 1D dönüşüm.
- Gerçek ve karmaşık sayılardan oluşan girdi verileri için yerinde ve uzakta dönüşümler.

NVIDIA firması bu kütüphaneyi ücretsiz olarak sunmaktadır[41].

3.3.3.3. CUBLAS Kütüphanesi

Hızlı Fourier dönüşüm kütüphanesine ek olarak, temel lineer cebir rutinlerinin uygulandığı bir kütüphane de CUDA araç kiti içinde sunulmaktadır. BLAS(basic linear algebra subprograms) bir fortran kütüphanesidir. NVIDIA bu kütüphane ile BLAS'ın yaygın kullanımını CUDA platformunda da desteklemiştir. Bir fortran kütüphanesi

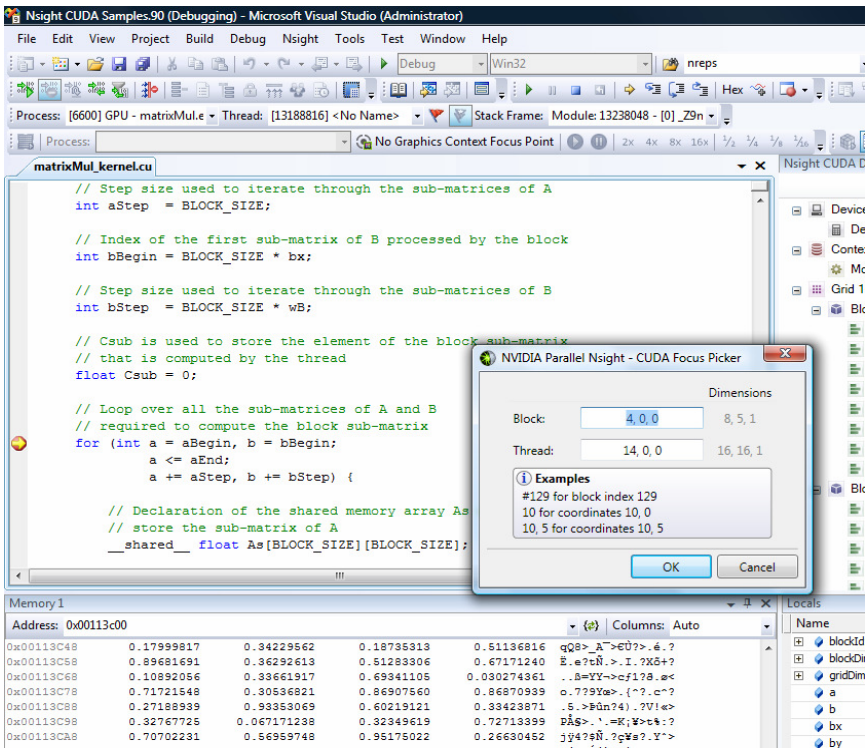
olması itibari ile dizilerde ilk koordinat kolonu gösterir. C ve C++ ise satır ilk koordinat değeridir. NVIDIA, BLAS ile oluşturulmuş uygulamaların CUDA platformunu geçişini kolaylaştırmak için dizilerde kolon öncelikli anlayışı korumuştur.

3.3.3.4. NVIDIA GPU Hesaplama SDK'sı

NVIDIA GPU hesaplama SDK(software development kit)'sı CUDA araç kitinden ve sürücülerden ayrı olarak dağıtılır. İçerisinde binlerce CUDA kod örneği bulunur. Burada sunulan bütün örnekler her platformda çalışabilir.

3.3.3.5. NVIDIA Parallel Nsight

GPU kodunun Windows işletim sisteminde, hata ayıklama modunda çalıştırılabilmesi için NVIDIA tarafından sunulmuştur. Parallel Nsight, binlerce kanaldan oluşan uygulamaların hata ayıklama modunda çalıştırılmasını mümkün kılar.



Şekil 3.21: Parallel Nsight örnek ekranı.

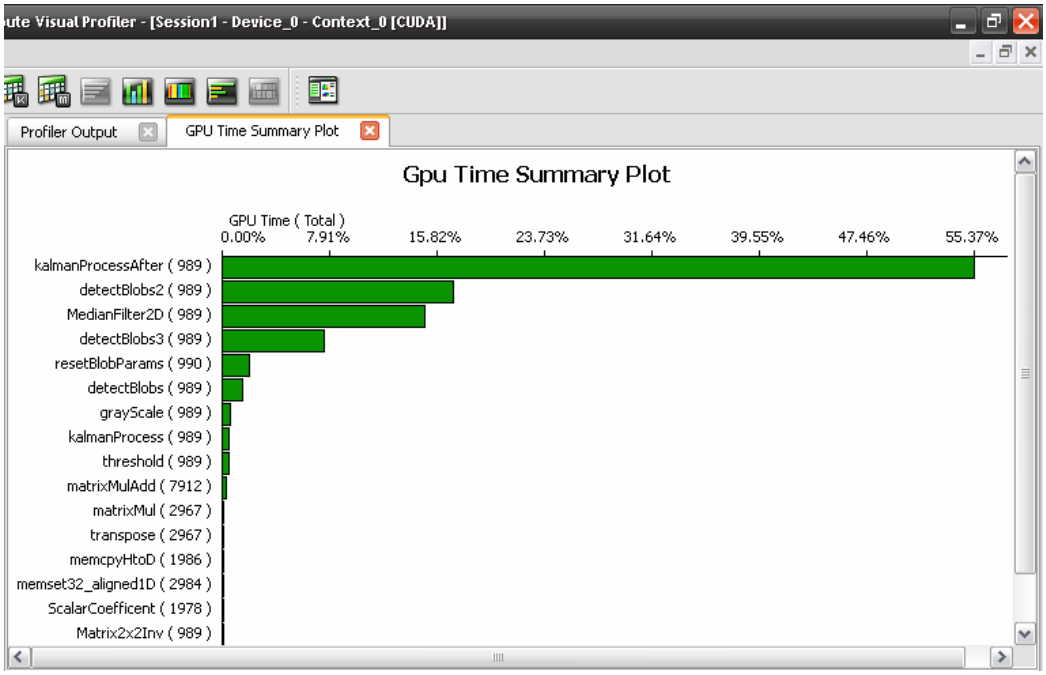
Bu program Microsoft Visual Studio[40] için sunulan ilk GPU/CPU entegre hata ayıklayıcıdır. Kullanıcılar CUDA C kod bloklarında kesmeler kullanarak, uygulamaların o noktaya eriştikleri andaki bellek durumunu bellek penceresinden gözlemleyebilirler.

3.3.3.6. NPP Kütüphanesi

NPP(NVIDIA performance primitives) kütüphanesi hızlı veri işleme fonksiyonları bir araya getiren bir kütüphanedir. Bu kütüphane ağırlıklı olarak resim ve görüntü işleme alanlarında çalışmalarını sürdüren yazılım geliştiricileri hedef olarak kabul etmiş ve daha çok bu alanlarda kullanılabilecek fonksiyonlar sunmuştur.

3.3.3.7. CUDA Visual Profiler

Profiler türündeki yazılımlar, herhangi bir dilde geliştirilen uygulamanın bellek kullanım miktarı, veri aktarım hızı, işlem hızı ve bunlara benzer performans metriklerini değerlendiren ve yazılımcının geliştirdiği uygulamayı teknik olarak analiz etmesini sağlayan yazılımlardır.

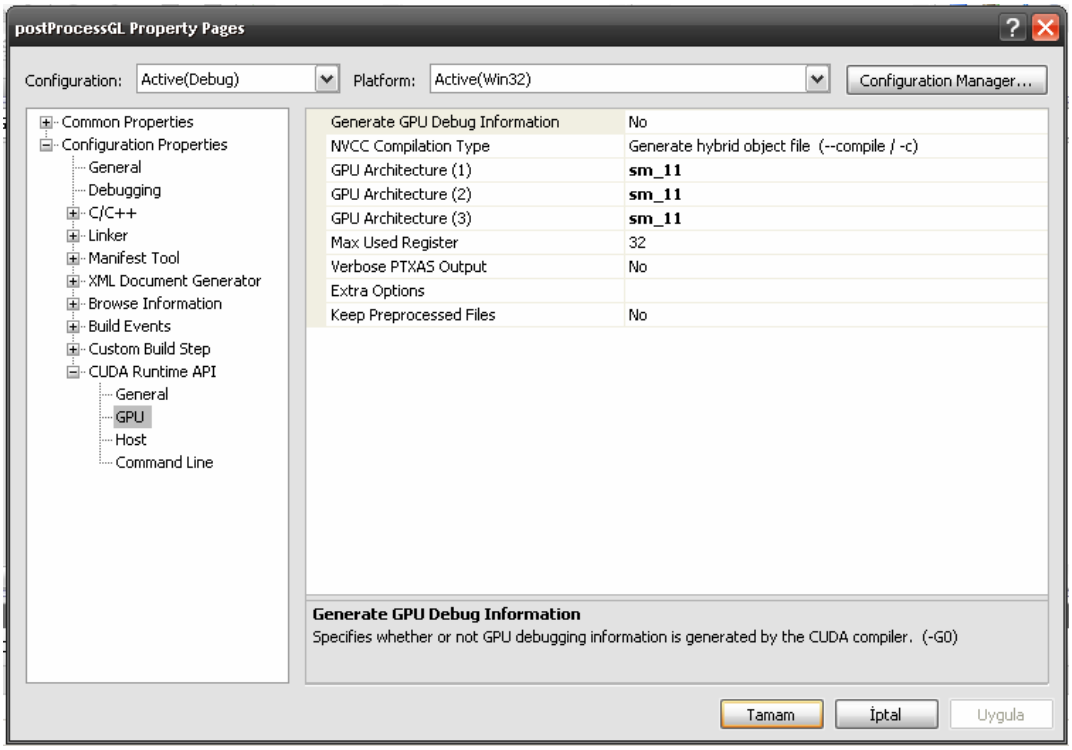


Şekil 3.22: Visual Profiler örnek ekranı.

CUDA yüksek performansı hedefleyen bir platform olduğu için, geliştirilen yazılımın performansını etkileyecek kod katarlarından sakınmak gerekir. Bu anlamda geliştirilen kodun test ve analizi önemlidir. NVIDIA firması bu amaçla Visual Profiler aracını yazılım geliştiricilere sunmaktadır. Şekil 3.22'de örnek bir Visual Profiler ekranı görülmektedir.

3.3.3.8. Visual Studio 2008 ve CUDA Entegrasyonu

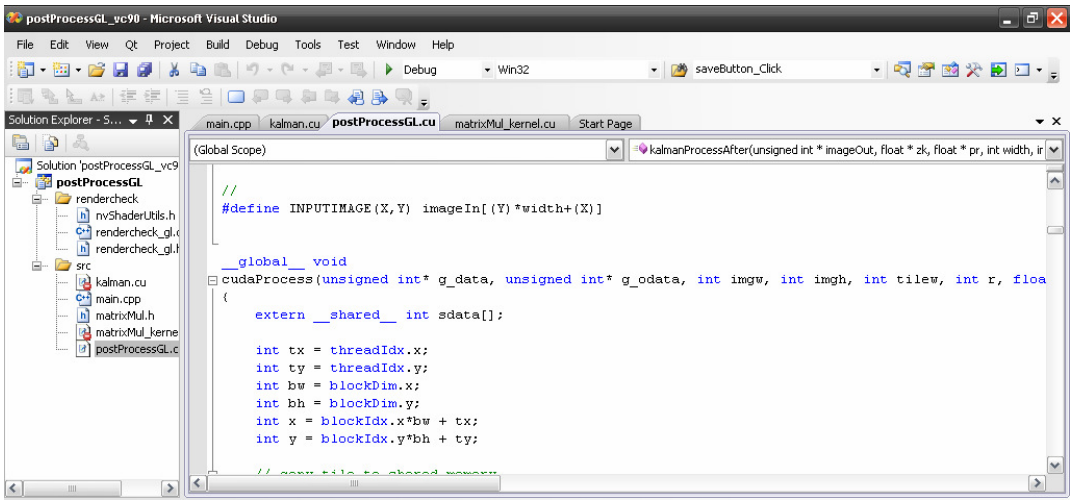
Visual Studio[40] gelişmiş bir program geliştirme ortamıdır. Derleme, hata ayıklama, çalıştırma gibi işlemleri kolayca yapmanızı ve kodu düzenli yazmanızı sağlar. CUDA C dili ile geliştirme yaparken karşılaşılan başlıca sorun yazılan kodun derlenmesidir. Visual Studio 2008-CUDA entegrasyonu ile bu zorluklar kolayca aşılabilir. Ayrıca CUDA C ile gelen eklentilerin yönetilmesi ve yeni anahtar kelimelerin diğer komut satırlarından ayırt edilebilmesi için de entegrasyondan faydalanılır.



Şekil 3.23: Visual Studio 2008'de CUDA projesi için derleyici ayarları.

Entegrasyon işlemini gerçekleştirmek için CUDA SDK ile gelen derleme kurallarını içeren dosyaların Visual Studio kurulum dizinine eklenmesi yeterlidir. Şekil 3.23'te

derleme için CUDA nvcc ayarlarını düzenlenme ekranı, şekil 3.24'de ise CUDA C anahtar kelimelerinin renklendirildiği geliştirme ortamı görülmektedir.



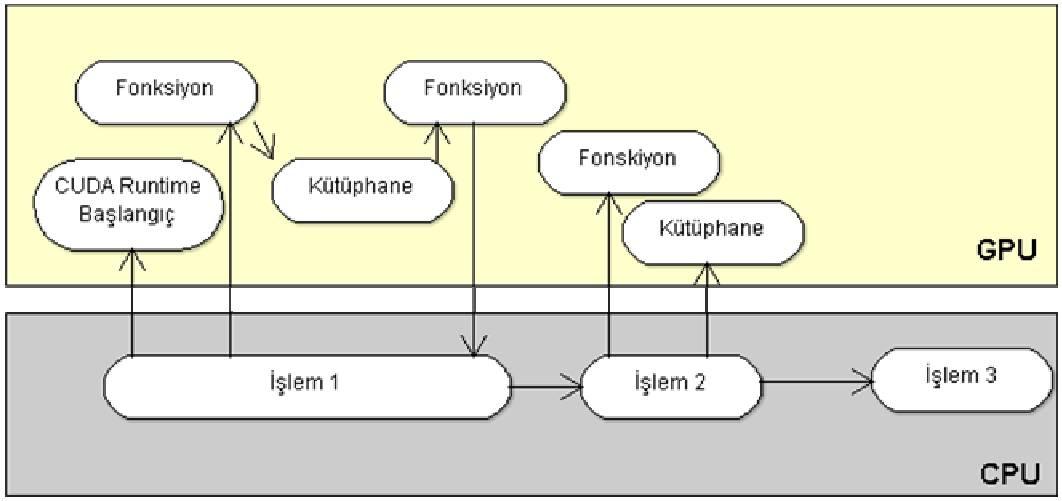
Şekil 3.24: Visual Studio 2008'de CUDA C desteği.

4. BULGULAR

Bu bölümde tez için geliştirilen uygulamaların ayrıntılarına yer verilmiştir. Ayrıca çeşitli performans ölçüm araçları ile yapılan testler ve alternatif uygulamalar ile yapılan karşılaştırma sonuçları da bu bölümde ele alınmıştır.

4.1. UYGULAMANIN TASARLANMASI

Bu bölümde CUDA C ile yazılmış bir nesne takip uygulamasına yer verilmiştir. Uygulama tasarımı temel olarak 2 bölümden oluşmaktadır. Birinci kısım uygulamanın CPU tarafında çalışacak olan kısmıdır. Diğeri ise görüntü işleme adımlarını yürütecek olan GPU kısmıdır. Bu temel ayrımın sebebi uygulamanın iki farklı alan arasında gerçekleşen bir iş akışına sahip olmasıdır. GPU teknolojisi hesaplamaları hızlandıracak ve CPU'ya destek olacak bir teknoloji olarak çözüm içerisinde konumlandırılmıştır. Şekil 3.25'te örnek bir CPU-GPU organizasyonu görülmektedir.

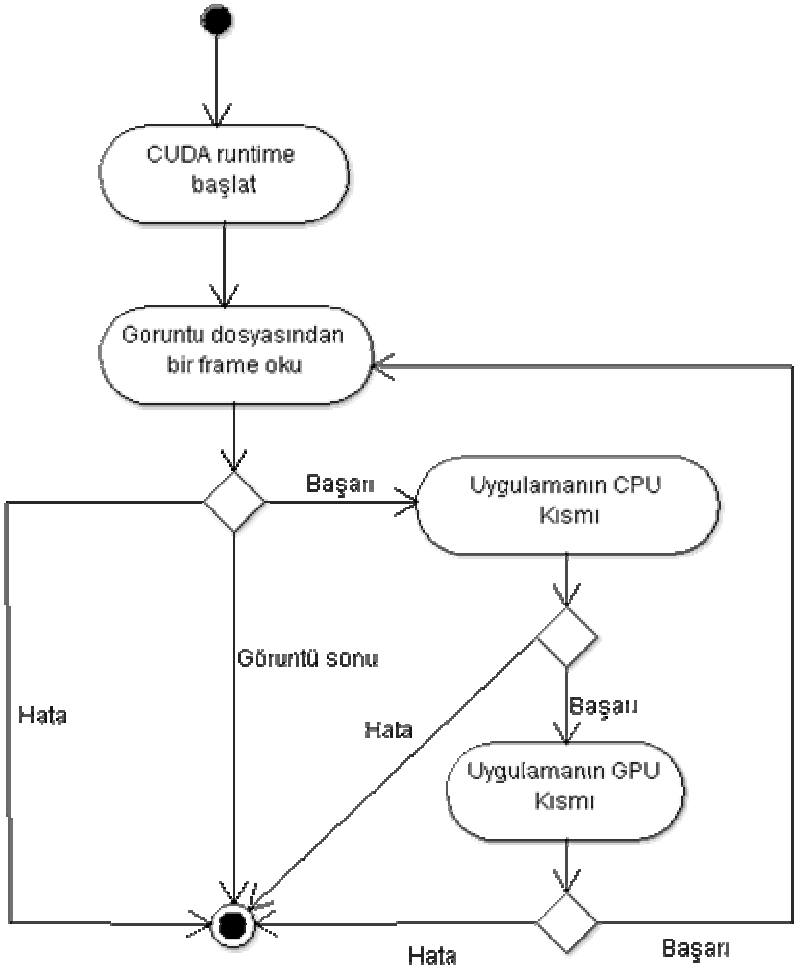


Şekil 3.25: CPU-GPU organizasyon örneği.

CPU(host) kısmında temel girdi-çıktı işlemleri ve pencere yönetimi gerçekleşmektedir. Ayrıca uygulamada kullanılan diğer teknolojilerin ayağa kaldırılması bu bölüm işlevlerinden bir tanesidir.

GPU kısmında görüntü işleme adımları gerçekleştirilmektedir. Bu aşamada OpenGL teknolojisi, görüntünün CUDA tarafından erişilebilir olması ve işlenen görüntünün CPU tarafına aktarılması işlemlerinde çokça kullanılmaktadır. Ama bu teknoloji ile ilgi durum ve sonuçlar CPU başlığı altında ele alınacaktır.

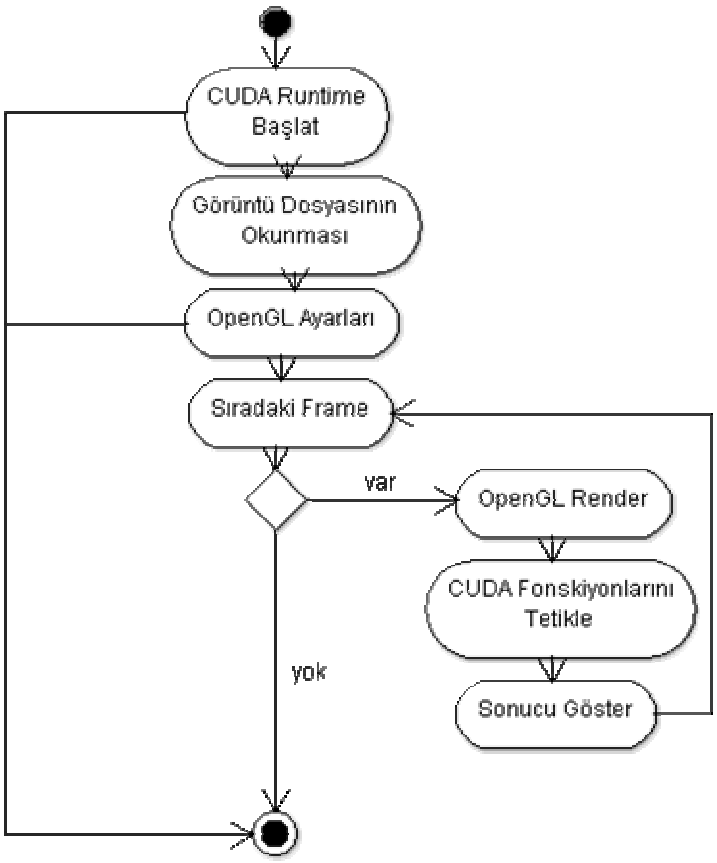
Şekil 3.26'da bu çalışma için hazırlanan uygulamaya ait temel iş akışı görülmektedir.



Şekil 3.26: Nesne takip uygulaması temel iş akışı.

4.1.1. CPU Kodunun Tasarımı

Bu kısımda verinin GPU belleğine aktarılması ve işlenen verinin sonuç olarak kullanıcıya döndürülmesi temel amaçtır. Şekil 3.27'de uygulamanın CPU sorumluluğundaki kısmı için iş akışı görülmektedir.



Şekil 3.27: CPU sorumluğundaki iş akışı.

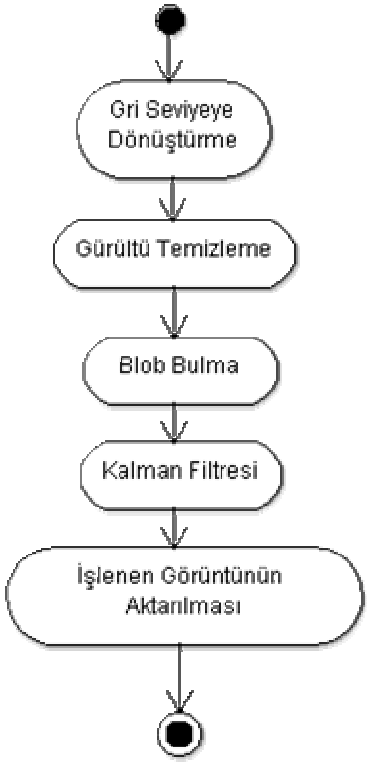
İşlem akışındaki basamaklar aşağıdaki gibidir:

1. CUDA destekli aygıtın seçilmesi ve mevcut host kanalı ile ilişkilendirilmesi.
2. Görüntü dosyasının açılması.
3. CUDA parametre ve değişkenlerinin atamalarının yapılması.
4. OpenGL'in ayağa kaldırılması ve gerekli parametrelerin atamalarının yapılması.
5. OpenGL piksel tampon nesnesi ve frame tampon nesnelerinin okunan görüntünün boyut bilgilerine göre bu nesnelerin şekillendirilmesi.
6. OpenGL kaynaklarının CUDA ortamına kaydedilmesi.

7. Sıradaki ilk frame verisinin okunması.
8. İstenilen CUDA C fonskiyonlarının çağırılması.
9. İşlenilen görüntünün render edilmesi.
10. Eğer görüntü devam ediyorsa 7. adımdan işlemlerin tekrar edilmesi.
11. Eğer görüntü sona erdiyse uygulamadan çıkış işlemin gerçekleştirilmesi.

4.1.2. GPU Kodunun Tasarımı

Bu kısımda alınan görüntü paralel çalışma yöntemiyle işlenir. Temel görüntü işleme tekniklerinin yanı sıra Kalman filtresi ile nesne takibi gibi ileri görüntü işleme tekniklerinin CUDA C diline uyarlanmış versiyonları bu bölümün parçalarını oluşturur. Şekil 3.28'de GPU sorumluğundaki iş akışı görülmektedir.



Şekil 3.28: GPU sorumluluğundaki iş akışı.

İşlemler eşlenmiş OpenGL verisinin adres bilgisinin alınmasıyla başlamaktadır. Daha sonra sırasıyla gri seviyeye dönüştürme, gürültü temizleme, blob(nesne) bulma ve Kalman[42] filtresinin uygulanması adımları gelir. İşlemler sonucu oluşan görüntü OpenGL işbirliği sayesinde OpenGL tarafından okunabilen piksel tampon alanına

yazılır. Daha sonra bu veri CPU tarafında OpenGL tarafında render edilir ve kullanıcı tarafından izlenebilecek hali alır.

Bu kısımdaki bütün görüntü işleme teknikleri CUDA C ile yeniden düzenlenerek işleme konulmuştur. Bölüm 4.2.2'de detaylar yer almaktadır.

4.2. UYGULAMANIN KODLANMASI

Uygulamanın kodlama aşamasında birden fazla kütüphane kullanılmıştır. Bunlardan başlıcaları ünlü görüntü işleme kütüphanesi OpenCV ve 3D grafik kütüphanesi OpenGL'dir. Bu iki kütüphane de CPU iş akışı içerisinde gereken noktalarda kullanılmıştır.

4.2.1. CPU İş Akışının Kodlanması

Uygulamanın giriş noktasından itibaren CPU iş akışı devrededir. Görüntü devam ettiği sürece iş akışının devam ettirilmesi CPU kontrolünde gerçekleşir. Aynı zamanda CUDA kaynaklarının yönetimi, veri için alan tahsisi ve bu alanların silinmesi de CPU kontrolünde gerçekleşir.

4.2.1.1. CUDA Aygıtının Seçilmesi

CUDA aygıtının seçilmesi projenin niteliklerine göre değişlik gösteren bir aşamadır. Örneğin OpenGL işbirliği kullanan bir uygulamada "cudaGLSetGLDevice" fonksiyon çağrısı ile OpenGL işbirliğinin uygulanacağı aygıtın belirtilmesi gerekir.

Bir host üzerinde birden fazla device olabileceği için önce hangi aygıtın kullanılacağının belirlenmesi gerekmektedir. Bu projede hangi cihazın uygulama için kullanılacağı CUDA SDK ile gelen ek kütüphanelerde yer alan "cutGetMaxGflopsDeviceId" fonksiyonu ile belirlenmiştir. Bu fonskiyon sistemdeki CUDA aygıtlarının bir listesini alır ve bunlar içersinden hesaplama hızı en yüksek olanı kullanılmak üzere seçer.

4.2.1.2. Görüntüyü Alma

Farklı görüntü dosya formatlarının işlenebilmesi işlemi zor ve zaman isteyen bir çalışma gerektirir. Görüntünün kodlanması(encode) ve kodlanmış görüntünün kodunun çözümü(decode) bu zorluğun temel kaynağıdır. Farklı formatlarda dosyalar için bu işlemlerde farklılaşır. Bu nedenle görüntü alınması işleminde OpenCV kütüphanesi kullanılmıştır. Bu kütüphanenin ilgili görüntüyü okuyabilmesi için sistemde o dosya formatına ait decode bilgisinin bulunması yeterlidir.

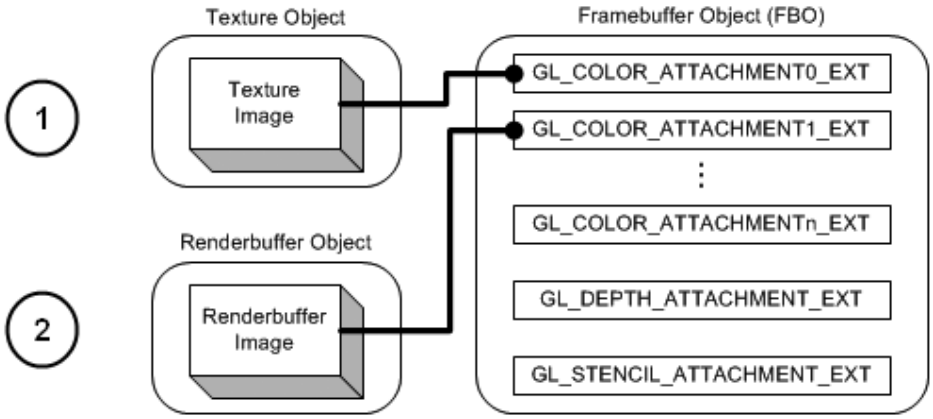
4.2.1.3. OpenGL Nesnelerinin Oluşturulması ve Veri Transferleri

Uygulamamın ilk versiyonunda OpenGL işbirliği bulunmamaktadır. Bölüm 4.3.4'te ilk versiyonda pencere yönetimi ve görüntünün render edilmesi işlemlerinin hangi teknoloji ile gerçekleştirildiği konusuna ve performans kıyaslamalarına yer verilmiştir.

İkinci versiyonda OpenGL işbirliği tekniği kullanılmıştır. Burada öncelikle CUDA'ya kaydedilmesi ve CUDA taradından erişime açık olması gereken PBO(Pixel Buffer Object-Piksel Tampon Nesnesi)'lardan biri alınan biri de işlenen görüntüye işaret etmek üzere 2 adet oluşturulur. Sonrasında dosyadan okunan frame'i saklayacağımız FBO(Frame Buffer Object-Frame Tampon Nesnesi) oluşturulur.

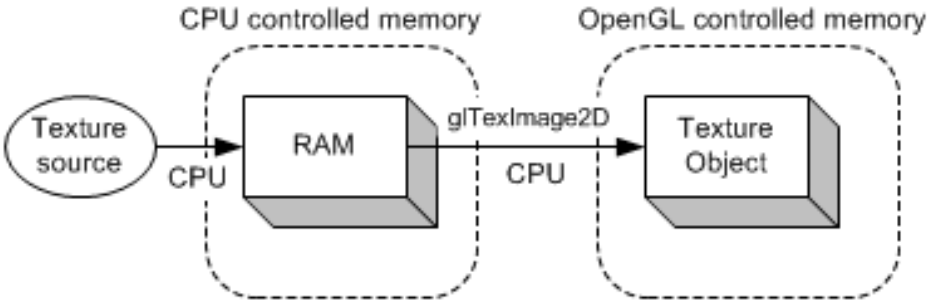
Bir OpenGL eklentisi olan "GL_EXT_framebuffer_object", görüntülenemez FBO'ları oluşturmayı sağlayan bir arayüz sunar. Oluşturulan bu FBO'lar pencere sistemi tarafından sağlanan varsayılan FBO'dan ayrılması için application-created(uyuglama tarafından oluşturulan) FBO'lar olarak isimlendirilirler. Bu tür FBO'ları kullanarak görüntüleme sonucunu yine bu FBO'lara yönlendirmek mümkündür ve bu tamamen OpenGL kontrolünde gerçekleşir[1, 36].

FBO'lara görüntü ekleme işlemi için 2 tür görüntü mevcuttur. Bunlardan ilki ve bu uygulamada kullanılan texture türünde görüntülerdir. Diğeri ise "renderbuffer" görüntülerdir. Şekil 3.29'da 1 numaralı kısım texture görüntülerin, 2 numaralı kısım ise renderbuffer görüntülerin FBO'ya eklenişini göstermektedir.



Şekil 3.29: FBO için eklenebilir görüntü türleri[36].

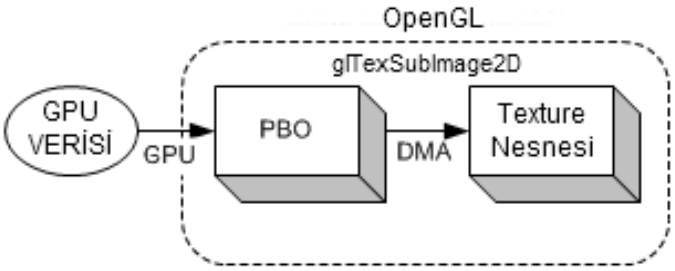
Oluşturulan FBO'nun ilişkilendirilme işleminden sonra OpenCV kütüphanesi ile okunan frame bilgisinin FBO'ya transfer işlemi gerçekleştirilir. Bu işlem "glTexImage2D" fonksiyonu aracılığıyla sağlanır. Şekil 3.30'da verinin aktarımı gösterilmektedir.



Şekil 3.30: Frame verisinin OpenGL ortamına aktarımı[36].

"glReadPixels" metodu ile alınan görüntü verisi CUDA tarafından erişime açık olan PBO'ya kopyalanır ve bu aşamadan sonra GPU iş akışı devreye girer.

Verinin CUDA C ile işlenmesi sonrasında oluşan veri, diğer kayıtlı PBO nesnesine kopyalanır. İlgili PBO'nun ilişkilendirilmesi işleminden sonra OpenGL bu texture verisini direkt olarak render edebilir. Şekil 3.31'de verinin PBO'dan texture nesnesine aktarılışı gösterilmektedir.



Şekil 3.31: GPU çıktısının OpenGL erişimine açılması işlemi.

PBO nesnelerinin hem CUDA hem OpenGL tarafından erişilebilir olmasının dışında bir avantajı daha vardır. PBO ile bir grafik kartına yazma veya bir grafik kartından okuma işlemi DMA(Direct Memory Access-Direkt Bellek Erişimi) tekniğiyle CPU devreden çıkarılarak gerçekleştirilebilir. Verinin transferi OpenGL ve GPU işbirliğiye gerçekleştirilir. Uygulamanın 2. versiyonunda kaydedilen performans artışının en önemli nedenlerinden biri budur.

4.2.2. GPU İş Akışının Kodlanması

GPU iş akışı genel olarak uygulama için gerekli görüntü işleme basamaklarını kapsar. Buradaki temel zorluk, sıralı düzende geliştirilmiş algoritma, filtre ve hesaplamaların CUDA C ile paralel olarak yeniden ele alınmasıdır. Bu da halihazırda kullanılmakta olan basit filtrelemelerin dahi yeniden elden geçirilmesini gerektirir.

Bu çalışma için gri seviyeye dönüştürme, gürültü temizleme, nesne tespit gibi çokça kullanılan filtre ve metotların paralel işleyen versiyonları oluşturulmuştur.

Normal bir görüntüde renk bileşenleri her biri 8 bit alanlarda saklanır. Böylece RGB düzeydeki bir resmin bir pikseli 24 bit değer ile ifade edilir. Eğer RGBA ise her bir piksel 32 bit ile ifade edilir. Bu uygulamada kırmızı, yeşil ve mavi renk değerlerini ayrı ayrı almak yerine, bir tamsayı içinde 8 bitlik değerler halinde saklama yöntemi tercih edilmiştir. Böylece görüntünün her bir pikseli bir tamsayı değer ile temsil edilmiştir. RGB değerlerinin tamsayıya dönüşümü ve tamsayıdan RGB'ye dönüşüm şekil 3.32'de görülmektedir.

```
r = clamp(r, 0.0f, 255.0f);
g = clamp(g, 0.0f, 255.0f);
b = clamp(b, 0.0f, 255.0f);
pixel=(int(b)<<16) | (int(g)<<8) | int(r);

float r = float(pixel&0xff);
float g = float((pixel>>8)&0xff);
float b = float((pixel>>16)&0xff);
```

Şekil 3.32: Tamsayı-RGB dönşümleri.

GPU iş akışındaki filtreleme ve piksel değerlerine bağlı diğer işlemler, piksel verisinin okunması ve yazılması anında şekil 3.32'deki çevrimleri yapmaktadır.

4.2.2.1. Gri ve İkili Seviye Dönüşüm

Bu işlem ile 24 bitlik bir görüntü belli bir formüle dayanarak 8 bitlik durumuna getirilir. Eğer görüntünün gerçek 8 bitlik görüntü olması asıl hedef değilse, yani sadece gri seviyede yapılacak işlemler için gereklilik duyuluyorsa RGB değerlerini kullanarak gri seviye piksel değer hesabı yapılır. Sonrasında elde edilen 0-255 arasındaki bu yeni değer ilgili pikselin R,G ve B elemanlarına atanır. Bu şekilde hem render işleminde gri seviye görünüm sağlanır hem de gri seviye görüntü gerektiren işlemler kolaylıkla uygulanabilir. Gri seviye dönüştürmede aşağıdaki formül kullanılmıştır:

$$\text{Gri Seviye} = R*0.3 + G*0.6 + B*0.1 \quad (4.1)$$

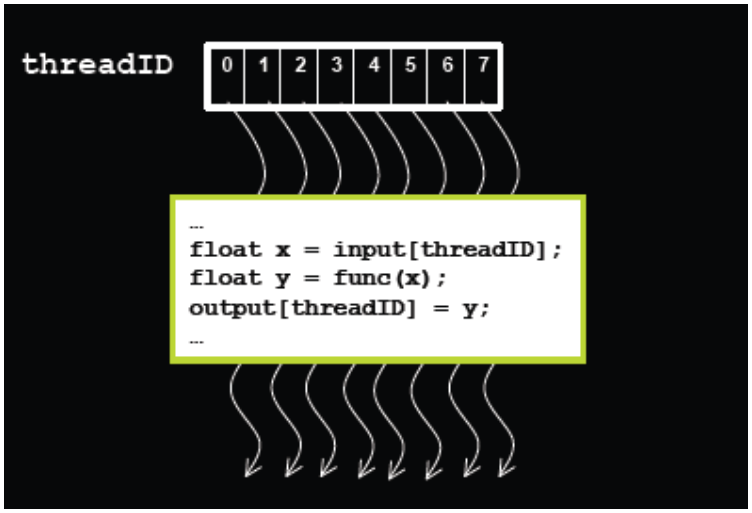
R: Pikselin kırmızı bileşeni.

G: Pikselin yeşil değeri.

B: Pikselin mavi değeri.

Bu basit işlemde önemli bir noktayı daha belirmek gerekir. CUDA kernelleri bir dizi kanal tarafından çalıştırıldığı için sıralı düzendeki gibi bir döngü yardımıyla görüntüye ait bütün pikselleri gezmeye gerek yoktur. Her bir kanal kendisine özel threadID ile çalışır. Bu noktada kernel için oluşturulan kanalların sayısı işlenecek olan görüntünün boyutlarından fazla olsa dahi, kernel içerisinde threadID'nin görüntü boyutlarıyla kıyaslanması şartıyla her bir piksel bir kanal tarafından işlenebilir. Bu durumda resim

üzerinde ilerleyerek dönüştürme yapmaya gerek kalmaz. İşlemimiz bütün pikseller için aynı anda gerçekleşmiş olur. Şekil 3.33'te paralel kanal anlayışı ve şekil 3.34'te uygulamamız da bu anlayışın kullanılışı yer almaktadır.



Şekil 3.33: CUDA paralel kanalları ve işleyiş modeli.

Şekil 3.34'te görüldüğü üzere uygulamamızda minimum düzeyde döngü kullanılmaktadır. Sıralı bir tekniğin CUDA C'ye uyarlanmasındaki temel prensip de budur.

```
__global__ void grayScale(unsigned int * imageIn,unsigned int * imageOut,
                          int width, int height)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int bw = blockDim.x;
    int bh = blockDim.y;
    int x = blockIdx.x*bw + tx;
    int y = blockIdx.y*bh + ty;

    float r = float(imageIn[y*width+x]&0xff);
    float g = float((imageIn[y*width+x]>>8)&0xff);
    float b = float((imageIn[y*width+x]>>16)&0xff);
    imageOut[y*width+x]= rgbToGray(r,g,b);

}
```

Şekil 3.34: Gri seviye dönüştürme

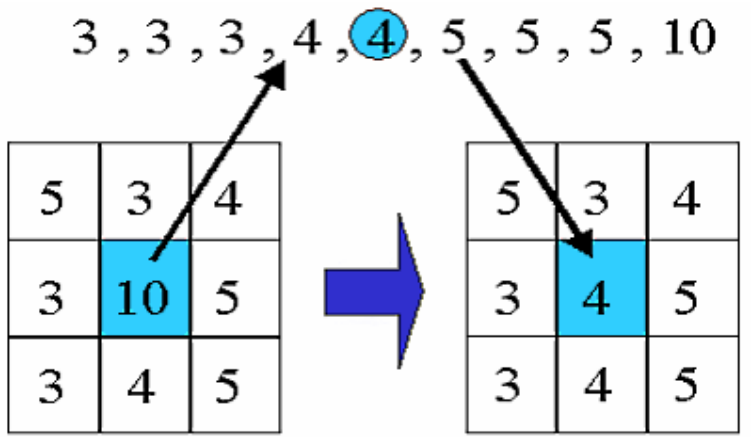
Görüntünün gri seviyeye dönüştürülmesinin ardından görüntünün ikili seviye görüntüye dönüştürülmesi işlemi başlar. Bu işlem de bütün pikseller verilen bir eşik değeri ile kıyaslanır. Eğer piksel değeri eşik değerinden büyükse piksel değerine 255 ataması yapılır, aksi halde piksel değerine 0 ataması yapılır.

4.2.2.2. Gürültü Temizleme

Gürültü temizleme işlemi, görüntü işlemede görüntünün bulanık hale getirilmesi ile mümkündür. Bunun için birçok yöntem bulunmaktadır. Bu uygulamada CUDA paralel programlama tekniğine uygun olması nedeniyle median ve average yöntemleri denenmiş ve kullanılmıştır.

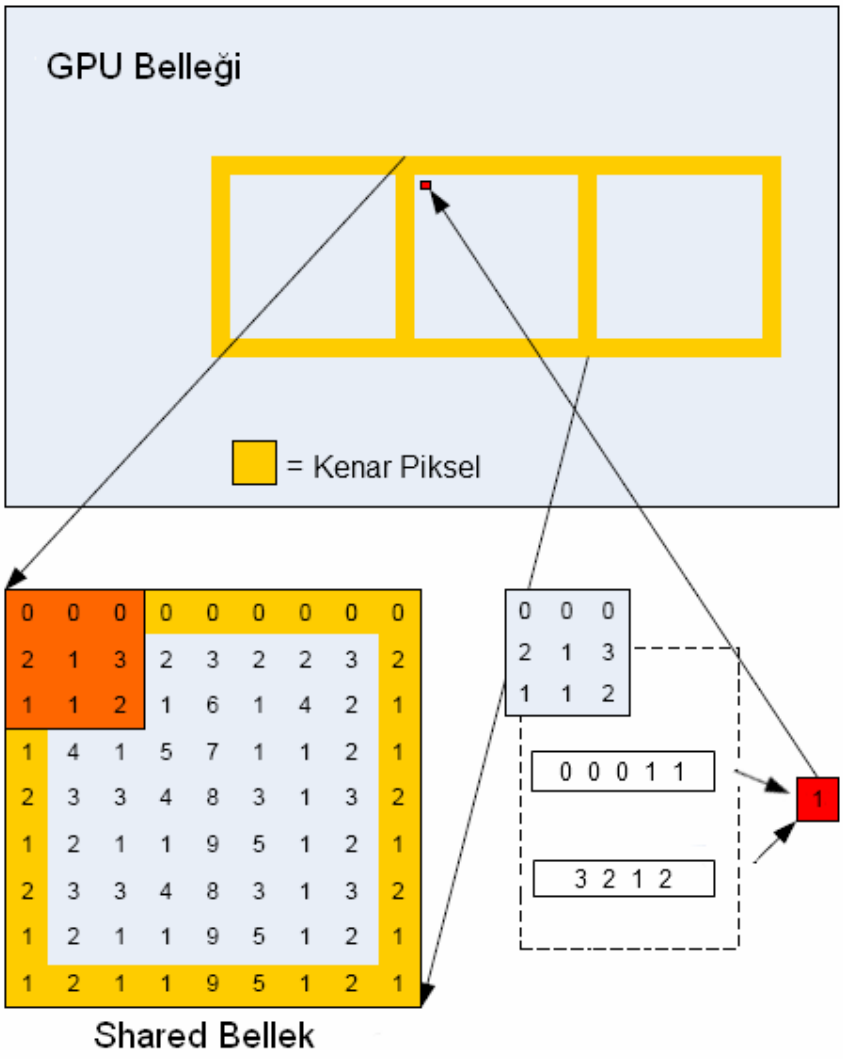
Uygulamanın ilk versiyonunda median filtresi shared(ortak) bellek olmadan kullanılmış ancak istenilen performans değerlerine ulaşılamadığı için yöntem, shared bellek kulanacak şekilde değiştirilmiştir. Bölüm 4.3.3'te gürültü temizleme için kullanılan metotlar ve performans değerlendirmelerine yer verilmiştir.

Median metodunda ilgili piksel merkezde olmak üzere etrafında istenilen büyüklükte bir kare boyunca pikseller sıralanır ve ortada olan değer ilgili piksele atanır. Böylece pikseller arası geçişler yumuşatılır ve gözde bulanıklık etkisi yaratılır. Şekil 3.35'te standart median filtresi ve çalışma şekli görülmektedir.



Şekil 3.35: Median filtresinin çalışma şekli.

Tez için yapılan uygulamanın ikinci versiyonunda önce ilgili piksel değerleri shared belleğe alınmıştır. Sonra bu elemanların sadece yarısı sıralanmış ve en küçük değer saptanmıştır. Daha sonra kalan elemanlar bu en küçük değerle kıyaslanmış ve bu değerler içinde ortaki değer saptanmıştır. Şekil 3.36'da shared bellek ile median filtrelemenin bir görseli yer almaktadır.



Şekil 3.36: Median filtresi CUDA uyarlaması.

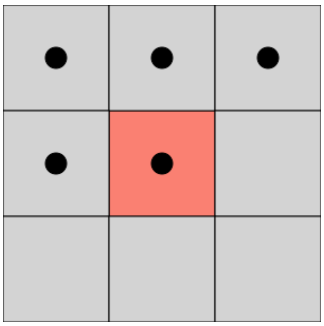
Şekil 3.36'da görüldüğü gibi kenar pikseller üzerinde herhangi bir işlem yapılmadan direkt shared bellekte ilgili alana kopyalanırlar. Sıralama yapılacak piksellerin shared belleğe alınması işlemler sırasında sürekli global hafızadan veri okuma ile gelecek olan

performans sorunlarından kurtarır. Böylece sıralama ve kıyaslama işlemlerinde kerneller veriyi shared bellekten alarak işlem gerçekleştirirler.

4.2.2.3. Nesne Tespiti

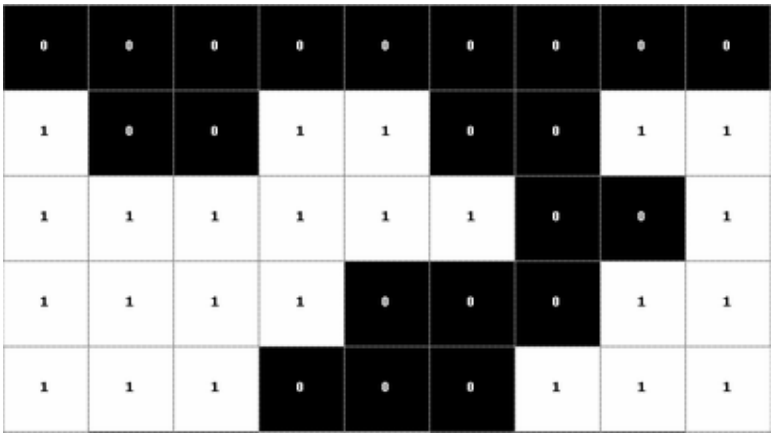
CPU üzererinde yazılan uygulamalarda kötü performansları nedeniyle eleştirilen bazı algoritmaların çalışma şeklinin paralel çalışma yöntemine uygun olması, çok az kullanılan tekniklerin dahi uygulamalarda yer bulmalarına neden olmuştur. Nesne tespit algoritmaları genel olarak görüntü üzerinde her bir pikseli değerlendikleri için sıralı programlama teknikleri ile uygulandıklarında düşük performans değerleri sunarlar. Bu çalışmada nesne konum tespiti için blob bulma yönteminden faydalanılmıştır.

Standart iki adımlı çözümün birinci adımında şekil 3.37'de gösterilen komşu değerler ile merkezdeki pikselin değeri karşılaştırılır. İşaretli olan komşulardan değeri 255 olan ve etiket değeri en küçük olan pikselin etiket değeri merkez piksele atanır.

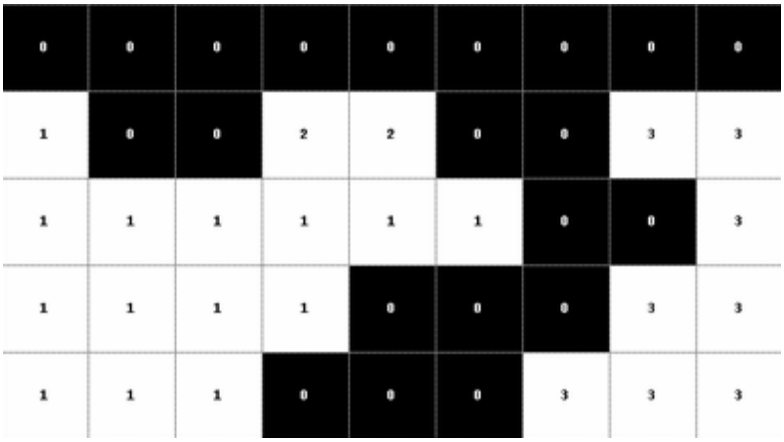


Şekil 3.37: Piksel etiketlemede sekizli komşu yaklaşımı.

Birinci adım sonrasında şekil 3.38'deki gibi bir görüntü etiketlenerek şekil 3.39'daki duruma gelir.



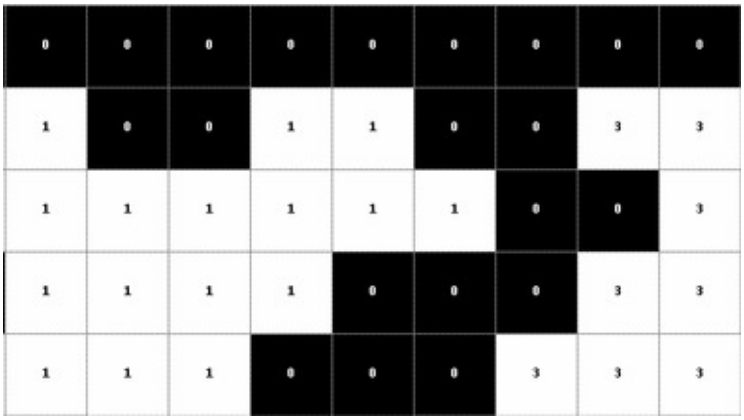
Şekil 3.38: Etiketleme öncesi pikseller ve değerleri.



Şekil 3.39: Etiketleme birinci adım sonu.

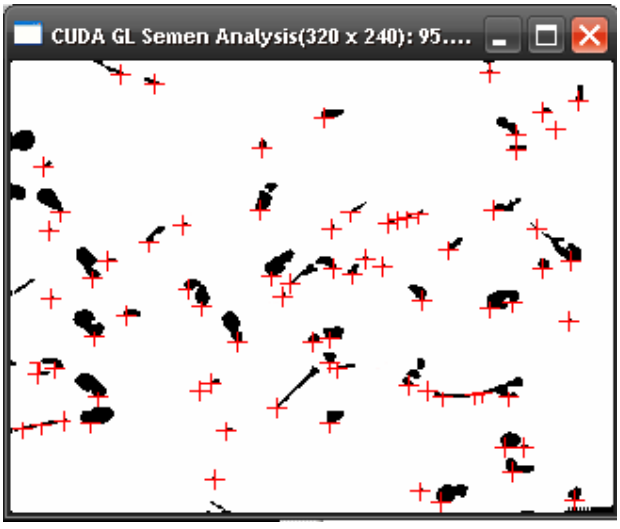
Bu çalışmada ise eğer bir pikselin kendinden önceki dört komşusunun değerleri 255 değilse veya etiket değerleri merkez pikselden küçükse sıralı olarak düzenlenen etiket numarası yerine pikselin görüntüdeki piksel dizisi içindeki sırası etiket değeri olarak kullanılmıştır. Bu aşama CUDA C ile düzenlendiği için her bir kanal bir piksel ve onun komşuları arasında işlemler gerçekleştirir. Yani birinci adım hiçbir döngü kullanılmadan paralel olarak bütün resim için gerçekleştirilir.

Standart çözümün ikinci adımında ise etiketlenen pikseller için tekrarlamalı bir yaklaşım sergilenir. Birbirine komşu olan etiket değerleri içinden en küçük olan etiket değeri ilgili piksele atanır. Şekil 3.40'ta bu işlemin sonucu görülmektedir.



Şekil 3.40: Etiketleme ikinci adım sonu.

Çalışmamızda standart çözümün ikinci adımındaki tekrarlamalı yaklaşım değiştirilmiştir. Her bir piksel için etiket değeri ile pikselin piksel dizisi içerisindeki sıra numarası aynı olana kadar ilerlenir. Bu tekrarlamanın son pikselinin etiket değeri ilgili piksele atanır. Sıralı çözümlerde böyle bir yaklaşım performansı dikkat çekici bir şekilde düşürür. Ancak CUDA C uyarlamasında bu yaklaşım bütün pikseller için döngü yerine bütün bir görüntü için tek bir döngü gibi düşünülebilir. Çünkü bütün kanallar paralel olarak pikselleri işlemektedir.



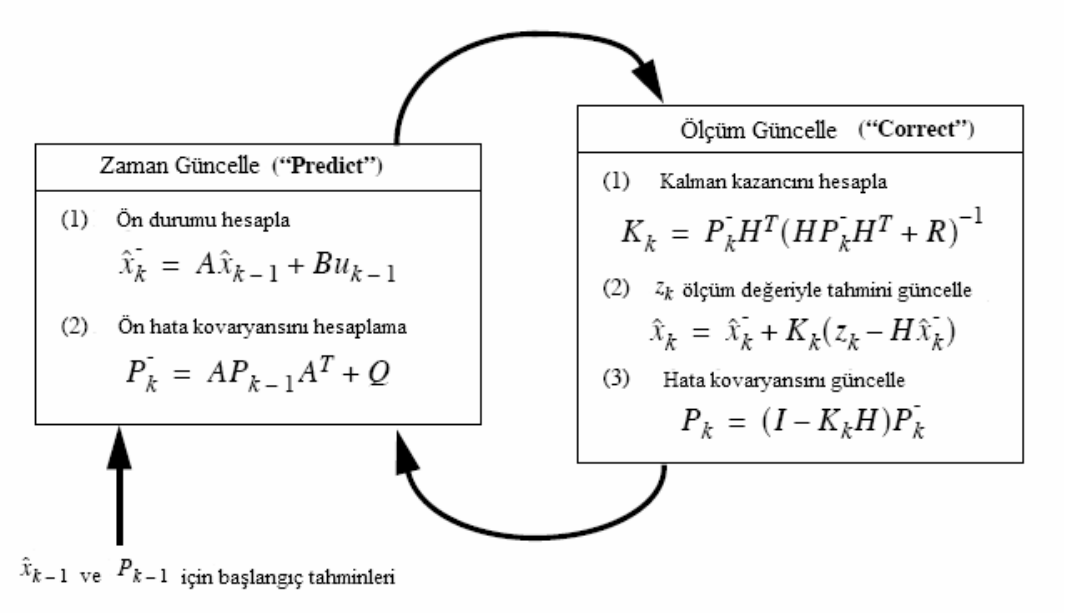
Şekil 3.41: Çoklu nesne tespiti.

Uygulanan adımlar sonucunda piksel sıra değeri ile etiket değerleri aynı olan pikseller görüntü üzerinde takip edilmek üzere bir sonraki aşamaya iletilirler. Kullanılan çözüm görüntü üzerinde bütün nesnelerin etiketlenmesi sağlanmıştır. Ancak bu çözüm CUDA C ile atomik işlemlerin performans problemlerini gözlemlemek adına ortaya koyulmuş bir çözümdür. Temel olarak çok iyi bir filtrelemenin ardından nesne merkezinin önemli olmadığı tekil nesne tespit uygulamalarında bulunan en küçük 0 dan farklı piksel değeri nesnenin bir parçasıdır. Bu nedenle yapılan uygulamada detaylar düşünülmeden daha yüksek performanslı bir çözüm uygulanabilir. Şekil 3.41'de uygulamamızda şu ana kadar anlatılan metotlarla nesne tespitinin yapıldığı örnek bir an görülmektedir.

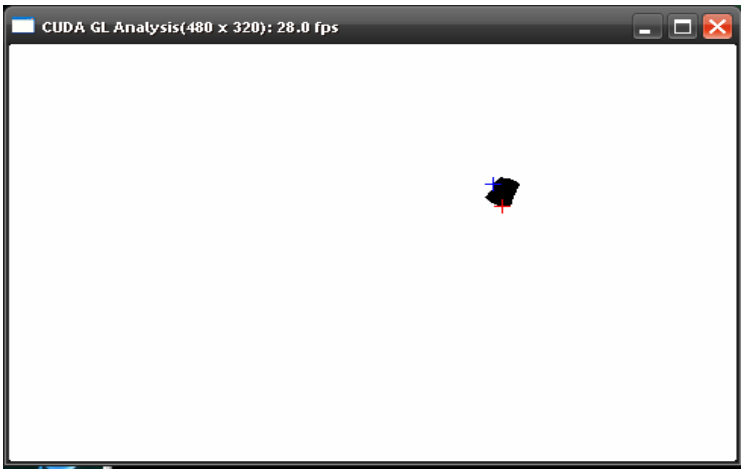
4.2.2.4. Nesne Takibi

Nesne takibi son 20 yıldır hakkında görüntü işleme alanında çok fazla araştırmanın yapıldığı bir konudur. Uygulanabilecek bir çok yöntem vardır. Bu uygulamada Kalman[42] metodu CUDA C diline uyarlanarak kullanılmıştır. Bu yöntem basit olarak herhangi bir "k" anında eldeki ölçüm verilerini kullanarak durum tahmininde bulunmayı sağlar. Şekil 3.42'de Kalman metodunun çalışma şeklini ifade eden döngü yer almaktadır.

Bu hesaplamada "H", "A", "B", "Q" matrisleri sabit katsayı matrisleri olarak kullanılmıştır. Gerçekte bu matrisler herbir zaman diliminde değişsen değerlerdir. "u" değeri kontrol girdisi olarak kullanılmıştır. "P" hata kovaryansını, "K" ise Kalman kazancını ifade etmektedir. "x" durum tahmini, "z" ise o durum için yapılan ölçüm değerini ifade eder. Uygulamamızda bütün bu matris hesaplamaları CUDA C ile uyarlanmış ve shared bellek kullanan metotlar ile gerçekleştirilmiştir.



Şekil 3.42: Kalman döngüsü[42].



Şekil 3.43: Kalman filtreleme sonucu.

Çoklu nesne takibinde önemli bir unsur olan nesne eşleştirme yöntemleri tekil nesnelerin takibinde gerekmez. Bu nedenle görüntü üzerinde herhangi bir t+1 anında tespit edilen nesne, t anındaki nesneyle aynı kabul edilir ve Kalman döngüsüne devam edilir. Şekil 3.43'te Kalman filtresinin uygulanması ile elde edilen tahmin değeri mavi, ölçüm değeri ise kırmızı artı olarak görülmektedir.

4.3. PERFORMANS TESTLERİ VE İYİLEŞTİRMELER

Uygulama için iki versiyon hazırlanmıştır. İlk versiyondaki performans test sonuçlarına bağlı olarak bazı bölümlerde iyileştirmeler yapılmıştır. Performans testleri Visual Profiler test aracı ile hazırlanmıştır. Bu bölümde yapılan testlere ve performans iyileştirmelerine dair detaylar yer almaktadır.

4.3.1. Test Ortamı

Uygulamanın testlerinin yapıldığı platformun özellikleri aşağıdaki gibidir:

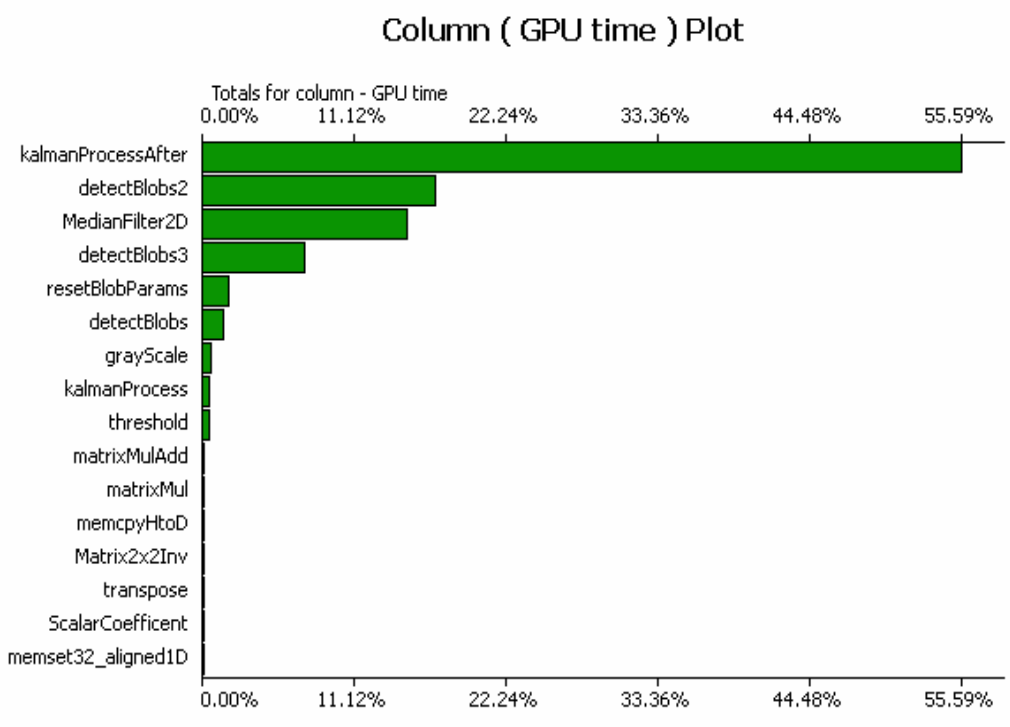
- AMD Sempron LE-1200 2.11 Ghz işlemci.
- 1 GB 800 Mhz DDR2 bellek.
- NVIDIA GeForce 9500GT grafik kartı.
- Windows XP Service Pack 3 işletim sistemi.
- CUDA SDK Versiyon 3.2
- Visual Studio 2008

Test sonuçları özellikle grafik kartının değişimi konusunda son derece hassastır. Bu nedenle diğer etkenler sabit kalmak koşuluyla sadece grafik kartının değiştirilmesi test sonuçlarını büyük miktarda değiştirebilir.

4.3.2. Visual Profiler Testleri

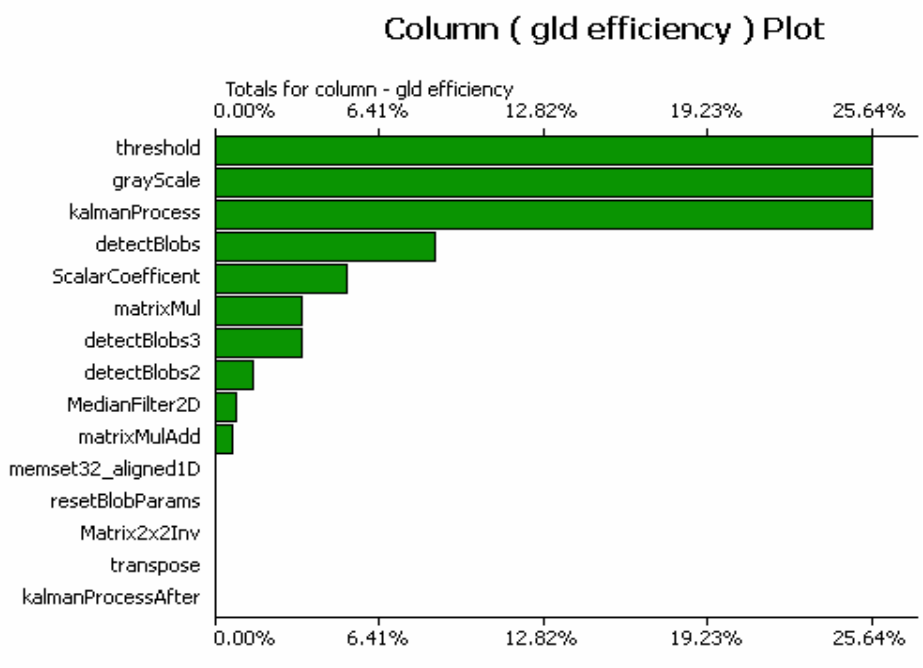
Visual profiler aracı CUDA araç kiti ile birlikte gelmektedir. Bu araç ile veri transfer, çalışma süreleri analizleri ve karşılaştırmalı analizler yapmak mümkündür. Bu çalışma süresince yapılan uygulamalar Visual Profiler ile test edilmiştir.

Uygulamanın ikinci aşamasıda Kalman filtreleme yöntemi çalışmaya eklenmiş ve CUDA ile video üzerinde tekil nesne takibi çalışmalarına geçilmiştir. Kalman yönteminde kernellere dayalı bir çözüm yerine matris çarpımlarında CUDA hesaplama teneklerinden çokca faydalanan bir yapı tercih edilmiştir. Şekil 3.44'te hazırlanan uygulamaya ait GPU zaman analizleri yer almaktadır.



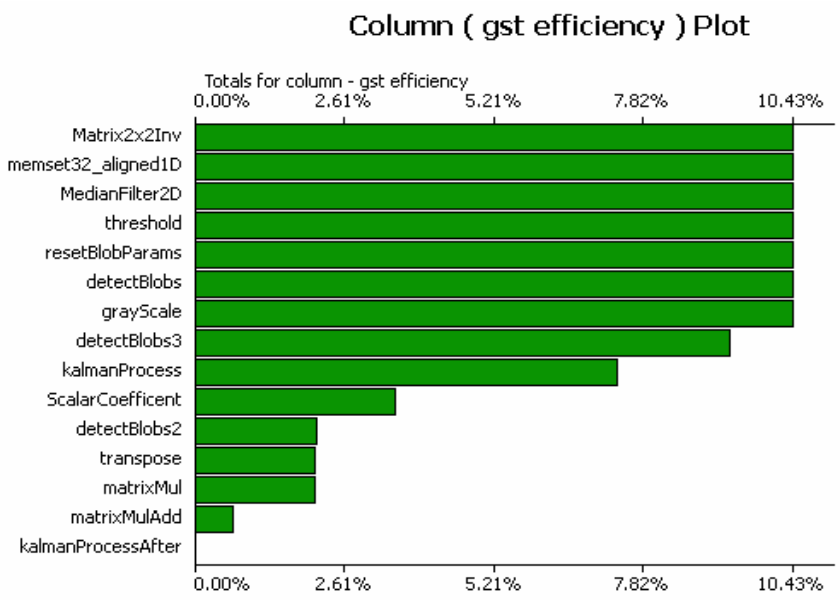
Şekil 3.44: GPU zaman analizleri.

Yapılan çalışmada Kalman filtreleme işlemlerinin GPU zamanın %54.13'ünü kullandığı görülmektedir. Daha sonra iterative işlemler gerçekleştiren blob bulma işleminin ikinci basamak adımlarını gerçekleştiren "detectBlobs2" fonksiyonunun GPU zamanın yaklaşık %16'sını kullandığı görülmektedir. Sonra sırasıyla median filtreleme ve blob renklendirme işlemlerinin GPU zamanını en çok kullanan fonksiyonlar olduğu görülmektedir. Şekil 3.45'te metotlara göre global bellekten okuma verimlilik değerleri görülmektedir.



Şekil 3.45: Global bellek okuma verimlilik değerleri.

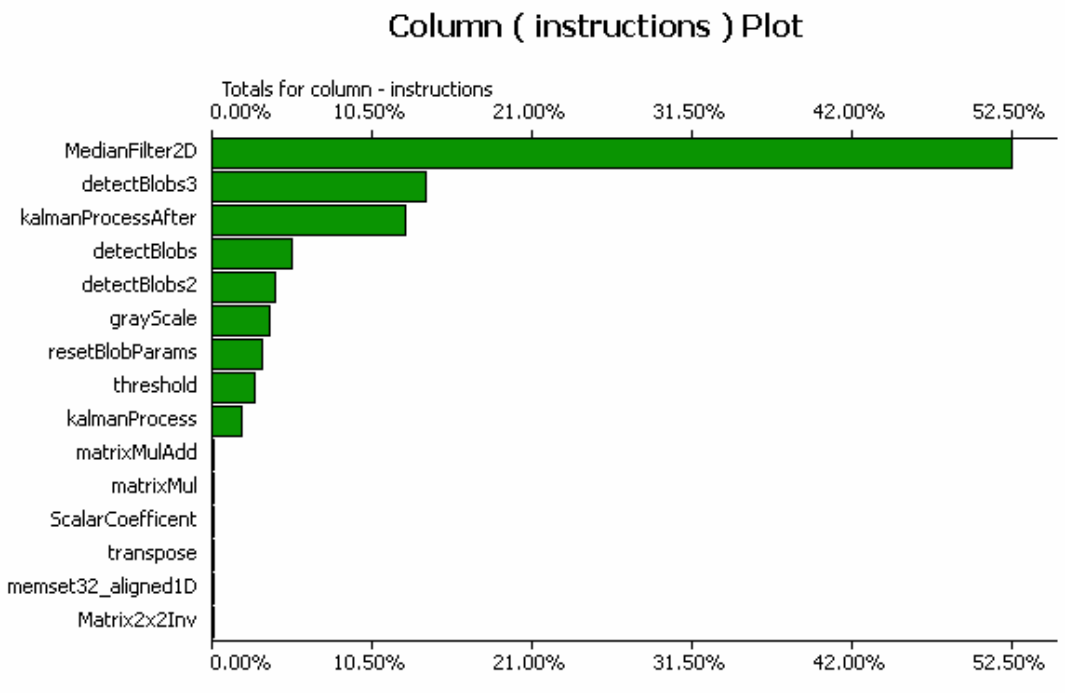
Yukarıdadaki grafikte kernel içerisinde global bellekten yapılan okuma sayısı fazla olan fonksiyonların verimliliklerinin diğer fonksiyonlara göre daha az olduğu görülmektedir. Şekil 3.46'da metotlara göre global belleğe yazma verimlilikleri görülmektedir.



Şekil 3.46: Global belleğe yazma verimlilik değerleri.

Şekil 3.46'deki grafik, kernel işlemleri esnasında global belleğe yazma girişimi diğerlerine göre daha az olan metotların global belleğe yazma verimliliklerinin diğer fonksiyonlara göre daha yüksek olduğu görülmektedir.

Son olarak 3.47'de metotlara göre çalıştırılan komut sayısı istatistiği görülmektedir.



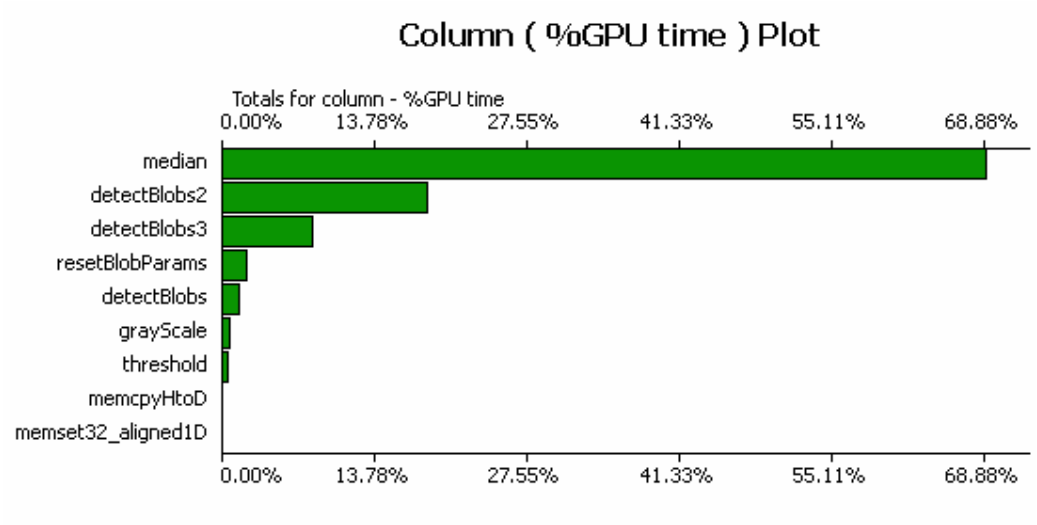
Şekil 3.47: Metot başına çalıştırılan komut sayıları.

İstatistikler sonucu metotların GPU üzerinde çalıştırılma zamanlarının birden fazla faktöre bağlı olduğu görülmektedir. Metotlar için GPU zamanının ağırlıklı olarak global belleğe yazma verimlilik değerinden etkilendiği tespit edilmiştir. Sırasıyla global bellekten okuma verimliliği ve komut sayısı faktörleri GPU zamanını belirlemede diğer önemli etkenler olarak gözlenmiştir.

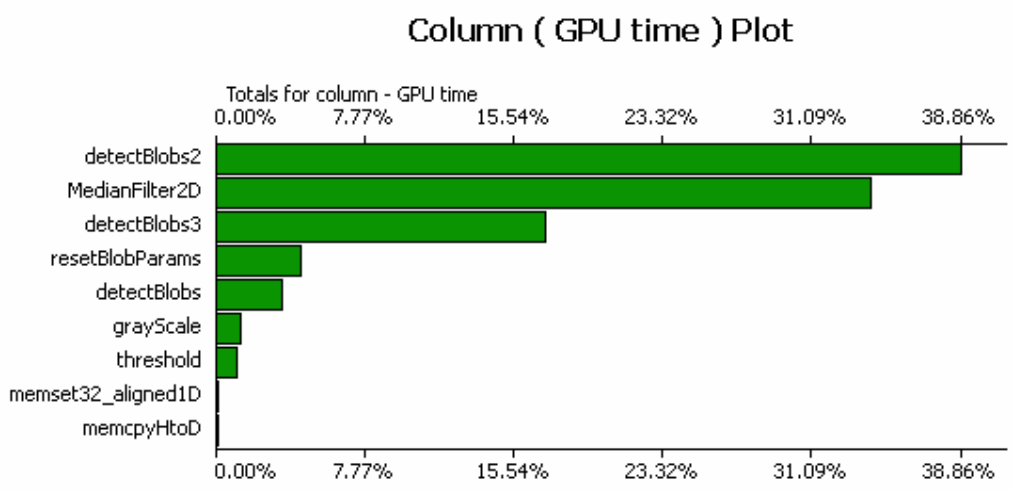
Uygulamanın birinci adımı sonunda yukarıdaki testler ve bazı ek bellek veri transfer testleri yapıldıktan sonra metotlarda iyileştirmenin gerekliliği görülmüştür. Öncelikli olarak kullanılan filtreler ve ek kütüphaneler ele alınmıştır.

4.3.3. Filtrelerde İyileştirme

Uygulamanın birinci versiyonunun sonunda yapılan testler sonucu global bellek erişimlerinin filtrelerin performans değerlerini önemli derece etkilediği sonucuna varılmıştır. Bu nedenle median filtresi gibi bir pikselin değerini hesaplamak birçok kez global bellekten okuma yapan metotlarda shared bellek kullamına geçilmiş ve global bellek erişim sayısı azaltılmıştır. Şekil 3.48'de median filtresinin shared bellek kullanıldığı durum ve şekil 3.49'da shared belleğin kullanılmadığı durum için GPU zaman kullanım istatistikleri yer almaktadır.



Şekil 3.48: Uygulamanın ilk versiyonu için GPU zaman kullanımı.

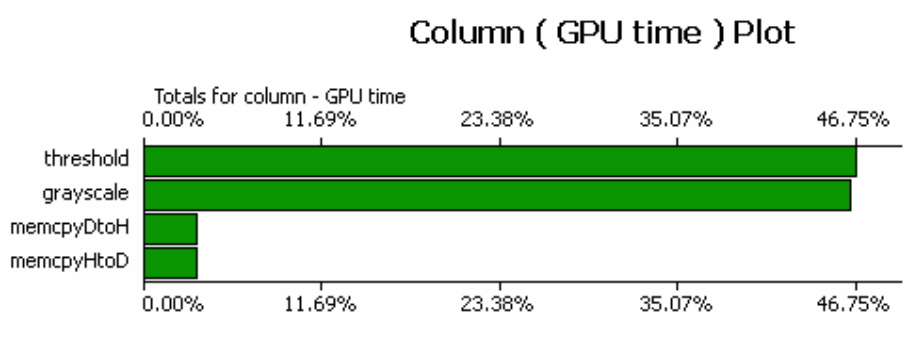


Şekil 3.49: Uygulamanın ikinci versiyonu için GPU zaman kullanımı.

Görüldüğü üzere shared bellek kullanımı ile median filtresinin performansı %100 artmıştır.

4.3.4. Veri Transfer Performansında İyileştirme

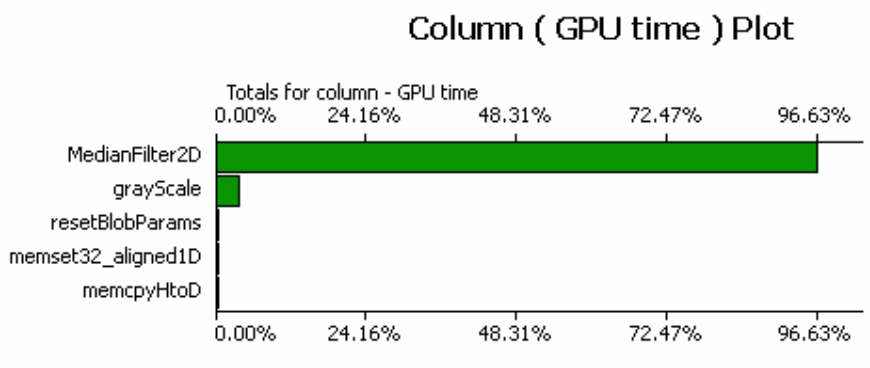
Uygulamanın ilk versiyonunda OpenGL işbirliği yerine OpenCV kütüphanesinden faydalanılmıştır. Veri dosyadan okunduktan sonra GPU belleğinde açılan alana kopyalanmıştır. Daha sonra işlenen veri GPU belleğinden okunarak OpenCV pencere yöneticisinde gösterilmiştir. Tüm bu veri transfer işlemleri uygulamanın performansı önemli derecede etkilediği tespit edilmiştir. Şekil 3.50'de uygulamanın birinci versiyonunda OpenCV kullanımı nedeniyle veri transferleri için harcanan GPU zamanı görülmektedir.



Şekil 3.50: OpenGL işbirliği olmadan veri transferleri için harcanan GPU zamanı.

Yukarıdaki şekilde "memcpyDtoH" ifadesi GPU'dan CPU'ya veri aktarım için harcanan zamanı ifade eder. "memcpyHtoD" ifadesi ise CPU'dan GPU'ya veri aktarımı için harcanan süreyi ifade eder. OpenCV'nin işlenen görüntüyü pencere yöneticisine yükleyebilmesi için önce veriyi CPU ortamına kopyalaması gerekmektedir.

OpenGL işbirliği tekniğinde işlenen görüntünün pencere yöneticisine yüklenebilmesi için verinin GPU'dan CPU'ya kopyalanması gerekmez. Bu nedenle OpenGL işbirliğinden faydalanan uygulamamızın ikinci versiyonu veri transferleri için %50 daha az GPU zamanı kullanır. Şekil 3.51'de OpenGL işbirliği ile veri transfer performansı iyileştirilmiş uygulamamız için GPU kullanım zamanları görülmektedir.



Şekil 3.51: OpenGL işbirliği ile veri transferleri için harcanan GPU zamanı.

5. TARTIŞMA, SONUÇ VE GELECEKTEKİ ÇALIŞMALAR

GPU'nun genel amaçlı hesaplamalar için kullanılması yeni bir fikir olmamasına rağmen, bu fikri destekleyecek, yazılım geliştirici ve araştırmacıların kolayca GPU destekli uygulamalar geliştirebilecekleri ortam ve araçların geç sunulması uzunca süredir bilinen bir teknolojinin detaylarını kavramak noktasında geç kalınmasına neden olmuştur.

Bu araştırmada NVIDIA firmasının CUDA destekli platformlarında GPGPU desteği ve paralel programlama ile yüksek performanslı işlemler gerçekleştirme imkanı sağlayan CUDA C dili incelenmiştir. Bu dil ile örnek bir tekil nesne takibi uygulaması gerçekleştirilmiştir.

Çalışma boyunca üzerinde durulan bazı temel noktalar aşağıda sıralanmıştır:

- Kerneller için global bellek kullanımı
    - Global bellek okuma verimlik analizleri
    - Global bellek yazma verimlilik analizleri
- Shared bellek kullanımı ve performans analizleri
- Veri transferleri ve performans analizleri
- Görüntü işleme filtreleri için GPU zaman kullanımı analiz ve karşılaştırmaları
- Kerneller için komut sayısı analizleri
- GPU harcanan zamanı ve komut sayısı ilişkisi
- GPU harcanan zamanı ve global bellek erişimi ilişkisi

Yapılan uygulamanın iki farklı versiyonu için yukarıdaki başlıklar altında analizler yapılmış ve sonuçlar sunulmuştur. Sonuçlara bağlı olarak bazı noktalarda yöntem değişiklikleri veya iyileştirmelere başvurulmuştur.

Kullanılan çeşitli araç ve yöntemler ile CUDA C dilinin ve CUDA platformununun görüntü işlemede kullanımı konusunda aşağıdaki sonuçlara ulaşılmıştır:

1. CUDA C ile uygulama geliştirilebilecek tam anlamda CUDA patformuna entegre olmuş bir geliştirme ortamının olmaması uygulamaların geliştirilme sürelerini uzatmakta, hata ayıklama işlemlerini ve derleme seçeneklerinin tam anlamıyla kullanımını zorlaştırmaktadır.
2. NVIDIA firması görüntü işleme konusunda çalışan araştırmacı ve yazılımcıları CUDA teknolojisini kullanmak konusunda desteklemektedir. Ancak CUDA C dili içerisinde görüntünün pencere yöneticisinde gösterilmesini sağlayacak, bu işlem için gereken veri transferlerini kullanıcıdan soyutlayacak, görüntü dosyalarını okuma ve düzenleme kolaylığı sağlayacak ek kütüphaneler sunmamaktadır. Bu da yazılım geliştiricilerin genelde GPU entegrasyonu olmayan diğer kütüphanelere yönelmelerine neden olmaktadır.
3. Yazılan kernel fonksiyonlarının en etkili şekilde çalışmalarını sağlayacak grid, blok ve kanal sayısı konfigürasyonunu otomatik olarak gerçekleştirecek bir çözümün olmaması performans açısından sorun teşkil etmektedir.
4. İşlenilen görüntünün yükseklik ve genişlik değerleri grid, blok ve kanal değerleri ile doğrudan ilişkilidir. Grid, blok ve kanal değerleri için yapılan yanlış atamalar erişim sınırlaması olan bellek bölgelerine erişim gibi ölümcül(fatal) hatalarla karşılaşılmasına neden olabilir. Bazı durumlarda yanlış atamalara rağmen uygulamanın çalıştığı ancak elde edilen sonuçların yanlış olduğu gözlemlenmiştir.
5. Görüntü boyutları ile grid, blok ve kanal sayılarının tam uyumlu olmaması durumunda grid, blok ve kanal değerleri için görüntüyü kapsayacak şekilde yapılan atamalarda bazı kanalların koordinatlarının görüntü alanına denk gelmediği görülmüştür. Bu durumlarda kanalların işleme devam etmelerini engelleyecek kontrollere ihtiyaç duyulmuştur.
6. Kerneller arasında genelde kontrol amaçlı olarak kullanılan global bellek değişkenlerinden okuma veya yazma işlemlerinin sayısının fazlalığı ilgili kernel için GPU'da harcanan zamanın önemli derecede artmasına neden olmaktadır.

7. Kernellerin sahip oldukları komut sayısının GPU'da harcanan zaman üzerindeki etkisi, kernelin global hafıza erişimlerinin etkisinden daha azdır.
8. Shared bellek kullanımı, hesaplamalar esnasında bellek erişimini çokca kullanan fonksiyonlarda önemli derecede performans artışı sağlamıştır.
9. Yazılan uygulamalarda bilinen tekniklerin değiştirilmesine çokca ihtiyaç duyulmaktadır. Alternatif çözümler ile aynı fonksiyonellik sağlanmak zorunda kalınmıştır. Örneğin blob bulma işleminde blob alanlarının hesaplanması ve belli bir eşik değere göre filtrelenmesi bilinen bir uygulamadır. CUDA C ile bu işlemi gerçekleştirebilmek için hesaplama kapasitesi 1.1 ve üzeri olan aygıtlarla gelen atomik işlem desteğinden faydalanılır. Bu yapılar ilgili bellek alanının senkronize çalışmasını sağlar. Yani aynı anda sadece bir erişime izin verirler. Ancak yapılan testler sonucu, görüntüde bulunan nesnelerin alanlarının piksel türünden hesaplanması işleminde atomik işlemlerin kullanılmasının %70-%75 performans gerilemesine neden olduğu görülmüştür. Bu nedenle atomik işlemler uygulamada tercih edilmemiştir. Alternatif çözüm olarak gürültü filtrelerinin daha etkili çalışmaları sağlanmıştır.
10. Global bellek değişkenlerinin kernellere girdi olarak gönderilmediği bazı durumlarda, grafik kartının ve ona bağlı olarak CPU'nun çağrılara yanıt vermediği ve çoğu kez ölümcül hatalarla sistemin yeniden başlatıldığı gözlenmiştir. Bu nedenle uygulamada kullanılan bütün kernel fonksiyonları, erişimleri olan global değişkenleri girdi olarak alacak şekilde yeniden düzenlenmiştir.
11. Yapılan denemelerde CPU ortamında performansları kötü olan ve bu nedenle pek tercih edilmeyen algoritmaların, veri paralelliği ilkesine uyumlu olmaları durumunda GPU üzerinde CPU eşleneklerine göre daha verimli oldukları saptanmıştır.
12. CPU-GPU ve GPU-CPU veri transfer maliyetleri yüksek olduğu için verinin uygulama sonunda CPU'ya aktarıldığı çözümler daha verimlidir. Ancak bu seçimin gerçekleşebilmesi için grafik kartının belleğinin yeterli miktarda ve mümkünse büyük olması gereklidir.
13. Yapılan uygulama görüntü tabanlıysa ve yüksek performans hedefliyorsa OpenGL kütüphanesinden faydalanılmalıdır. CUDA, OpenGL işbirliği teknolojisi ile GPU'da işlenen verinin kullanıcıya gösterilmesi aşamasında veri

transfer işlemleri gerekmez. Böylece CPU-GPU ve GPU-CPU toplam veri transfer zamanı %50 oranında azaltılabilir. Yapılan performans analizlerinde bu durumla ilgili istatistiklere yer verilmiştir.

Yapılan uygulamalar, testler ve araştırmalar sonucu elde edilen sonuçlar GPU hesaplama yeteneklerinin özellikle veri paralelliği gerektiren alanlarda çok etkili olabileceğini göstermiştir. NVIDIA firması da bu talebe karşılık mimarisini geliştirmeye devam etmektedir. Fermi kod isimli mimarisi ile paralel programlamayı daha ileriye taşımayı amaçlamaktadır. Donanım gücünü de bu ölçüde artırmaya devam etmektedirler. Örneğin GeForce 560 GTX Ti grafik kartında 384 adet çekirdek bulunmaktadır. Bu anlamda CUDA mimarisi üzerinde çalışan birçok yeni projenin sunulması mümkündür.

Tez için geliştirdiğimiz uygulamalarda bir adet grafik kartı kullanılmıştır. Ancak NVIDIA SLI teknolojisi ile birden fazla grafik kartını aynı uygulama için kullanmak mümkündür. Bu tarz bir dizayn ile uygulama performansı önemli derecede artacaktır.

Ayrıca NVIDIA firması çok düşük fiyatlarla süper bilgisayar ortamını yaratmak adına Tesla ürününü kullanıcılara sunmaktadır. Bu ürün üzerinde yazılacak uygulamalarla supercomputing(süper hesaplama) alanında etkili çalışmalar yapmak mümkündür.

Gelecekte GPU dünyasındaki gelişmelerin işlemci teknolojisine yön vereceği uzmanlar tarafından tahmin edilmektedir. Bu nedenle bu projede sorun olarak iletilen durumların kendisini sürekli yenilen CUDA mimarisi ile yakın zamanda çözüleceği, programlama kapasitesinin ve yaygınlığının artacağı görülmektedir.

KAYNAKLAR

1. Open Graphics Library [online], http://www.opengl.org, [Ziyaret Tarihi: 10.02.2011].

2. Open Computing Language [online], http://www.khronos.org/opencl/, [Ziyaret Tarihi: 20.11.2010].

3. NVIDIA [online], http://www.nvidia.com/, [Ziyaret Tarihi: 5.10.2010].

4. NVIDIA GeForce 8 Series [online], http://www.nvidia.com/page/geforce8.html, [Ziyaret Tarihi: 17.10.2010].

5. NVIDIA CUDA C Programming Guide 3.1 [online], http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf, [Ziyaret Tarihi: 28.10.2010].

6. Microsoft DirectX [online], http://msdn.microsoft.com/tr-tr/directx/, [Ziyaret Tarihi: 03.11.2010].

7. Open Source Computer Vision [online], http://opencv.willowgarage.com/wiki/, [Ziyaret Tarihi: 05.12.2010].

8. General-Purpose Computation on Graphics Hardware [online], http://gpgpu.org/, [Ziyaret Tarihi: 02.11.2010].

9. Embedded Metaprogramming Language [online], http://libsh.org/. [Ziyaret Tarihi: 05.01.2011].

10. BrookGPU [online], http://graphics.stanford.edu/projects/brookgpu/index.html, [Ziyaret Tarihi: 05.01.2011].

11. IAN, B., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M. and HANRAHAN, P., August 2004, *Brook for GPUs: Stream Computing on Graphics Hardware,* ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2004, 23(3), 777-786.

12. PODLOZHNYUK, V., 2007, *Image Convolution with CUDA* [online], http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf, [Ziyaret Tarihi: 14.12.2010].

13. PODLOZHNYUK, V., 2007, *FFT based 2D Convolution* [online], http://developer.download.nvidia.com/compute/cuda/2_2/sdk/website/projects/convolutionFFT2D/doc/convolutionFFT2D.pdf, [Ziyaret Tarihi: 15.12.2010].

14. KHARLAMOV, A. and PODLOZHNYUK, V., 2007, *Image Denoising* [online], http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/imageDenoising/doc/imageDenoising.pdf, [Ziyaret Tarihi: 24.12.2010].

15. CHE, S., BOYER, M., MENG, J., TARJAN, D., W. SHEAFFER, J. and SKADRON, K., October 2008, *A performance study of general-purpose applications on graphics processors using CUDA,* Journal of Parallel and Distributed Computing, 68(10), 1370-1380.

16. Open Multi-Processing [online], http://openmp.org/wp/, [Ziyaret Tarihi: 02.02.2011].

17. GARLAND, M., LE GRAND, S., NICKOLLS, J., ANDERSON, J., HARDWICK, J., MORTON, S., PHILLIPS, E., ZHANG, Y. and VOLKOV, V., 2008, *Parallel Computing Experiences with CUDA*, Micro, IEEE*,* 28(4), 13-27.

18. YANG, Z., ZHU, Y. and YONG, P., 2008, *Parallel Image Processing Based on CUDA*, 2008 International Conference on Computer Science and Software Engineering, 198-201.

19. ALLUSSE, Y., HORAIN, P., AGARWAL, A. and SAIPRIYADARSHAN, C., 2008, *GpuCV: An OpenSource GPU-Accelerated Framework for Image Processing and Computer Vision,* MM '08 Proceeding of the 16th ACM international conference on Multimedia, 1089-1092.

20. GPU-accelerated Computer Vision [online], https://picoforge.int-evry.fr/cgi-bin/twiki/view/Gpucv/Web/WebHome, [Ziyaret Tarihi: 24.11.2010].

21. HUANG, J., PONCE, S.P., PARK, S.I, YONG, C. and QUERK, F., 2008, *GPU-Accelerated Computation for Robust Motion Tracking Using the CUDA Framework*, 5th International Conference on Visual Information Engineering, 437-442.

22. LEI, P., LIXU, G. and JIANRONG, X., 2008, *Implementation of Medical Image Segmentation in CUDA,* International Conference on Information Technology and Applications in Biomedicine, 82-85.

23. FUNG, J. and MANN, S., 2008, *Using graphics devices in reverse: GPU-based Image Processing and Computer Vision,* IEEE International Conference on Multimedia and Expo, 9-12.

24. KIM, Y., HWANGBO, M. and KANADE, T., 2009, *Realtime Affine-photometric KLT Feature Tracker on GPU in CUDA Framework,* Robotic Institute, Carnegie Mellon University.

25. CABIDO, R., CONCHA, D., PANTRIGO, J.J. and MONTEMAYOR, A.S., 2009, *High Speed Articulated Object Tracking using GPUs: A Particle Filter Approach,* 10th International Symposium on Pervasive Systems, Algorithms, and Networks, 757-762.

26. BOYER, M., TARJAN, D., ACTON, S.T. and SKADRON, K., 2009, *Accelerating Leukocyte Tracking using CUDA: A Case Study in Leveraging Manycore Coprocessors,* IEEE International Symposium on Parallel&Distributed Processing, 1-12.

27. MATLAB [online], http://www.mathworks.com/products/matlab/, [Ziyaret Tarihi: 03.01.2011].

28. PRISACARIU, V.A. and REID, I., 2009, *fastHOG - a real-time GPU implementation of HOG*, Technical Report No. 2310/09, Department of Engineering Science, University of Oxford.

29. MEMBARTH, R., DUTTA, H., HANNING, F. and TEICH, J., 2011, *Efficient Mapping of Streaming Applications for Image Processing on Graphics Cards,* Transactions on HiPEAC, 5(3).

30. NVIDIA Tesla Workstation Solutions [online], http://www.nvidia.com/object/personal-supercomputing.html, [Ziyaret Tarihi: 05.03.2011].

31. NVIDIA CUDA C Programming Guide 1.0 [online], http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf, [Ziyaret Tarihi: 23.10.2010].

32. Wikipedia – CUDA [online], http://en.wikipedia.org/wiki/CUDA, [Ziyaret Tarihi: 17.10.2010].

33. XBITLABS Graphics Articles [online], http://www.xbitlabs.com/articles/graphics/display/gf8800.html, [Ziyaret Tarihi: 24.01.2011].

34. GLASKOWSKY, P.N., 2009, *NVIDIA's Fermi: The First Complete GPU Computing Architecture* [online], http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf, [Ziyaret Tarihi: 07.05.2011].

35. NVIDIA CUDA C Programming Guide 2.0 [online], http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf, [Ziyaret Tarihi: 21.11.2010].

36. OpenGL tutorials and notes [online], http://www.songho.ca/opengl/, [Ziyaret Tarihi: 02.03.2011].

37. BRADSKY, G. and KAEHLER, A., 2008, *Learning Opencv,* O'Reilly.

38. OpenGL Giriş [online], http://www.opengltr.com/, [Ziyaret Tarihi: 08.03.2011].

39. NVIDIA CUDA C Programming Guide 3.2 [online], http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf, [Ziyaret Tarihi: 11.12.2010].

40. Microsoft Visual Studio 2008 [online], http://www.microsoft.com/visualstudio/tr-tr, [Ziyaret Tarihi: 25.11.2010].

41. SANDERS, J. and KANDROT, E., 2010, *CUDA by Example,* Addison-Wesley.

42. WELCH, G. and BISHOP, G., 2006, *An Introduction to the Kalman Filter,* Department of Computer Science, University of North Carolina at Chapel Hill.

ÖZGEÇMİŞ

Ertan YILDIZ 29.11.1985 tarihinde Kocaeli/Türkiye'de doğdu. İlk, orta ve lise eğitimini Kocaeli'de tamamladıktan sonra, 2004 yılında İstanbul Üniversitesi, Mühendislik Fakültesi, Bilgisayar Mühendisliği bölümünde lisans eğitimine başladı. 2008 yılında mezun oldu. 2008 yılın Ekim ayında İstanbul Üniversitesi, Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Ana Bilim Dalı'na Yüksek Lisans eğitimine başladı.