GÖMÜLÜ SİSTEMLER ÜZERİNDE OPENCL TABANLI GÖRÜNTÜ İŞLEME KÜTÜPHANESİ VE KERNEL FÜZYON

HAKKI DOĞANER SÜMERKAN

YÜKSEK LİSANS TEZİ

BİLGİSAYAR MÜHENDİSLİĞİ

TOBB EKONOMİ VE TEKNOLOJİ ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

TEMMUZ 2014

ANKARA

Fen Bilimleri Enstitü onayı

_____

Prof. Dr. Osman EROĞUL
Müdür

Bu tezin Yüksek Lisans derecesinin tüm gereksinimlerini sağladığını onaylarım.

_____

Doç. Dr. Erdoğan DOĞDU
Anabilim Dalı Başkanı

HAKKI DOĞANER SÜMERKAN tarafından hazırlanan GÖMÜLÜ SİSTEMLER ÜZERİNDE OpenCL TABANLI GÖRÜNTÜ İŞLEME KÜTÜPHANESİ VE KERNEL FÜZYON adlı bu tezin Yüksek Lisans tezi olarak uygun olduğunu onaylarım.

_____

Doç. Dr. Oğuz ERGİN
Tez Danışmanı

Tez Jüri Üyeleri

Başkan : Yrd. Doç. Dr. Ahmet Murat Özbayoğlu _____

Üye : Doç. Dr. Oğuz ERGİN _____

Üye : Yrd. Doç. Dr. Zülfükar Saygı _____

TEZ BİLDİRİMİ

Tez içindeki bütün bilgilerin etik davranış ve akademik kurallar çerçevesinde elde edilerek sunulduğunu, ayrıca tez yazım kurallarına uygun olarak hazırlanan bu çalışmada orijinal olmayan her türlü kaynağa eksiksiz atıf yapıldığını bildiririm.

Hakkı Doğaner SÜMERKAN

Üniversitesi	**:**	**TOBB Ekonomi ve Teknoloji Üniversitesi**
Enstitüsü	**:**	**Fen Bilimleri**
Anabilim Dalı	**:**	**Bilgisayar Mühendisliği**
Tez Danışmanı	**:**	**Doç. Dr. Oğuz ERGİN**
Tez Türü ve Tarihi	**:**	**Yüksek Lisans – Temmuz 2014**

Hakkı Doğaner SÜMERKAN

GÖMÜLÜ SİSTEMLER ÜZERİNDE OPENCL TABANLI GÖRÜNTÜ İŞLEME KÜTÜPHANESİ VE KERNEL FÜZYON

ÖZET

Gelişen teknoloji ile birlikte, mobil sistemlerin hesaplama kapasitesi de artmaktadır. Modern gömülü platformlar grafik işlemlerini daha iyi yapabilmek ve daha iyi bir arayüz sunabilmek için grafik işlem birimleri(GPU) bulundurmaktadırlar. Mobil aygıtların, görüntü işleme gibi yoğun hesaplama gücü gerektiren uygulamaları başarılı bir şekilde çalıştırabilmeleri amacıyla bazı mobil platform üreticileri kendi platformlarına OpenCL destekli GPU'lar koymaktadırlar. Böylelikle genel amaçlı grafik işlem birimi(GPGPU) altyapısından faydalanarak bu tür hesaplama yoğun işleri yapabilmeyi hedeflemişlerdir. Bu sayede GPGPU altyapısının sunduğu hesaplama gücünü, mobil platformlar için önemli bir kısıt olan güç verimliliğini de sağlayarak yerine getirebilmektedirler.

Bu çalışmada, OpenCL tabanlı, gömülü platformlar üzerinde çalışabilen bir görüntü işleme kütüphanesi sunulmuştur. Deneysel sonuçlar görüntü işleme alanında bir standart haline gelen OpenCV ile karşılaştırılmış ve ortalamada 7 kat hızlanma sağlanmıştır.

Anahtar Kelimeler: GPGPU; OpenCL; Gömülü Sistemler;Paralel Hesaplama; Görüntü İşleme.

University	:	**TOBB University of Economics and Technology**
Institute	:	**Institute of Natural and Applied Sciences**
Science Programme	:	**Computer Engineering**
Supervisor	:	**Assoc. Prof. Oğuz ERGİN**
Degree Awarded and Date	:	**M.Sc. – JULY 2014**

Hakkı Doğaner SÜMERKAN

OPENCL BASED IMAGE PROCESSING LIBRARY AND KERNEL FUSION ON EMBEDDED SYSTEMS

ABSTRACT

Embedded computing is an increasingly improving area. With every new generation, the computation capacity of the mobile devices is improving. Contemporary mobile devices include a graphical processing unit (GPU) in order to offer better use interface in terms of graphics. As mobile devices require more performance to cope with computation-intensive applications, such as image processing, recently some embedded GPUs also included the support for OpenCL that allows the use of computation capacity of embedded GPUs for general purpose computing. Exploiting GPGPU on embedded platforms is also more efficient in terms of energy efficiency.

In this thesis, we present a new OpenCL-based image processing library, which is specifically designed to run on an embedded platform. Our results show that the functions of TRABZ-10 show 7x speedup on embedded platform over the functions of OpenCV on average.

Keywords: GPGPU; OpenCL; Embedded Systems;Parallel Computing; Image Processing.

TEŞEKKÜR

Bana bu çalışmada ve yüksek lisans eğitimim süresince çeşitli konularda araştırma olanağı sağlayan, uyarıları ve yönlendirmesi ile sonuca ulaşmama yardımcı olan Sayın tez danışmanım Doç. Dr. Oğuz Ergin'e; tez jürimde yer alan değerli hocalarıma; öğrenim hayatım boyunca bana emeği geçen tüm TOBB ETÜ ailesine,kıymetli çalışma arkadaşlarım Osman Seçkin Şimşek, Mustafa Çavuş, Hasan Hassan,Abdullah Giray Yağlıkçı,Muhammet Özgür ve diğer Kasırga Mikroişlemciler Labaratuvarı üyelerine;bu projeyi SAN-TEZ programı kapsamında destekleyen Bilim Sanayi ve Teknoloji Bakanlığına, çalışma ortamımızı sağladığı için TOBB ETÜ Fen Bilimleri Enstitüsüne ve her zaman sabırla bizlere yardımcı olan enstitü sekreteri Sayın Ülüfer Nayır'a teşekkürlerimi sunarım.

Çalışmalarım süresince her türlü yardımı esirgemeden bana destek olan çok değerli aileme minnet, şükran ve teşekkürlerimi sunmayı zevkli bir görev sayarım.

İçindekiler

Şekil Listesi

Tablo Listesi

1. GİRİŞ

Günümüzde grafik işlem birimleri yüzlerce paralel işlem birimi barındırabilirken bu işlem birimleri binlerce paralel iş parçacığı çalıştırabilmektedir. Bu donanımsal altyapı grafik işlem birimlerinin genel amaçlı olarak kullanılmaya başlanması ile programlanabilir hale gelmiş ve grafik işlem birimleri aritmetik işlem gücü ve bellek bant genişliği bakımından merkezi işlem birimlerinin performansına ulaşmış ve hatta önüne geçmiştir.

Grafik işlem birimlerinin bu performans ve güç verimliliğinin merkezi işlem birimlerinin önüne geçmesi sonucunda ise bilgisayar endüstrisi çok çekirdekli paralel mimariler üzerinde çalışmalara odaklanmıştır. Bunun sonucu olarak ise GPU programlama kütüphaneleri iyileştirilmeye başlanmış, OpenGL (Open Graphics Library) [1] grafik kütüphanesi entegrasyonu ve OpenCL (Open Computing Language) [2], CUDA gibi farklı mimariler üzerinde çalışabilen diller geliştirilmiş ve grafik işlem birimleri yaygın olarak genel amaçlı olarak kullanılmaya başlanmıştır.

OpenCL, (NVIDIA ve AMD gibi) çeşitli üreticilerin masaüstü GPU'ları da dahil olmak üzere çeşitli platformlarda çalışabilen, platformdan bağımsız, açık bir standart olup, GPU, CPU, FPGA ve OpenCL ile kullanılabilen diğer her türlü işlem birimi gibi birçok işlem kaynağının eş zamanlı kullanımını destekler. OpenCL açık bir standart olduğundan, her türlü üretici, ürünlerinde OpenCL desteği verebilir.

Geniş bir yelpazede uygulamalarla uyumlu çalışan OpenCL'in yaygın kullanım alanlarından biri, görüntü / video işlemedir. Görüntü işleme alanındaki en önemli zorluklardan biri, yüksek doğruluk ve gerçek zamanlı işleme sağlamak için yüksek işlem gereksinimidir. Video çözünürlüklerinin ve veri boyutlarının artması sonucunda, video karelerinin işlenmesi sırasında gerçek zamanlı performans elde etmek zorlaşmaktadır. Buna ek olarak, cep telefonu, tablet gibi çeşitli gömülü cihazlarda yüksek çözünürlüklü video desteklenir. Bu da, başka bir yaygın sorun olan, performans kaybı yaşamaksızın düşük güç gereksinimini ortaya çıkarır.

Güç verimliliği ve yüksek veri paralelliğine sahip hesaplamaya olanak tanıyan GPGPU(General Purpose GPU), tasarımcılara seçenek sunmaktadır. Çoğu görüntü işleme algoritması, veri paralelliğine sahip uygulamalar üzerinde iyi bir performans gösterir. Dolayısıyla bu algoritmalar, GPU'ların tek buyruk çoklu veri (SIMD) mimarisinden faydalanır ve etkin bir biçimde paralel hale getirilebilir. Ancak, tüm görüntü işleme algoritmalarının veri paralelliği yaklaşımına uyum gösterdiğini söylemek mümkün değildir ve bu algoritmalar, GPU'lara aktarıldığında kayda değer bir hızlanma sağlamaz. Görüntü işleme algoritmalarının, GPU'ların çok büyük ölçekte paralel mimarisine iyi uyum sağlamasına karşın, algoritmaların bazıları, performansta hızlanma sağlamayı başaramaz. Buna ek olarak, GPU'ların donanım kısıtlamaları nedeniyle, bazılarının aktarılması mümkün değildir. Bunların tümüne ek olarak, gömülü mimari kısıtlamaları ek kısıtlamalar getirir ve bu durum, belirli işlevlerin GPU'ya aktarılmasını zorlaştırır.

Grafik işlem birimlerinin genel amaçlı olarak kullanımının kolaylaşması ve bunun gömülü sistemler üzerinde de desteklenmesi sonucu bu altyapının kullanılmasına yönelik çeşitli araştırmalar olmuştur. Görüntü işleme algoritmalarının da SIMD yapısına uygun , paralel olarak çalıştırılabilen algoritmalar olması nedeniyle bu tezde gömülü sistemler üzerinde de çalışabilen bir görüntü işleme kütüphanesi geliştirilmiştir. OpenCL tabanlı olarak geliştirilen bu kütüphane OpenCL destekli tüm aygıtlarda çalışabilmektedir. Bu tezde sunulan görüntü işleme kütüphanesi fonksiyonel ve performans sonuçlar açısından bu alanda bir standart haline gelmiş bir kütüphane olan OpenCV ile karşılaştırılmıştır.Tezin geri kalan kısmı şu şekilde düzenlenmiştir. 2. Bölümde OpenCL standardının ana yapısı, çalışma prensibi ve mimarisi hakkında bilgilendirme yapılmıştır. 3. Bölümde geliştirdiğimiz kütüphanede bulunan fonksiyonlar ve algoritmalar hakkında açıklamalar yapılmış ve bazı algoritmaların implementasyonları ile ilgili bilgiler verilmiştir. 4. Bölüm algoritmaların geliştirilmesinden sonra yapılan iyileştirme metotlarından bahsederken 5. Bölüm performans sonuçlarını OpenCV ile karşılaştırarak sunmaktadır.

2. GRAFİK İŞLEM BİRİMLERİ, GPGPU ve OPENCL

2.1 Grafik İşlem Birimleri ve GPGPU Kavramı

Grafik işlem birimleri(GPU) bilgisayarlarda ekrana görüntü vermeye yarayan kartlardır. Esas olarak grafik ile ilgili işlemleri yapmalarının yanında günümüzde genel amaçlı olarak da kullanılmaya başlanmıştır. Genel amaçlı grafik işlem birimi(GPGPU) yapısının öne çıkmaya başlamasıyla yüksek performanslı ve verimli bir şekilde işlem yapma amaçlı kullanılmaya başlanmıştır. Grafik işlem birimlerinin, merkezi işlem birimlerinden farklı kılan temel özellik az sayıda güçlü işlem birimlerinden değil de çok sayıda ve basit işlemleri yapmak için tasarlanmış işlem birimlerinden oluşmasıdır. Bu sayede GPU'lar tekli buyruk çoklu veri(SIMD) yapısında çalışarak ilgili veriyi paralel bir şekilde işleyebilmektedirler.

GPGPU bir çok uygulama alanında kazanımlar getirirken özellikle ses işleme, görüntü işleme, şifreleme, bioinformatik ve bazı bilimsel hesaplamalar gibi büyük bir veri üzerinde küçük iş parçacıklarının çalışabilmesine uygun algoritmik adımları olan uygulama alanlarında başarılıdırlar. Bu tezde görüntü işleme alanına yoğunlaşılmıştır. Bu alanda en önemli zorluklardan biri gerçek zamanlı video işlemeyi yakalayabilmek için yüksek hesaplama gereksinimidir. Gün geçtikçe büyüyen resim çözünürlükleri ve veri büyüklükleri nedeni ile gerçek zamanlı video işleme daha zor bir hale gelmektedir. Bunun yanında gelişen teknolojinin sunduğu işlem gücü bu zorluğu nispeten kaldırmaktadır. Bunlara ek olarak yüksek çözünürlüklü videolar cep telefonları, tabletler gibi gömülü sistemler üzerinde de desteklenmektedir. Bu durum ise diğer bir önemli problem olan düşük güç gereksinimini ön plana çıkarmaktadır. Buna karşın GPGPU kullanıcılara güç tüketimi açısında verimli paralel hesaplama kabiliyeti sunmaktadır. Bir çok görüntü işleme kütüphanesi veri bazında paralelleştirmeye uygun algoritmalardan oluşmaktadır. Bu sayede SIMD yapısına uygun olan GPU'lar üzerinde verimli bir şekilde paralelleştirilebilirler. Fakat bütün görüntü işleme algoritmaları paralel

olarak GPU'lar üzerinde geliştirilebilir diyemeyiz. Üzerinde geliştirme yapılacak olan GPU'nun donanımsal kısıtlarının yanında gömülü sistemler üzerinde çalışmanın verdiği bazı ek kıstılar da bulunmaktadır. Herhangi bir görüntü işleme algoritmasını GPU'lar üzerinde paralel olarak geliştirmede en büyük engel bu kısıtlardır. Bunlar tezin ilerleyen bölümlerinde anlatacağı gibi ana olarak bellek kısıtlar ve algoritmik yapıdan kaynaklanan kısıtlar gibi maddelerden bahsedilebilir.

GPGPU'ların daha iyi performans elde etmek için görüntü işleme işlemlerinde kullanılması oldukça yaygınlaşmaktadır. Önce DirectX ve OpenGL ile başlayan bu akım günümüzde daha çok OpenCL ve CUDA ile devam etmektedir. Bu konuda girişimler OpenGL ve DirectX uygulama programlama arayüzlerinin(API) geliştirilmesiyle, GPU shader çekirdeklerinin ve programlanabilir boru-hattının GLSL kullanarak geliştirmek olmuştur[2, 5]. Genel amaçlı GPU modeli ilk olarak programlanabilir shader altyapısını kullanarak geliştirilmiştir. Programlanabilir shader altyapısı kullanılarak genel amaçlı GPU modeli elde edilmiştir. GLSL, sadece grafik shader kodu değil, aynı zamanda genel amaçlı hesaplama da sağlayan shader programlama dilidir.

GPU'ların hesaplama ve programlama kabiliyetlerini değiştiren NVidia, "Compute Unified Device Architecture(CUDA)"[11] geliştirerek, GPU ve GPGPU'ların tekli komut çoklu iş parçacığı (SIMT) ve tekli buyruk çoklu veri(SIMD) mimarisini desteklemesini sağlamıştır. Bu mimariler, tek bir komutun birçok iş parçacığı tarafından kullanılmasına olanak sağlayıp, birden çok veri işleyebilir ve böylece, büyük miktarda performans kazancı ortaya koyar. CUDA mimarisinin ve ilgili programlama arayüzlerinin geliştirilmesinin ardından Apple önderliğinde altyapısı oluşturulan ve Khronos tarafından spesifikasyonları belirlenen OpenCL sunuldu. OpenCL platform bağımsız bir yani farklı üreticilerin bir çok farklı mimarisi üzerinde koşturulabilen bir standart iken CUDA sadece NVIDIA platformlarında koşturulabilen bir alt yapısı sunmaktadır. OpenCL CPU, GPU, DSP,FPGA gibi bir çok hesaplama birimi üzerinde çalışabilmektedir. Bunun yanında OpenCL açık bir standart olmasından dolayı herhangi bir üretici istediği şekilde implement edebilir. Bu tezde belirtildiği üzere geliştirdiğimiz kütüphane

gömülü sistemleri ve başka mimarileri de destekleyeceğinden dolayı OpenCL kullanılarak geliştirilmiştir.

Görüntü işleme uygulamalarında bu performanstan faydalanmak için, GPGPU'lar kapsamlı bir biçimde kullanılmıştır. Bazı araştırmalarda görüntü işleme fonksiyonlarının sıfırdan uygulanmasına karşın[12], bazılarında bu fonksiyonlar mevcut kütüphaneler üzerinden uygulanır[13, 14]. Belirli görüntü işleme algoritmaları hakkında yapılmış araştırmalar da mevcuttur. Bunlara ek olarak, Luo ve ark. Canny kenar algılama algoritmasını güçlendirmek için CUDA kullanmıştır [15]. Singhal ve ark. elde taşınan cihazlar için birçok görüntü işleme, renk çevrim ve dönüşüm algoritması ile Harris köşe algılama ve gerçek zamanlı video boyutlandırma gibi uygulamalar yürütmüş olup, bu amaç için OpenGL ES[4] ile doku(texture) donanımını kullanmıştır[7]. GPUCV[8], MinGPU[9] ve OpenVIDIA[10] gibi, açık kaynak kodlu başka projeler de mevcuttur. GPUCV ve MinGPU'nun aynı anda GPU ve GPGPU işlevselliklerinin kullanılmasını desteklemesine karşın, OpenVIDIA yalnızca CUDA kullanır ve filtreleme fonksiyonları başta olmak üzere bir dizi görüntü işleme fonksiyonunu uygular. Bu ve benzer gelişmelerin ardından CUDA ve OpenCL bu kullanımı üst seviyelere taşınarak günümüzdeki yoğunluğuna ulaşmıştır. Geliştirdiğimiz görüntü işleme kütüphanesi OpenCL altyapısı kullanılarak implement edilmesi ile nedeniyle OpenCL daha detaylı bir şekilde anlatılmıştır.

2.2 OpenCL

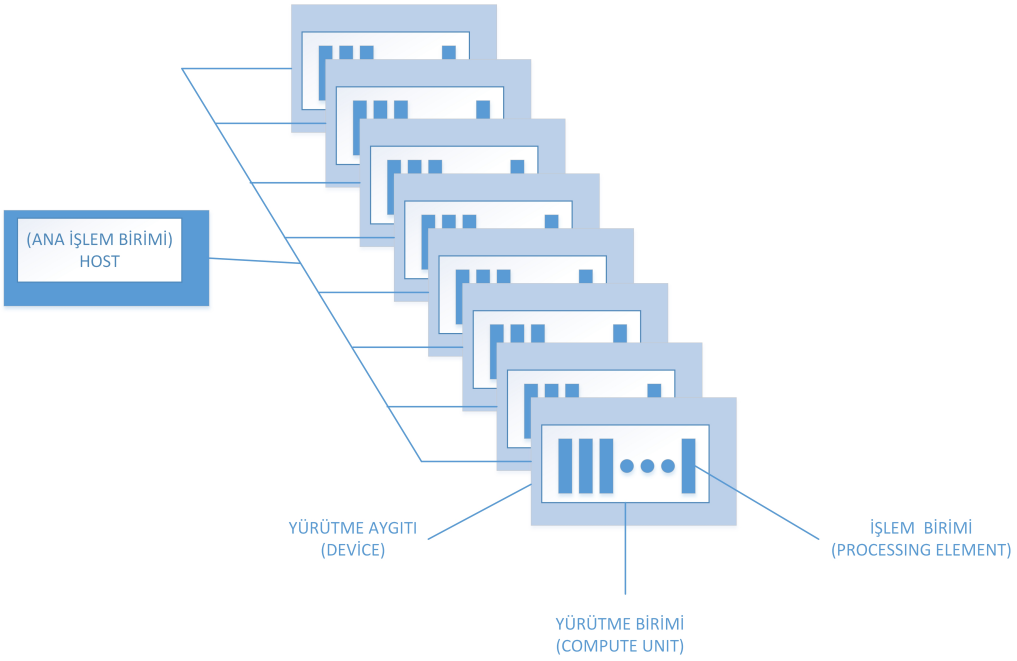
OpenCL farklı işlem birimlerinde(GPU, DSP, FPGA, CPU. .) heterojen olarak paralel programlama yeteneği sunan bir programlama arayüzüdür(API). Khronos grup tarafından spesifikasyonları belirlenen ve tanımlanan bu arayüz bir çok farklı üretici tarafından implement edilmektedir. OpenCL standardını ana hatlarıyla anlatabilmek için 4 ana model altında inceleyebiliriz.

- OpenCL Platform Modeli

- OpenCL Bellek Modeli
- OpenCL Yürütme Modeli
- OpenCL Programlama Modeli

2.2.1 OpenCL Platform Modeli

OpenCL platform modeline göre OpenCL platformu bir "host" ve bir veya birden çok OpenCL "device" barındırır. "Host" paralel işlem birimlerinde çalıştırılacak olan program parçacığının ana konfigrasyonlarını yapan işlem birimi iken "device" bu kod parçacığının paralel olarak çalıştırıldığı işlem birimidir. OpenCL device bir veya birden fazla yürütme birimine(Compute Units-CEs) ayrılmıştır ve bu yürütme birimleri de bir veya birden çok işlem biriminden(Processing Units-PEs) oluşur. OpenCL platform modeli Şekil 2.1'de gösterilmiştir.

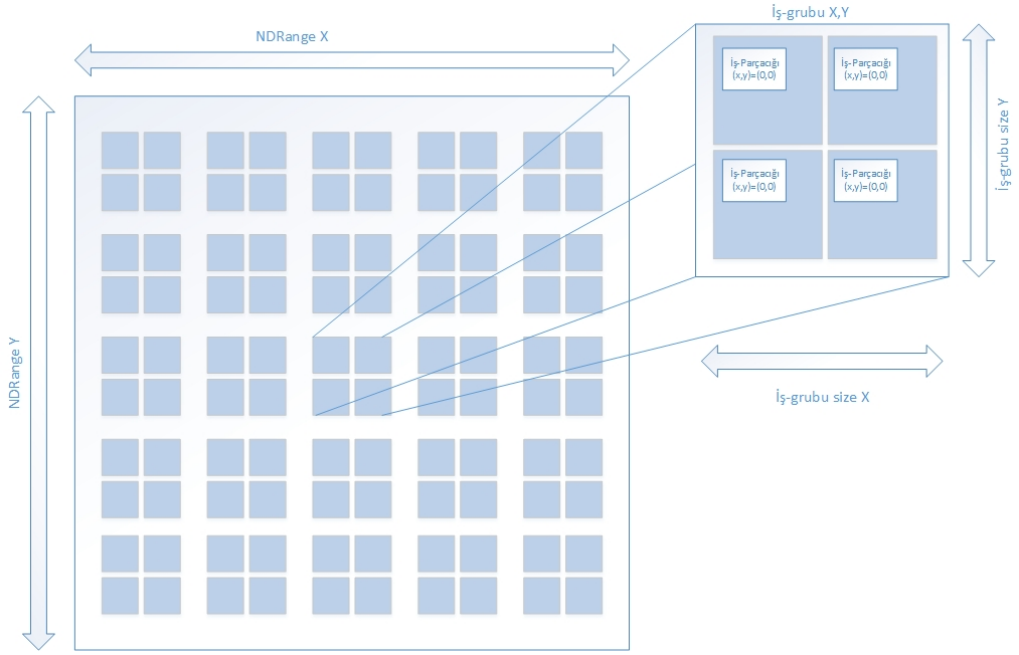


Şekil 2.1: OpenCL Platform Modeli

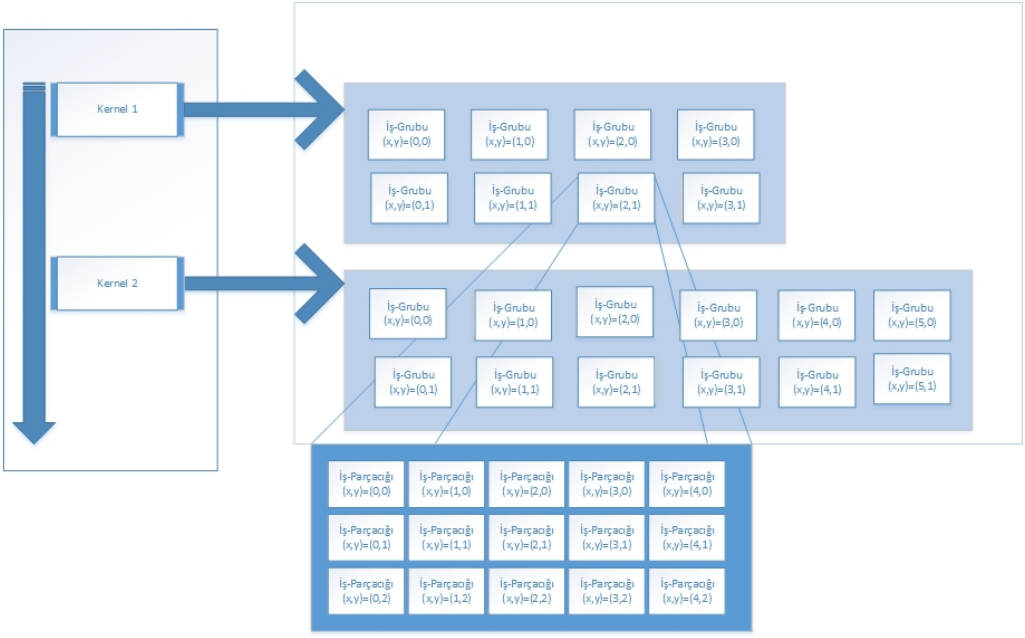
2.2.2 OpenCL Yürütme Modeli

OpenCL yürütme modeli iki ana bölümden oluşur. OpenCL device üzerinde çalışan kısmına "kernel" denirken, host tarafında çalışan kısıma host program denir. Genellikle grafik kartları ve merkezi işlem biriminden oluşan bir platformda CPU üzerinde çalışan kısıma host program, GPU üzerinde çalışan kısıma ise kernel diyebiliriz. Ancak OpenCL destekli CPU'lar üzerinde de kernel programı çalıştırılabilir. Bu durumda CPU'ların biri host olurken diğerleri device durumunda çalışacaktır.

OpenCL yürütme modeli daha çok kernelların çalışma altyapısını belirleyen modeldir. Host tarafından, bir kernel programı, device tarafına gönderildiğinde indeks alanı oluşturulur ve kernelların bu indeks aralığında kendi verileri üzerinde çalışan iş parçacıklarını(work-item) barındırılır. Her bir iş-parçacığı indeks alanında bulunan ve kendi kullandığı veri noktası ile ayırt edilir. Yani her bir iş-parçacığı bu indeks alanında kullandıkları verinin ID'lerine göre bir diğerinden ayırt edilebilir. Bu ID'ler genel indeks ve yerel indeks bilgisini barındırırlar. İş-parçacıkları , kernellar tarafından eş zamanlı olarak yürütülür. Her bir iş-parçacığı aynı kernel fonksiyonunu indeks alanından kendine yöneltilen veri üzerinde diğer iş-elemanları ile paralel olacak şekilde yürütülür. Bir iş-elemanı diğerinden genel ve yerel indeks bilgisi kullanılarak ayırt edilebilir. Bir ve ya birden çok iş-parçacığı, iş-grubunu(work-group) oluşturur. İş-grubu altındaki iş-parçacıkları bir sonraki bölümde daha detaylı anlatacağımız yerel bellek türünü ortak olarak kullanabilirler. Tüm veriyi kapsayan indeks alanına NDRange denir. OpenCL yürütme modeli Şekil 2.2 ve Şekil 2.3'de gösterilmiştir.



Şekil 2.2: OpenCL Yürütme Modeli



Şekil 2.3: OpenCL Yürütme Modeli

2.2.3 OpenCL Bellek Modeli

Bellek hiyerarşisini ve yapısını detaylandıran bellek modeli, OpenCL arayüzünde kullanım alanlarına, büyüklüklerine ve erişim sürelerine göre kademelere ayırabileceğimiz bir bellek modeli bulunmaktadır. Bu modele göre 4 çeşit bellek bulunmaktadır.

2.2.3.1 Genel(Global) Bellek:

Bu bellek alanına tüm iş-grupları ve bu iş-grupları altındaki tüm iş-parçacıkları okuma ve yazma yapabilirler.

2.2.3.2 Değişmez(Constant) Bellek:

Genel belleğin, bir kernelın çalıştırılması sırasında sabit kalan bölümüdür. Değişmez belleğe yerleştirilen bellek nesneleri, ana bilgisayar tarafından atanır ve başlatılır.

2.2.3.3 Yerel(Local) Bellek:

Bir çalışma grubunun yerel bellek bölgesidir. Bu bellek bölgesi, söz konusu çalışma grubundaki tüm iş parçacıklarının paylaştığı değişkenler atamak için kullanılabilir. Bu belleği, OpenCL cihazı üzerindeki belleğin ayrılmış bölgeleri olarak uygulamak mümkündür. Alternatif olarak, yerel bellek bölgesi, genel belleğin bölümleri üzerine eşlenebilir.

2.2.3.4 Özel(Private) Bellek:

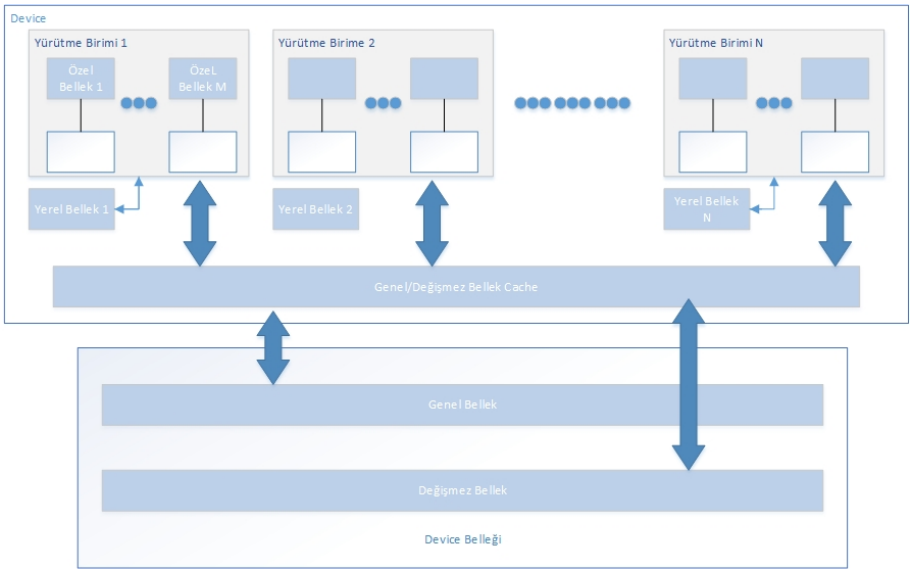
Belleğin, bir çalışma unsuruna özel bölgesidir. Bir çalışma unsurunun özel belleği üzerinde tanımlanan değişkenler, başka bir çalışma unsuru tarafından görülmez.

Tablo 2.1 OpenCL bellek modeli genel hiyerarşisi hakkında özet bilgiyi sunmaktadır.

Tablo 2.1: OpenCL Bellek Hiyerarşisi

	Genel	Değişmez	Yerel	Özel
Host	Dinamik Tahsis Yazma/Okuma Yetkisi	Dinamik Tahsis Yazma/Okuma Yetkisi	Dinamik Tahsis Yazma/Okuma Yetkisi	Dinamik Tahsis Yazma/Okuma Yetkisi
Kernel	Dinamik Tahsis Yazma/Okuma Yetkisi	Dinamik Tahsis Yazma/Okuma Yetkisi	Dinamik Tahsis Yazma/Okuma Yetkisi	Dinamik Tahsis Yazma/Okuma Yetkisi

Bellek alanları ve bu alanların platform modeli ile olan bağlantıları Şekil 2.4'de gösterilmiştir.



Şekil 2.4: OpenCL Bellek Modeli

2.2.4 OpenCL Programlama Modeli

OpenCL yürütme modeli, veri paralel ve görev paralel programlama modellerini ve bu modellerin melez modellerini destekler. OpenCL'in tasarımının işlemesini sağlayan birincil model, veri paralel modeldir.

2.2.4.1 Veri Paralel Programlama Modeli

Veri paralel programlama modeli, hesaplamayı, bir buyruk dizisinin çoklu bellek elemanına uygulanması şeklinde tanımlar. İş-parçacıkları ve verilerin iş-parçacıkları ile nasıl eşleneceği, OpenCL yürütme modeliyle özdeşleştirilen indeks alanı tarafından tanımlanır. Mutlak veri paralelliğine sahip bir modelde, iş-parçacıkları ile bellek nesnesinde, çekirdeğin üzerinde paralel olarak yürütülebileceği eleman sayısı arasında bire bir eşleme mevcuttur. OpenCL, mutlak birebir işlemenin gerekli olmadığı veri paralel programlama modelinin daha gevşek bir versiyonunu uygular.

OpenCL, hiyerarşik bir veri paralel programlama modeli sunar. Hiyerarşik alt bölünmeyi belirtmenin iki yolu vardır. Açık modelde, paralel olarak yürütülmesi gereken toplam iş-parçacığı sayısını ve iş-parçacıklarının iş-gruplarına nasıl bölüneceğini programcı tanımlar. Kapalı modelde, programcı yalnızca paralel olarak yürütülecek iş-parçacıklarının toplam sayısını belirtir ve iş-gruplarına bölünme, OpenCL'in uygulamasıyla yönetilir.

2.2.4.2 Görev Paralel Programlama Modeli

OpenCL görev paralel programlama modeli, çekirdeğin tek bir örneğinin herhangi bir indeks alanından bağımsız yürütüldüğü bir model tanımlar. Bu da mantıksal olarak, çekirdeği, tek bir iş-parçacığı içeren iş-grubuna sahip bir hesaplama ünitesi üzerinde yürütmeye denktir. Bu model kapsamında, kullanıcılar paralelliği şöyle ifade eder:

- Device tarafından implement edilmiş vektör veri tiplerini kullanmak

- Birden fazla görevi kuyruk halinde sıralayarak ve(ya)

- OpenCL'e ortagonal bir programlama modeli ile geliştirilen kernelları kuyruk halinde sıralayarak

2.2.4.3 Senkronizasyon

OpenCL'de iki senkronizasyon alanı mevcuttur:

- Tek bir iş grubundaki iş-parçacıkları

- Tek bir context içerisinde buyruk kuyruğuna sokulmuş buyruklar

Tek bir iş-grubu içerisindeki iş-parçacıkları arasındaki senkronizasyon, iş-grubu bariyeri(barrier) kullanılarak gerçekleştirilir. Bir iş-grubunun tüm iş-parçacıklarını, bariyerin ötesinde yürütmeye devam etmelerine izin verilmeden önce bariyeri yürütmesi gerekir. İş-grubu bariyerinin, çekirdeği yürüten iş-grubunun içindeki tüm iş-parçacıklarının tümü tarafından karşılanması veya hiçbiri tarafından karşılanmaması gerekir. İş-grupları arasında senkronizasyon için bir mekanizma mevcut değildir.

Buyruk kuyruklarında buyruklar arasındaki senkronizasyon noktaları şunlardır:

- Buyruk kuyruğu bariyeri: Buyruk-kuyruk bariyeri, daha önce kuyruğa sokulan buyrukların yürütülmesinin sona ermesini ve bunun sonucunda bellek nesnelerinde gerçekleşen her türlü güncellemenin, sonrasında kuyruğa sokulan buyruklara, bu komutların yürütülmeye başlamasından önce görünür olmasını sağlar. Bu bariyer, ancak tek buyruklı kuyruktaki buyruklar arasında senkronizasyon sağlamak için kullanılabilir.

- Yeni bir fonksiyonu beklemek: Buyrukları kuyruğa sokan tüm OpenCL API fonksiyonları, güncellediği komutu ve bellek nesnelerini belirleyen bir olay döndürür . Olaya eşlik eden daha sonraki bir buyruğa, bu bellek nesnelerine yapılan güncellemelerin, buyruk yürütülmeye başlamadan önce görünür olduğu garanti edilir.

2.2.5 Gömülü Platformlar üzerinde OpenCL

Diğer Khronos standartlarının aksine, OpenCL'in ES (gömülü sistemler) özelliği mevcut değildir fakat gömülü profil, OpenCL özelliklerine dahildir. Gömülü profil, OpenCL standardının bir alt kümesidir ve üreticilerin gerekli fonksiyonları uygulaması gerekir. Örneğin, gömülü profilin double, long veya half veri tiplerini desteklemesi gerekmez, yuvarlama modları sıfıra yuvarlama gibi orta düzeyde olabilir veya 3D görüntü desteği zorunlu değildir.

OpenCL'in gömülü profil özelliğine sahip olmasına karşın, yakın zamana kadar buna uygun cihaz veya üretici yoktu. Gömülü profilli OpenCL üzerinde uygulamak için bazı çalışmaların yapılmasına karşın, donanım desteği olmadan, bunlar olması gerektiği kadar verimli değildi[23]. Bu araştırmalar arasındaki ana fikir, OpenGL ES 2. 0 destekleyen gömülü sistem GPU'larının kullanılmasıydı. OpenGL ES 2. 0, programlanabilir shader'ların kullanımına olanak sağlar ve OpenGL arayüzü sayesinde, kapsamlı bir çalışma ile kullanışlı fonksiyonlar uygulanabilir[24]. Bu tezde kullanılan, Vivante Corporation'ın GPGPU'ları, OpenCL 1.1 Gömülü versiyonunu desteklemektedir.

GPGPU üzerindeki en yaygın uygulamalar, hızlandırıcı görüntü işleme algoritmalarıdır. Görüntü işleme işlemleri, birçok piksel üzerinde aynı hesaplamayı gerçekleştirir, tek komut çoklu veri (SIMD) mimarisini kullanabilir ve CPU uygulamalarından daha iyi performans gösterebilir.

Görüntü işleme işlemlerinde daha iyi bir performans elde etmek için Genel Amaçlı Grafik İşleme Ünitelerinin (GPGPU) kullanılması, son yılların en yaygın konularından biridir. Bu konuda ilk yaklaşımlar, GPU'ların shader çekirdekleri ve GLSL kullanan programlanabilir boru-hattı programlanması için OpenGL ve DirectX API'lerinin kullanılmasını içerir. CUDA ve OpenCL'in geliştirilmesi, GPGPU'ların görüntü işleme ve diğer uygulamalar için kullanımını en üst düzeye çıkarmıştır.

3. TRABZ-10 GÖRÜNTÜ İŞLEME KÜTÜPHANESİ

3.1 Giriş

TRABZ-10, gömülü sistemler üzerinde çalıştırılmaya uygun olarak geliştirilmiş bir görüntü işleme kütüphanesidir. C, C++ ve OpenCL tabanlı olan bu kütüphane Linux ve Windows işletim sistemleri üzerinde test edilmiş ve çalışır durumdadır. Bu kütüphanenin geliştirilme amacı gömülü sistemler üzerinde çalışabilen ve OpenCL'in sunduğu paralel hesaplama altyapısını kullanabilen ve bu sayede görüntü işleme uygulamalarını daha az güç kullanarak ve daha hızlı bir şekilde çalıştırabilmektir. Gömülü sistem üzerinde geliştirme ve test Freescale i. Mx6q platformunda yapılmıştır. Bu platform OpenCL destekli Vivante GC2000 GPU'ya sahip bir platformdur. Freescale i. Mx6q platformu güçlü ve verimli bir platform olmasına rağmen bazı dezavantajlara da sahiptir. Bu eksiklikler mobil bir platform olmasından ve doğal olarak bazı ödünleşmelerin verilmesinden dolayı kaynaklanan problemlerdir. Düşük güç tüketimine sahip, küçük boyutlu bir GPGPU üretebilmek için Vivante, performanstan bir miktar ödün vermiştir. Cihaz üzerinde küçük boyutlu bir bellek ve sınırlı sayıda yazmaç mevcuttur. Yerel bellek, çip üzerinde değil, genel belleğin bir parçasıdır. Bu nedenle, yerel belleğin kullanılması performansı yükseltmez. CPU'lar kayan nokta işlemlerinde daha yüksek hassasiyette işlem yapabildikleri için, sonuçlar arasında hassasiyetten kaynaklanan farklar olabilir.

TRABZ-10 görüntü işleme kütüphanesi fonksiyonları ana olarak 6 grup altında toplanmıştır. Bu gruplar şu şekildedir;

- Dizi ve Matris İşlemleri
- Filtreleme İşlemleri
- Morfolojik Operasyonlar

- Dönüşüm İşlemleri
- Renk Küremsi Dönüşüm İşlemleri
- Histogram İşlemleri

3.1.1 Dizi ve Matris İşlemleri

Bu grupta bulunan fonksiyonların altında temel matris işlemleri bulunmaktadır. Bu işlemler matris toplama, çıkarma, çarpma, tersini alma gibi fonksiyonlardır. Matris işlemleri genel yapısı itibari ile GPU üzerinde işlemeye uygun işlemlerdir. Temel olarak yapılan tek bir işlem bütün matris elemanları üzerinde aynı şekilde yapılacağından SIMD yapısını oldukça verimli bir şekilde kullanabilmektedir. Bu nedenle bu grup altında bulunan fonksiyonların bir çoğu GPU üzerinde gerçekleştirilmiş ve önemli performans kazanımları elde edilmiştir. Min,max gibi matris üzerinde en büyük veya en küçük değeri bulmaya yarayan bazı fonksiyonlar ise GPU performansını olumsuz etkileyecek şekilde dallanmalara sebep olduğu için CPU üzerinde geliştirilmiştir. Bunlara ek olarak determinant alma ve matrisin tersini alma işlemleri de CPU üzerinde geliştirilmiştir. Bu fonksiyonlar algoritma yapıları nedeniyle özyinelemeli olarak çalışmaktadır. OpenCL özyinelemeli fonksiyonları desteklemediği için bu fonksiyonlar için özyinelemeli olmayan algoritmalar kullanılabileceği gibi CPU üzerinde özyinelemeli olarak da gerçekleştirilebilirler. Bu algoritmalar üzerinde yapılan çalışmalarda CPU üzerinde geliştirmenin daha verimli olacağı görüldüğü için bunlar CPU üzerinde özyinelemeli olarak gerçekleştirilmiştir.

Matris işlemleri altında gerçekleştirilen fonksiyonlar ve detayları şu şekildedir;

3.1.1.1 operator+ :

Matris toplama işlemi yapmaktadır. + Operatoru overload edilmiş olup tanımlı iki matris için $C_{i,j} = A_{i,j} + B_{i,j}$ işlemi yapılır.

3.1.1.2 operator+ (skalar) :

Matris toplama işlemi yapmaktadır. + Operatoru overload edilmiş olup tanımlı bir matris ve parametre olarak alınan skalar bir değerin toplamı için kullanılır. Matrisin tüm elemanları bu skalar değer ile toplanır. $C_{i,j} = A_{i,j} + k$ işlemi yapılır.

3.1.1.3 operator- :

Matris çıkarma işlemi yapmaktadır. - Operatoru overload edilmiş olup tanımlı iki matrisi için $C_{i,j} = A_{i,j} - B_{i,j}$ işlemi yapılır.

3.1.1.4 operator- (skalar) :

Matris çıkarma işlemi yapmaktadır. - Operatoru overload edilmiş olup tanımlı bir matristen parametre olarak alınan skalar bir değer çıkarılmaktadır. Matrisin tüm elemanlarından bu skalar değer ile çıkarılır. $C_{i,j} = A_{i,j} - k$ işlemi yapılır.

3.1.1.5 operator/ (skalar) :

Matris bölme işlemi yapmaktadır. / Operatoru overload edilmiş olup tanımlı bir matris ve parametre olarak alınan skalar bir değere bölüm için kullanılır. Matrisin tüm elemanları bu skalar değer ile bölünür. $C_{i,j} = A_{i,j}/k$ işlemi yapılır.

3.1.1.6 operator* :

Matris çarpma işlemi yapmaktadır. * Operatoru overload edilmiş olup tanımlı iki matris için $C_{i,j} = A_{i,j}xB_{i,j}$ işlemi yapılır.

3.1.1.7 operator*(skalar) :

Matris çarpma işlemi yapmaktadır. * Operatoru overload edilmiş olup tanımlı bir matrisin tüm elemanları parametre olarak alınan skalar bir değer ile çarpılır. $C_{i,j} = A_{i,j}xk$ işlemi yapılır.

3.1.1.8 operator| :

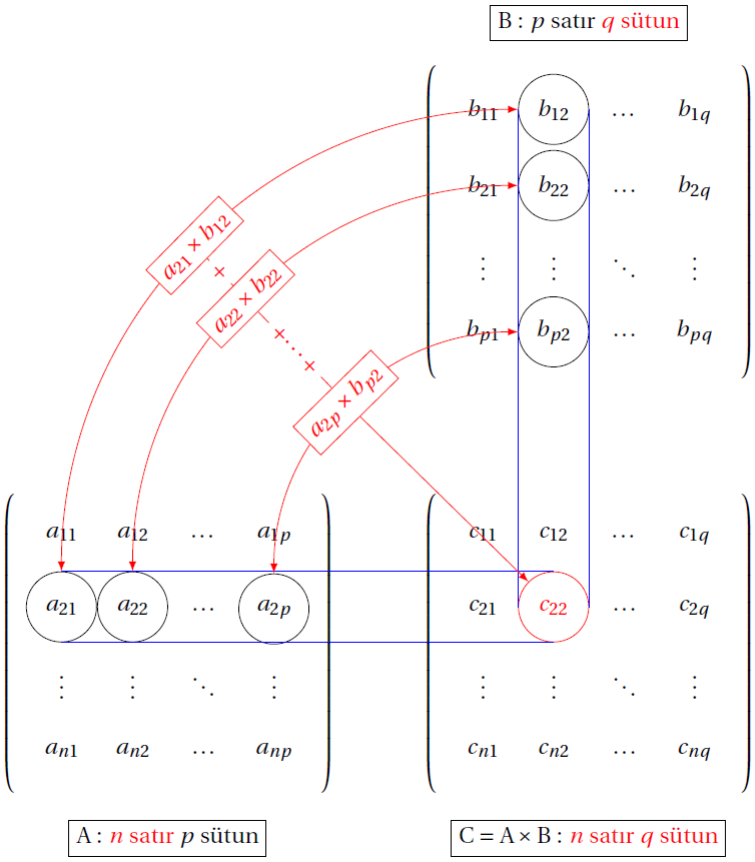
İki matrisin elemanlarını bit tabanlı mantıksal veya(or) işlemini uygulayarak yeni bir matrise yazan fonksiyondur. | Operatoru overload edilmiştir.

3.1.1.9 operator& :

İki matrisin elemanlarını bit tabanlı mantıksal ve(and) işlemini uygulayarak yeni bir matrise yazan fonksiyondur. & Operatoru overload edilmiştir.

3.1.1.10 matrixMul:

İki matrisin çarpım işlemi yapılır. Çarpım işlemi Şekil 3.1' de gösterildiği gibi yapılmaktadır.



Şekil 3.1: Matris Çarpım İşlemi

3.1.1.11 log :

Matris elemanlarının her birinin logaritmasını alarak, bu logaritma değerlerden oluşan yeni bir matris döndüren fonksiyondur. Logaritma hesaplanırken matristeki her bir elemanın mutlak değeri üzerinden hesaplanır. Logaritması hesaplanan matris elemanının 0 olması durumunda büyük bir negatif değer döndürülmektedir.

$$dst(I) = \begin{cases} log|src(I)|, & \text{if } src(I) \neq 0 \\ C, & \text{diğer durumlarda} \end{cases} \tag{3.1}$$

3.1.1.12 abs :

Matris elemanlarının her birinin mutlak değerini alarak, bu mutlak değerlerden oluşan yeni bir matris döndüren fonksiyondur.

3.1.1.13 min :

Matris elemanları arasında sayısal olarak en küçük olan elemanı döndüren fonksiyondur.

3.1.1.14 max :

Matris elemanları arasında sayısal olarak en büyük olan elemanı döndüren fonksiyondur.

3.1.1.15 inverse :

Parametre olarak verile bir matrisin tersini alan ve bunu başka bir matrise yazan fonksiyondur.

3.1.1.16 transpose :

Parametre olarak verile bir matrisin transpozunu alan ve bunu başka bir matrise yazan fonksiyondur.

$$dst(i, j) = src(j, i) \tag{3.2}$$

3.1.1.17 determinant :

Girdi olarak floating point bir kare matris alır ve bunun determinantını döndürür. Determinant hesaplanırken kullanılan fonksiyon 3.3' de gösterilmiştir. C, A matrisinin kofaktörünü, M ise minörünü ifade etmektedir.

$$det(A) = \sum_{j=1}^{n} A_{i,j} C_{i,j} = \sum_{j=1}^{n} A_{i,j} (-1)^{i+j} M_{i,j} \tag{3.3}$$

3.1.1.18 absDiff :

Parametre olarak alınan iki matrisin mutlak değer farkını alan ve yeni oluşan matrisi döndüren fonksiyondur.

3.1.1.19 absDiff (skalar) :

Parametre olarak alınan bir matris ve parametre olarak alınan skalar bir değerin mutlak farkını alan ve oluşan yeni matrisi döndüren fonksiyondur.

3.1.1.20 exp :

Parametre olarak alınan matrisde bulunan her bir elemanın sxponentini alan fonksiyondur.

$$dst(I) = e^{src(I)} \tag{3.4}$$

3.1.2 Filtreleme İşlemleri

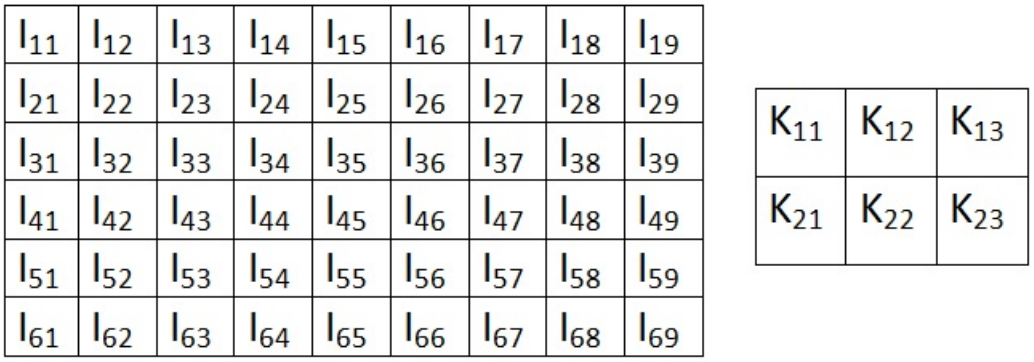
Filtreleme işlemleri görüntü üzerinde belirli ayrıntıların ayıklanması, bazı kısımlarının daha belirgin hale getirilmesi veya bazı öz-niteliklerin bulunması amacıyla

yapılan operasyonlardır. Bu operasyonlar çoğunlukla konvolüsyon tabanlı işlemleri içermektedir.

Filtreleme algoritmaları görüntü işleme uygulamalarının temelini oluşturduğu gibi OpenCL ile paralel hesaplama altyapısını oldukça uygundur. Bellek erişimlerinin diğer algoritmalara göre daha az olması ve matematiksel karmaşıklıkları filtreleme algoritmalarını OpenCL ile paralel hesaplama altyapısına oldukça uymaktadır. Bu kazanımları en üste çıkarmak ve OpenCL'in bizlere sunduğu paralel hesaplama altyapısını daha iyi kullanabilmek için imgeyi bloklar halinde işlemek ve bu blokları iş parçacıklarına paylaştırmak uygulanmıştır. Her bir iş parçacığı genel belleğe ulaşmak ve gerektiği durumlarda da piksel verisini yerel belleğe kopyalamakla sorumludur. Bu veri üzerinde uygulanan filtreleme işlemine uygun matematiksel işlem yapılmaktadır. Bu grup altında genellikle konvolüsyon tabanlı çeşitli filtreleme algoritmaları implement edilmiştir. Geliştirilen bu algoritmalar ve detay bilgileri şu şekildedir;

3.1.2.1 convolveImageWithFilter :

Kullanıcı tarafından tanımlanmış bir filtre ile görüntü üzerinde konvolüsyon işlemi yapan fonksiyondur. Konvolüsyon işlemi görüntü üzerinde bir maskenin sol üst köşeden başlanarak, maskenin merkezi her bir piksel üzerinden geçecek şekilde bütün resmin taranması işlemidir. Bu tarama işlemi sırasında maske içerisinde kalan her bir piksel maskenin katsayıları ile çarpılıp bu çarpımların toplamı, yeni resimde maskenin merkezinin geldiği konuma yazılır. Bu işlem aşağıdaki Şekil 3.2 de gösterildiği gibidir. Örnek görüntüdeki değerler formül 3.5 işlemine göre yapılır. Bu işlemin genelleştirilmiş gösterimi formül 3.6 'de gösterilmiştir. Konvolüsyon işleminde kullanılan matris, içerdiği değerlere göre resmi filtreleme, bulanıklaştırma ve kenar bulma gibi önemli sonuçlar bulunmasını sağlamaktadır.



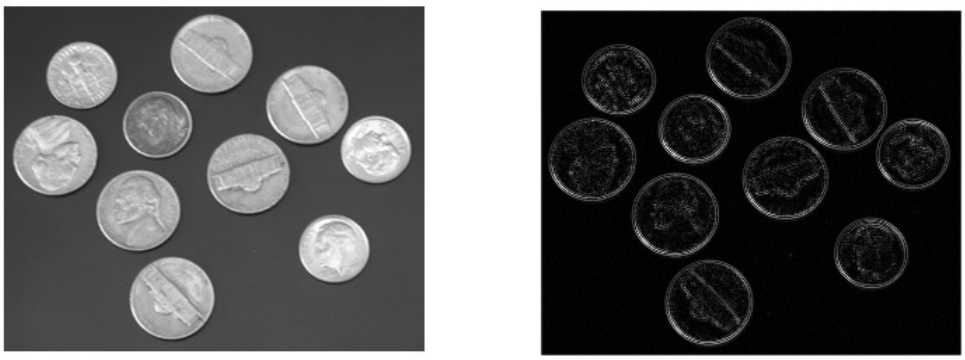
Şekil 3.2: Konvolüsyon Çalışma Örneği

$$O_{5,7} = I_{5,7}K_{1,1} + I_{5,8}K_{1,2} + I_{5,9}K_{1,3} + I_{6,7}K_{2,1} + I_{6,8}K_{2,2} + I_{6,9}K_{2,3} \tag{3.5}$$

$$O(i,j) = \sum_{k=1}^{m}\sum_{l=1}^{n} I(i+k-1, j+l-1)K(k,l)$$
$$i = 1...M-m+1$$
$$j = 1...N-n+1 \tag{3.6}$$

3.1.2.2 Laplacian:

Laplace filtresi basitçe bir resimdeki kenar hatlarını belirlemek için kullanılır. Burada kenar ile kastedilen objeleri genelde arka plandan ayıran keskin renk ayrılıklarıdır. Keskinleştirme filtresi (Sharpening Filter) ismi ile de anılan laplace filtresi çalışırken çeşitli maskeler kullanır.



Şekil 3.3: Laplace Uygulanmış Resim Örneği

Laplace operatörü, bir görüntünün I(x,y) 2. uzaysal türevinin 2 boyutlu bir izotropik ölçümüdür 3.7. Laplace operatörü, bir görüntüde hızlı yoğunluk değişimlerinin görüldüğü bölgeyi vurgular ve dolayısıyla, genellikle kenar algılama işlemleri için kullanılır. Laplace algoritması, genellikle, gürültü hassasiyetini azaltmak için Gauss düzleştirme filtresine yakın bir operatörle düzgünleştirilmiş bir görüntüye uygulanır. Görüntü üzerinde ikinci bir türev ölçümüne karşılık gelmesi nedeniyle, bu operatörler gürültüye oldukça duyarlıdır. Bunu önlemek için, Laplace filtresi uygulanmadan önce, görüntüye genellikle Gauss düzleştirmesi yapılır. Bu ön işleme adımı, farklılaşma adımından önceki yüksek frekanslı gürültüyü azaltır. Laplace operatörü, normalde bir gri düzey görüntüyü girdi olarak alır ve başka bir gri düzey görüntüyü çıktı olarak verir. Şekil 3.4' deki gibi bir maske kullanılmasıyla, Laplace operatörü, standart konvolüsyon yöntemleri kullanılarak hesaplanabilir.

$$L(x, y) = \frac{\delta^2 I}{\delta x^2} + \frac{\delta^2 I}{\delta y^2} \tag{3.7}$$

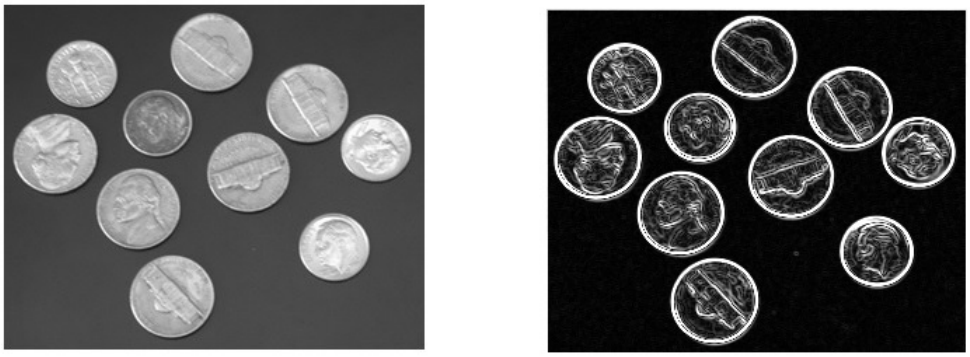
0	−1	0
−1	4	−1
0	−1	0

−1	−1	−1
−1	8	−1
−1	−1	−1

Şekil 3.4: Laplace Operatör Örnekleri

3.1.2.3 Sobel:

Sobel operatörü, görüntü üzerinde 2 boyutlu uzaysal gradyan ölçümü gerçekleştirir ve böylece, kenarlara karşılık gelen yoğunluk değişimi olan bölgelerini vurgular. Sobel operatörü, genellikle gri tonlamalı bir görüntünün her bir noktasındaki yaklaşık mutlak gradyan büyüklüğünü bulmak için kullanılır.



Şekil 3.5: Sobel Uygulanmış Resim Örneği

Sobel maskeleri, piksel değerlerine göre göre dikey ve yatay uzanan kenarları belirtmek için, bu birbirine dik yönlerin her biri için bir maske uygulanmak üzere tasarlanmıştır. Maskeler, her bir yön üzerinde gradyan bileşeninin ayrı ölçümlerini (bunlara Gx ve Gy denebilir) elde etmek için görüntü girdisine ayrı ayrı uygulanabilir. Daha sonra bunlar, her bir nokta üzerindeki gradyanın

mutlak büyüklüğünü ve söz konusu gradyanın yönünü bulmak için birleştirilebilir. Gradyan büyüklüğü şöyle elde edilir 3.8.

$$G = \sqrt{Gx^2 + Gy^2}$$
$$\Theta = arctan(Gx/Gy) \tag{3.8}$$

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

Gx

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

Gy

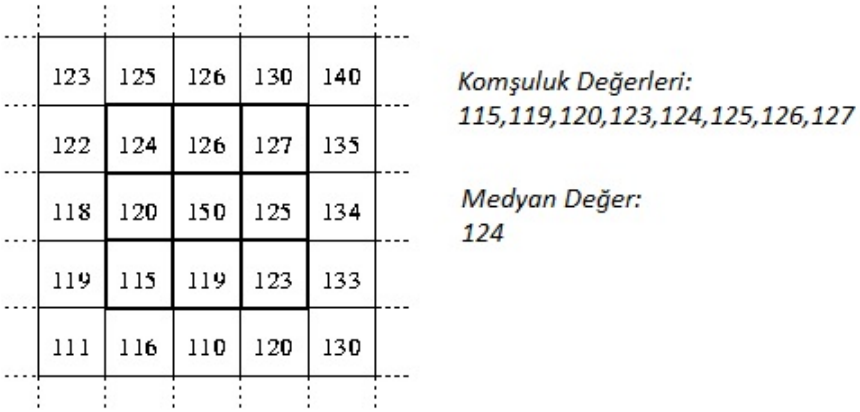
Şekil 3.6: Sobel Operatör Örnekleri

3.1.2.4 Median:

Medyan filtreleme görüntü üzerindeki gürültüyü azaltmak için kullanılan bir filtredir. Gürültülü görüntü üzerindeki her bir pikselin değeri o piksele komşu olan diğer piksellere ait değerlerin medyan değeri ile değiştirilir. Bu sayede gürültüyü yaratan piksel değeri o komşulukta bulunan medyan piksel değeri ile değiştirilmiş olur. Genellikle 3x3'lük maskeler kullanılırken görüntüdeki gürültü çeşidine göre maske büyüklüğü ve yapısı değiştirilebilir. Medyan filtreleme örnek bir görüntü üzerinde uygulanması Şekil 3.8' de gösterilmiştir.

Şekil 3.7: Medyan Uygulanmış Resim Örneği

Burada da görüleceği gibi uygulanan filtre her seferinde üzerinde çalışılan pikselin komşuluğunda bulunan değerleri okumaktadır. Bu durum OpenCL yapısına göre düşünüldüğünde her seferinde bellek erişimi diye düşünülebilir. Bu nedenle bu erişimleri daha hızlı yapabilmek için çeşitli optimizasyonlar yapılabilir. Bir sonraki bölümde daha detaylı anlatılacak olan bu iyileştirmelere bu filtre için uygun olan örnek lokal bellek kullanımı olacaktır. Bu sayede bellek erişimleri daha kısa sürede yapılabilecek ve performans kazanımı sağlanacaktır.



Şekil 3.8: Medyan Değerinin Bulunması

3.1.2.5 Gaussian:

Gaussian filtreleme aynı zamanda bir fourier dönüşümüdür. Sadece Gauss çan eğrisi formu ile Gaussian filtreleme olarak adlandırılmaktadır. Gauss filtre ile sonsuz bir transfer fonksiyonuna karşılık mekansal alanda sonlu bir pencerede (tarama penceresi) filtreleme yapılabilmektedir. Bu da filtrelemenin temel problemini daha kolay çözülebilir hale getirir[7]. Gauss düzleştirme operatörü, görüntüleri 'bulanıklaştırmak', ayrıntıları ve gürültüyü ortadan kaldırmak için kullanılan 2 boyutlu bir konvolüsyon operatörüdür. Bu bakımdan nbox filtresine benzese de, (çan şeklindeki) Gauss eğrisinin şeklini temsil eden farklı bir maske kullanır. Bu maske, ayrıntıları aşağıda verilen bazı özelliklere sahiptir. Gauss düzleştirmesinin arkasında yatan fikir, bu 2 boyutlu dağıtımı bir "nokta dağılması" fonksiyonu olarak kullanmaktır ve bu da konvolüsyonla sağlanır(Formül 3.9).

Görüntünün bir araya gelmiş ayrık pikseller olarak depolanması nedeniyle, konvolüsyon gerçekleştirmeden önce, Gauss fonksiyonuna ayrık bir yaklaşımlama üretmemiz gerekir. Teoride, Gauss dağılımı her yerde sıfırdan farklıdır ve bu durum, sonsuz büyüklükte bir maske gerektirecektir fakat pratikte çeşitli yakınsamalar mevcuttur. Şekil 3.9 Gauss fonksiyonunu $\sigma = 1.0$ ile yaklaşımlayan uygun bir tamsayı değerli maske görülmektedir. Gauss fonksiyonuna yaklaşımlamanın yapılması için maske değerlerinin nasıl seçileceği açıkça gösterilmemiştir. Gauss fonksiyonunun değerinin, maske üzerindeki bir pikselin merkezinde kullanılması mümkündür fakat Gauss fonksiyonunun değeri piksel boyunca doğrusal olmayan bir biçimde farklılık gösterdiğinden, bunu yapmak doğru sonuçlar vermeyecektir. Gauss fonksiyonunun değerini (Gauss fonksiyonunu 0. 001 artırıp toplamak suretiyle) tüm piksel üzerinde tamamladık. Diziyi, köşelerin değeri 1 olacak biçimde yeniden boyutlandırdık. İşlemin sonunda, maskedeki tüm değerlerin toplamı 273 çıktı ve bu şekilde $\sigma = 1.0$ değerli maskeyi üretmiş olduk.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3.9}$$

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

Şekil 3.9: Gauss Operatörü($\sigma = 1.0$)

Uygun bir çekirdeğin hesaplanmasının ardından, standart konvolüsyon yöntemleri kullanılarak Gauss düzleştirmesi gerçekleştirilebilir. Yukarıda geçen 2 boyutlu izotropik Gauss fonksiyonunun denklemi x ve y bileşenlerine ayrıştırılabildiğinden, konvolüsyon oldukça hızlı bir biçimde gerçekleştirilebilir. Böylece, öncelikle x doğrultusunda 1 boyutlu Gauss fonksiyonuyla, daha sonra y doğrultusunda başka bir 1 boyutlu Gauss fonksiyonuyla konvolüsyon uygulayarak, 2 boyutlu konvolüsyon işlemini gerçekleştirmek mümkündür. (Gauss operatörü, bu şekilde ayrıştırılabilen, tamamen dairesel simetrik tek operatördür.) Şekil 3.10'te, (sınır çevresindeki piksellerin değeri çoğunlukla 0 olduğundan, buradan bir piksel sırasını 273 ile boyutlandırıp, yuvarlayarak budadıktan sonra) Şekil 3.9'te gösterilen tam çekirdeği elde etmek için kullanılacak 1 boyutlu x bileşeni görülmektedir (Bu işlem, 7x7 matrisi, yukarıda görülen 5x5 matrise düşürür.). y bileşeni tamamen aynı fakat dikey doğrultudadır.

.006	.061	.242	.383	.242	.061	.006

Şekil 3.10: Gauss Operatörünü Elde Etmek İçin Kullanılan x Bileşeni

3.1.2.6 Box ve Normalized Box(nbox):

nbox filtreleme, görüntüleri düzleştirmenin, yani bir piksel ile bir sonraki arasındaki yoğunluk farkını azaltmanın basit, sezgisel ve uygulaması kolay bir yöntemidir. Bu yöntem, genellikle görüntülerdeki gürültüyü azaltmak için kullanılır. nbox filtrelemenin arkasındaki fikir, bir görüntüdeki her bir pikselin değerini, kendisi de dahil olmak üzere çevresindeki piksellerin ortalama değeriyle değiştirmektir. Bu işlem, çevresindeki değerleri yansıtmayan piksel değerlerinin ortadan kaldırılmasına yönelik bir etki doğurur. Sıklıkla Şekil 3.11'de gösterildiği gibi 3x3 maskeler kullanılsa da, daha ağır düzleştirme işlemleri için daha büyük çekirdekler de (örneğin, 5x5 kareler) kullanılabilir. (Büyük bir maskede tek bir uygulamada elde edilen etkiyle birebir aynı olmasa da benzer bir etki, küçük bir maskenin birden fazla kez uygulanmasıyla elde edilebilir.

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

Şekil 3.11: 3 x 3 NBOX Operatörü

3.1.2.7 edgeCanny :

Canny operatörü optimal bir kenar algılayıcı olarak tasarlanmıştır. Bu operatör, gri tonlamalı görüntüleri girdi olarak alır ve izlenen yoğunluk kusurlarını gösteren bir görüntü çıkarır.

Şekil 3.12: Canny Uygulanmış Resim Örneği

Canny operatörü, çok aşamalı bir süreç ile çalışır. Öncelikle, Gauss filtre ile görüntü düzleştirilir. Daha sonra 2 boyutlu bir ilk türev operatörü, görüntünün ilk uzaysal türevlerinin yüksek olduğu bölgelerini vurgulamak üzere, düzleştirilmiş görüntüye uygulanır. Kenarlar, gradyan büyüklüğü görüntüsünde çizgilerin oluşmasını sağlar(Sobel filtreleme). Algoritma, daha sonra bu çizgilerin üzerinden giderek, çıktı görüntüsünde ince bir çizgi ortaya çıkarmak için, non-maximal sıkıştırma adı verilen bir işlemle çizgilerin üzerinde olmayan tüm pikselleri sıfır olarak değiştirir. İzleme işlemi, iki eşikle yönetilen bir histerezis ortaya koyar(T1> T2 olmak üzere T1 ve T2). İzleme işlemi, ancak çizgi üzerinde T1'den yüksek bir noktada başlayabilir. İzleme, daha sonra bu noktadan çıkarak, çizginin yüksekliği T2'nin altına düşene dek her iki doğrultuda da ilerler. Histerezis, gürültülü kenarların birden fazla kenar parçasına bölünmemesine yardımcı olur.

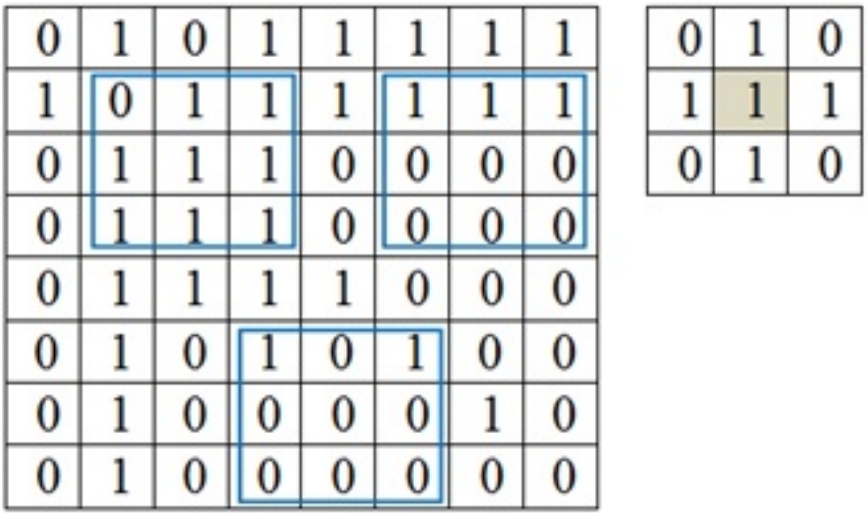
Genel olarak, iyi bir sonuç elde etmek için, üst izleme eşiği oldukça yüksek, alt

izleme eşiği ise oldukça düşük olarak ayarlanabilir. Alt eşiğin çok yüksek ayarlanması, gürültülü kenarlarda kopmalara yol açar. Üst eşiğin çok düşük ayarlanması ise, görüntü çıktısında görülen yanlış ve istenmeyen kenar bölünmelerini artırır. Temel Canny operatörüyle ilgili bir sorun, gradyan büyüklüğü görüntüsünde üç kenarın birleştiği Y birleşim noktalarıyla ilgilidir. Bu tür birleşim noktaları, bir kenarın başka bir nesne tarafından kısmen kesildiği noktalarda gerçekleşir. İzleyici, çizgilerin ikisini tek bir çizgi parçası olarak, üçüncü çizgiyi ise bu çizgi parçasına yaklaşan ama temas etmeyen bir çizgi olarak değerlendirilebilir.

Şekil 3.13: Canny Kenar Tanıma Algoritma Adımları

3.1.3 Morfolojik Operasyonlar

Morfolojik işlemler görüntü filtrelemenin bir alt kümesi olarak da adlandırılabilir. Görüntü filtreleme maske aralığındaki değerleri çarparak topladıktan sonar ilgili piksel değerini bulurken morfolojik operasyonda gezdirilen maskenin(structuring element) fit, hit ve miss gibi sonuçlarına göre ilgili piksel hesaplanır 3.14. Bu gruptaki fonksiyonlar özelikle medikal görüntü işleme ve robotik görüntü ile ilgili uygulamalarda sıklıkla kullanılmaktadır.



Şekil 3.14: Görüntü Üzerinde "cross" Yapısal Eleman ile "Hit" ve "Miss"

3.1.3.1 createStrEl :

Yapısal Eleman: Yapısal eleman olarak isimlendirilen yapı, imge üzerinde yapacağımız işleme ve yapmak istediğimiz uygulamaya göre istenilen boyutlarda ve istenilen şekilde hazırlanmış küçük ikilik bir imgedir. Yapısal eleman farklı geometrik şekillerden herhangi biri olabilir. En çok kullanılanları kare, dikdörtgen ve dairedir .

3.1.3.2 createLineStrEl :

Çizgi şeklinde yapısal eleman oluşturan fonksiyondur.

3.1.3.3 createRectangleStrEl :

Dikdörtgen şeklinde yapısal eleman oluşturan fonksiyondur.

3.1.3.4 createDiscStrEl :

, Disk şeklinde yapısal eleman oluşturan fonksiyondur.

3.1.3.5 createDiamondStrEl :

Daire şeklinde yapısal eleman oluşturan fonksiyondur.

3.1.3.6 createOctagonStrEl :

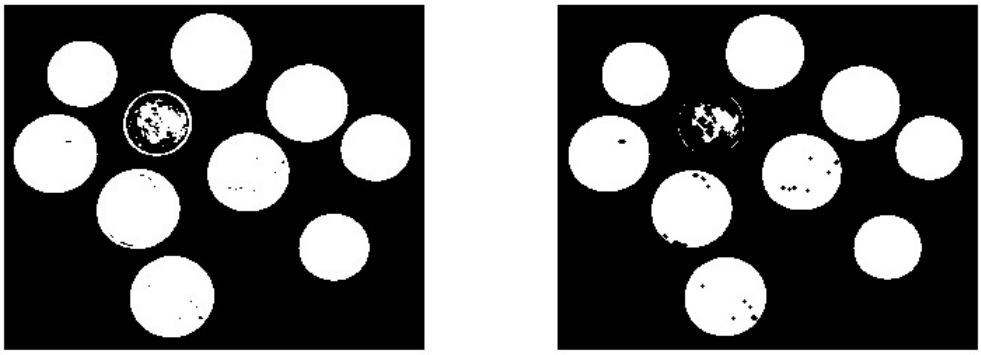
Oktagon şeklinde yapısal eleman oluşturan fonksiyondur.

3.1.3.7 createSquareStrEl :

Kare şeklinde yapısal eleman oluşturan fonksiyondur.

3.1.3.8 erodeImage :

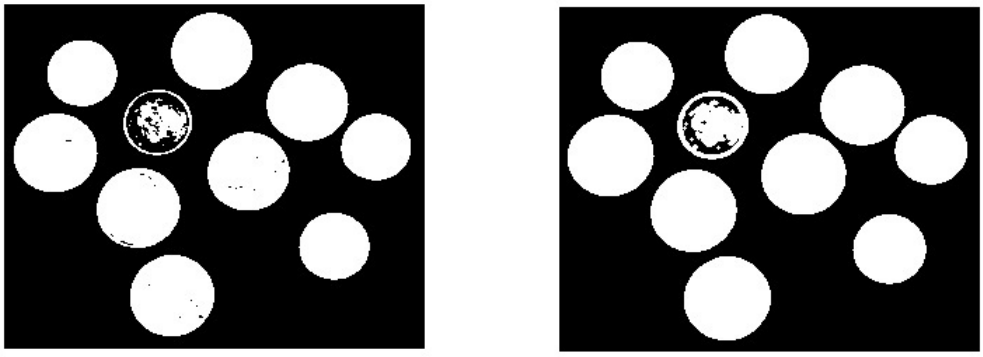
İkili imgedeki nesneyi küçültmeye ya da inceltmeye yarayan morfolojik işlemdir. İmge içerisindeki nesneler ufalır, delik varsa genişler, bağlı nesneler ayrılma eğilimi gösterir. Şekil 3.15'de ikili aşınma işlemi uygulanmış imgeler gösterilmektedir.



Şekil 3.15: Aşınma Uygulanmış Resim Örneği

3.1.3.9 dilateImage:

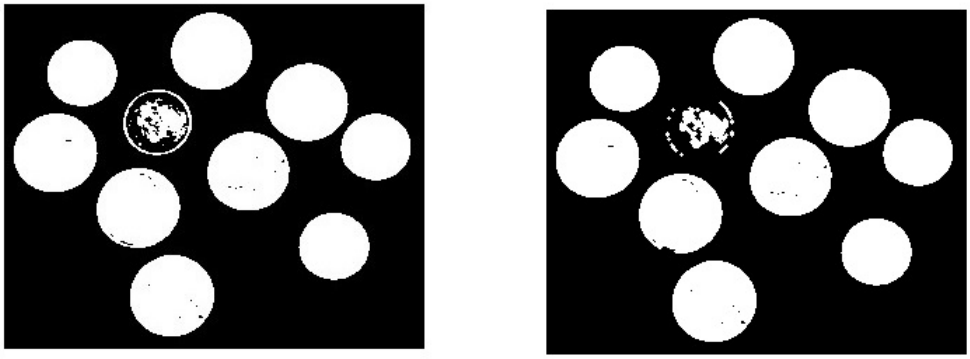
İkili imgedeki nesneyi büyütmeye ya da kalınlaştırmaya yarayan morfolojik işlemdir. Sayısal bir imgeyi genişletmek imgeyi yapısal elemanla kesiştiği bölümler kadar büyütmektir. İşlenecek imgenin her bir pikseli, yapısal elemanın merkez noktasına oturtularak genleşme işlemi yapılmaktadır. Kalınlaştırma işleminin nasıl yapılacağını yapısal eleman belirler. Genleşme işlemi uygulanmış bir imgede, imge içerisindeki deliklerin ve boşlukların doldurulması ve köşe noktasının yumuşaması gözlenir. Şekil 3.16'de ikili genleşme işlemi uygulanmış imgeler gösterilmektedir.



Şekil 3.16: Genleşme Uygulanmış Resim Örneği

3.1.3.10 imageApplyOpening :

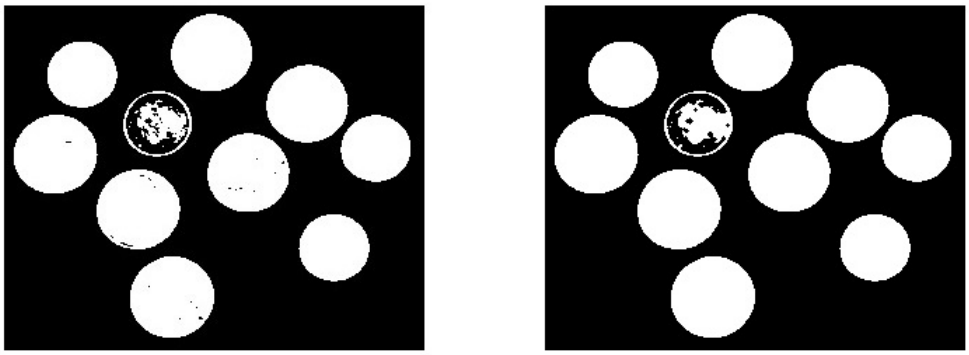
İmge üzerinde aşınma işleminin hemen ardından genleşme işlenmesi uygulanması sonucu açma işlemi elde edilir. İmge içerisindeki nesneler ve nesneler arasındaki boşluklar yapısal elemanın büyüklüğüne göre temizlenir. İmge üzerinde kalan nesneler orijinal imgedeki şekillerinden biraz daha küçük hale gelir. Açma işlemi ile birbirine yakın iki nesne imgede fazla değişime sebebiyet vermeden ayrılmış olurlar . Şekil 3.17' de açma işlemi uygulanmış imge gösterilmektedir.



Şekil 3.17: Açma Uygulanmış Resim Örneği

3.1.3.11 imageApplyClosing :

İmge üzerinde genleşme işleminin hemen ardından aşınma işleminin uygulanması sonucu kapama işlemi elde edilir. Dolayısıyla birbirine yakın iki nesne imgede fazla değişiklik yapılmadan birbirine bağlanmış olur. Şekil 3.18' de kapama işlemi uygulanmış imge gösterilmektedir. Kapama işlemi sonunda imge içerisindeki noktalar birbirlerini kapatırlar, imgedeki ana hatlar daha da dolgunlaşır. Genleşme işlemine benzer bir şekilde kapama işleminde de birbirine yakın olan noktalar arasındaki boşluklar dolar ve noktalar birleşir. İmge üzerinde kalan nesneler, orjinal imgedeki şekillerine bürünürler.



Şekil 3.18: Kapama Uygulanmış Resim Örneği

3.1.4 Dönüşüm İşlemleri

Bu bölümde görüntü üzerinde uygulanan dönüşüm fonksiyonları bulunmaktadır.

3.1.4.1 create2DRotMatrix :

Döndürme için kullanılacak olan 2 boyutlu matrisi oluşturur.

3.1.4.2 getAffineTransform :

Alınan 3 referans noktası yardımı ile afin transformasyon matrisini hesaplayan fonksiyondur.

3.1.4.3 warpAffine:

Birçok görüntüleme sisteminde, algılanan görüntüler, kamera(ların) sahneye göre konumunun, sahnenin geometrisinin görünen boyutlarını değiştirdiği perspektif düzensizliklerinin ortaya çıkardığı geometrik bozulmalara maruz kalmaktadır. Tek tip bozulmuş bir görüntüye afin dönüşüm uygulanması, ölçümleri ideal koordinatlardan gerçekte kullanılan koordinatlara dönüştürerek, bir dizi perspektif bozukluğunu düzeltebilir. (Örneğin bu işlem, geometrisi doğru yer haritalarının elde edilmek istendiği uydu görüntülemelerinde kullanışlı bir yöntemdir.) Afin dönüşümün genel formülü şu şekildedir.

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = A \times \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + B \tag{3.10}$$

Örnek olarak A ve B'yi matris olarak tanımlarsak;

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{matrix} b_1 \\ b_2 \end{matrix} \tag{3.11}$$

Afin dönüşümü, koordinat dönüşümlerine dayalı 3 temel işlem olan öteleme (translation), döndürme (rotation) ve ölçekleme (scaling) işlemlerinin genel formudur.

- Öteleme:
  Öteleme uygulandığı nesne, eski konumundan belirli uzaklıktaki başka bir konuma taşınmaktadır. Önceki bölümde anlatılan afin dönüşüm formülünde A ve B matrislerini öteleme yapmak amacıyla tanımlarsak B matrisinin elemanları olan b_1 ve b_2 kadar ötelemiş oluruz.

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, B = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{3.12}$$

- Döndürme:
  Döndürme bir noktanın çembersel bir rota üzerinde konum değiştirmesidir. Önceki bölümde anlatılan afin dönüşüm formülünde A ve B matrislerini

öteleme yapmak amacıyla tanımlarsak A matrisine göre θ kadar döndürmüş oluruz.

$$A = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}, B = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{3.13}$$

- Ölçekleme :
  Ölçeklendirme dönüşümü ise görüntünün büyüklüğünü değiştirir. Yani nesneyi büyütmek veya küçültmek için kullanılabilir. Ölçeklendirme dönüşümü nesneyi tanımlayan koordinatların ölçeklendirme faktörleri ile çarpılması yoluyla gerçekleştirilir. Önceki bölümde anlatılan afin dönüşüm formülünde A ve B matrislerini ölçeklendirme yapmak amacıyla tanımlarsak A matrisine göre $a_{11}, a22$ değerlerine göre ölçeklemiş oluruz.

$$A = \begin{bmatrix} a_{11} & 0 \\ 0 & a_{22} \end{bmatrix}, B = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{3.14}$$

3.1.4.4 mirror :

Görüntüyü parametre olarak alınan bir düzlemde aynalayarak bu aynalanmış görüntüyü döndürür.

3.1.4.5 resizeBilinear:

Bilinear interpolasyon yöntemi ile örneklemede, orijinal görüntüden alınan 4 pikselin ağırlıklı ortalaması yeni piksele atanır. Bu yöntemde, düzeltilmiş görüntü tamamen yeni değerleri alır.

3.1.4.6 resizeBicubic :

Kübik Eğrilik yöntemiyle örneklemede, orijinal görüntüden alınan 16 piksel bloğunun ağırlıklı ortalaması hesaplanarak düzeltilmiş görüntü pikseline atanır. Kübik Eğrilik yöntemde, Bilinear interpolasyon yönteminde olduğu gibi yeni piksel

değerleri üretilmektedir. Bu iki metodun kullanımı ile elde edilen görüntüler, kullanıcıya daha keskin bir görünüş izlenimi verirler.

3.1.4.7 resizeNearestNeighbour :

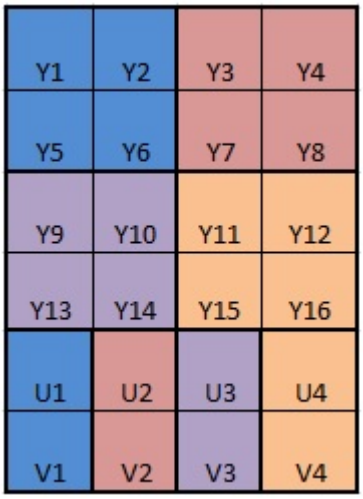
En Yakın Komşuluk yöntemi ile örneklemede, orijinal görüntüden alınan pikseller sayısal olarak düzeltilmiş görüntüdeki en yakın piksele atanır. Bu yöntemde orijinal değerler en az değişikliğe uğramakta bununla birlikte, bazı piksel değerleri çift olarak ortaya çıkmakta bazıları ise kaybolmaktadır.

3.1.5 Renk Kümesi Dönüşüm İşlemleri

Bu bölümde çeşitli modeller arası renk dönüşümleri yapmaya yarayan fonksiyonlar bulunmaktadır. Bir renk modelinin asıl amacı, genellikle kabul gördüğü biçimde bir takım standartlar ile renk tanımlamalarını kolaylaştırmaktır. Sayısal görüntü işlemede RGB, YUV,CMY,CMYK gibi adlandırılan çeşitli modeller bulunmaktadır. Bu bölümde bu modeller arası dönüşüm için kullanılan ve TRABZ-10 görüntü işleme kütüphanesinde gerçekşetirilen detayları algoritmaların verilecektir. Renk dönüşümü fonksiyonları veri bağımlılığı bulundurmayan algoritmik yapılara sahip olduğu için SIMD yapısına uygundur. Dolayısıyla kolayca paralelleştirilebilir ve performans kazanımları sağlanır. Ek olarak bazı algoritmalar farklı renk kümeleri üzerinde daha iyi performans verebilmektedirler, bu nedenle renk kümesi dönüşümüne ihtiyaç duyulur. Bu grupta geliştirilen fonksiyonlar şu şekildedir.

3.1.5.1 convertRGBtoYUV420 :

Görüntüyü RGB renk uzayı formatından YUV420 formatına dönüştürür. Örnek bir YUV 420 görüntüsü Şekil 3.19' de gösterilmiştir. Bu dönüşüm formül 3.15' e göre yapılır.



Şekil 3.19: YUV420 Yapısı

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 66 & 129 & 25 \\ -38 & -74 & 112 \\ 112 & -94 & -18 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$
$$\begin{aligned} Y &= (Y + 128) >> 8 \\ U &= (U + 128) >> 8 \\ V &= (V + 128) >> 8 \\ Y &= Y + 16 \\ U &= U + 128 \\ V &= V + 128 \end{aligned} \tag{3.15}$$

3.1.5.2 convertRGBtoYUV422 :

Görüntüyü RGB renk uzayı formatından YUV422 formatına dönüştürür. Bu dönüşüm formül 3.15' e göre yapılır.

3.1.5.3 convertRGBtoGRAY :

Görüntüyü RGB renk uzayı formatından gri ölçekli formata dönüştürür. Gri ölçekli format tek bir kanala sahip olduğu için ilgili kanallar eklenebilir veya çıkarılabilir. Bu dönüşüm formül 3.16' e göre yapılır.

$$x = 0.299R + 0.587G + 0.114B \tag{3.16}$$

3.1.5.4 convertYUV420toRGB :

Görüntüyü RGB renk uzayı formatından YUV422 formatına dönüştürür. Bu dönüşüm formül 3.17' e göre yapılır.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.28033 \\ 1 & -0.21482 & -0.38059 \\ 1 & 2.12798 & 0 \end{bmatrix} \times \begin{bmatrix} Y \\ U \\ V \end{bmatrix} \tag{3.17}$$

3.1.5.5 convertYUV422toRGB :

Görüntüyü RGB renk uzayı formatından YUV422 formatına dönüştürür. Bu dönüşüm formül 3.17' e göre yapılır.

3.1.6 Histogram İşlemleri

3.1.6.1 Histogram :

Görüntü işleme bağlamında, bir görüntünün histogramı normalde piksel yoğunluk değerlerinin histogramı anlamına gelir. Söz konusu histogram, görüntüde bulunan her farklı yoğunluk değerindeki piksel sayısını gösteren bir grafiktir. Gri tonlamalı 8 bit bir görüntü için 256 farklı yoğunluk olasılığı mevcuttur ve böylece histogram, bu gri tonlamalı görüntü değerlerinin arasında piksel dağılımını gösteren 256 adet sayıyı grafik üzerinde gösterecektir. Kırmızı, yeşil ve mavi kanallarının tekil histogramlarını alarak veya üç eksenden her birinin kırmızı, mavi ve yeşil kanallara karşılık geldiği ve her noktadaki parlaklığın piksel sayısını ifade ettiği 3 boyutlu histogram yoluyla, renkli görüntülerin histogramını almak da mümkündür.

3.1.6.2 calculateHistogram :

Girdi olarak alınan görüntünün histogramını döndüren fonksiyondur.

3.1.6.3 histogramEqualization :

Histogram eşitleme bir görüntüyü, yoğunluk histogramı istenen şekle gelecek biçimde değiştirerek, söz konusu görüntünün dinamik aralığı ve karşıtlığını düzenlemek için gelişmiş bir yöntem sunar. Karşıtlığı artırmanın aksine, histogram modelleme operatörleri, görüntü girdileri ve çıktıları arasındaki piksel yoğunluk değerlerini birbirine eşlemek için doğrusal olmayan ve tekdüze olmayan aktarım fonksiyonları kullanabilir. Histogram eşitleme, görüntü girdisindeki piksellerin yoğunluk değerlerini, görüntü çıktısında tek tip bir yoğunluk dağılımı (düz histogram) olacak biçimde yeniden atayan tek tip, doğrusal olmayan bir eşlem kullanır. Bu teknik, (ayrıntıların pekiştirilmesinde etkili olduğundan) görüntü

karşılaştırma işlemlerinde ve örneğin, bir dijitalleştirici veya görüntüleme sistemi ile ortaya koyulan doğrusal olmayan etkilerin düzeltilmesinde kullanılabilir.

3.1.6.4 histogramThreshold :

Parametre olarak alınan histogramdaki değerler üzerine eşik değeri uygulanır. Bu eşik üzerindeki değerler histogramdan silinir.

3.1.6.5 histogramNormalization :

Histogramı parametre olarak alınan değere yakınlaştıran fonksiyondur. Normalleştirme işleme histogramı belli bir değerle çarparak veya bölerek yapılabilir.

3.1.6.6 getMinMaxHistogram :

Histogramdaki en küçük ve en büyük değerlerin histogram üzerindeki yerlerini döndürür. Birden çok aynı değer olması durumunda sırasal olarak öncelikli olanı döndürmektedir.

4. GELİŞTİRME ve İYİLEŞTİRME YÖNTEMLERİ

4.1 Giriş

Bu bölümde OpenCL kernellarını geliştirirken uyguladığımız ve genel olarak bir OpenCL uygulaması geliştirirken yapılan geliştirme ve iyileştirme yöntemlerinden bahsedilecektir. Bu yöntemler uygulanırken öncelikli olarak fonksiyonel olarak doğru çalışan kernellar geliştirilmiş ardından aşağıda belirtilen iyileştirme yöntemlerinden, iyileştirme yapılacak kernelın yapısına ve üzerinde çalıştığımız gömülü platformun mimarisine uygun olanlar gerçekleştirilmiştir. Bu bölümde anlatılan iyileştirme yöntemleri örnek bir OpenCL kernelı üzerinden detaylandırılmıştır. Örnek olarak medyan filtre üzerinden detaylı açıklamalar yapılacaktır. Bölüm 3'de algoritmik yapısı anlatılan medyan filtresinin basitçe implement edilmiş kernelı Şekil 4.1' de gösterilmiştir. Kernel üzerinde görüleceği üzere öncelikle görüntü üzerinde ilgili veriye ulaşmak için indeks hesaplanmış, bu indeks değerinin kenar değerlerine ulaşmaması için gerekli koşullar düzenlenmiştir. Ardından 8 adet komşuluk değeri okunmuş ve ilgili değişkenlere yazılmış ve bu değerler üzerinden sıralama işlemi gerçekleştirilmiştir. Son olarak ise ilgili değere sıralanan değerler tarafından çıkarılan medyan değeri ilgili indeks ile piksel değeri olarak yazılmıştır.

```
void swap( uint*a, uint*b ) {
    uint tmp= *b;
    *b = *a;
    *a = tmp;
}
void sort3( uint* a, uint* b, uint* c ){
    if( *a > *b )
        swap( a, b );
    if( *b > *c )
        swap( b, c );
    if( *a > *b )
        swap( a, b );
}
__kernel void medianfilter( __global uint* id, __global uint* od, int width, int h, int r ){
    const int posx= get_global_id(0);
    const int posy = get_global_id(1);
    const int idx= posy*width + posx;
    if( posx== 0|| posy == 0|| posx== width-1|| posy == h-1){
        od[ idx] = id[ idx];
    }else{
        uint row00, row01, row02, row10, row11, row12, row20, row21, row22;
        row00 = id[ idx-1-width ]; row01 = id[ idx-width ]; row02 = id[ idx+ 1-width ];
        row10 = id[ idx-1]; row11 = id[ idx]; row12 = id[ idx+ 1];
        row20 = id[ idx-1+ width ]; row21 = id[ idx+ width ]; row22 = id[ idx+ 1+ width ];
        // sort the rows
        sort3( &(row00), &(row01), &(row02) );
        sort3( &(row10), &(row11), &(row12) );
        sort3( &(row20), &(row21), &(row22) );
        // sort the cols
        sort3( &(row00), &(row10), &(row20) );
        sort3( &(row01), &(row11), &(row21) );
        sort3( &(row02), &(row12), &(row22) );
        // sort the diagonal
        sort3( &(row00), &(row11), &(row22) );
        // median is the themiddle value of the diagonal
        od[ idx] = row11;
    }
}
```

Şekil 4.1: Medyan Filtre OpenCL Kernel

4.1.1 Vektör Operasyonlarını Kullanarak Her İş Parçacığına Daha Çok Veri

Vektör veri tiplerini kullanarak iş parçacıkları daha verimli kullanılabilir. Her iş parçacığı daha çok veri işleyebileceğinden iş parçacığı başına düşen iş miktarı artırılmış olur. char, uchar, short, ushort, int, uint, float, long, ve ulong vektör veri tipleri OpenCL tarafından desteklenmektedir. Desteklenen veri büyüklükleri

ise 2,3,4,8 ve 16'dır. Buna ek olarak vektör veri tipleri double ve half olarak da tanımlanabilmektedir. Bu tanımlamalar device' a bağlı olup device "double" veya "half precision"' ı destekliyorsa mümkün olabilmektedir. Vektör veri tiplerini kullanmanın getireceği kazançlar üzerinde çalışılan device'ın mimarisine bağlı kazanımlardır. Bazı mimarilerde vektör veri tiplerini kullanmak "bank conflict"'leri azaltabilirkern bazı mimarilerde ise "bank conflict"'lere neden olabilmektedir. Bunun dışında vektör veri tiplerini kullanmak genel olarak hesaplama süresini azaltarak kazanımlar getirmektedir.

Önceki bölümde verilen OpenCL kernelının vektör veri tipleri kullanılarak gerçeklenmesi sonucu yeni kernel Şekil 4.2' deki gibidir. Vektör veri tiplerinin kullanılması sonucu okuma yazma işlemleri 128 bit olarak yapılmış ve her iş parçacığı birden çok değer hesaplayarak döndürebilmiştir. Bunun sonucunda ilk kernela göre %20'lik bir performans kazanımı görülmüştür.

```
__kernel void medianfilter_x4( __global uint* id, __global uint* od, int width, int h, int r ){
    const int posx= get_global_id(0); // global width is 1/4 image width
    const int posy = get_global_id(1); // global height is image height
    const int width_d4 = width >> 2; // divide width by 4
    const int idx_4 = posx*(width_d4) + posy;

    uint4 left0, right0, left1, right1, left2, right2, output;
    // ignoring edge cases for simplicity
    left0 = ((__globaluint4*)id)[ idx_4 -width_d4 ];
    right0 = ((__globaluint4*)id)[ idx_4 -width_d4 + 1];
    left1 = ((__globaluint4*)id)[ idx_4 ];
    right1 = ((__globaluint4*)id)[ idx_4 + 1];
    left2 = ((__globaluint4*)id)[ idx_4 + width_d4 ];
    right2 = ((__globaluint4*)id)[ idx_4 + width_d4 + 1];

    // now compute four median values
    output.x= find_median( left0.x, left0.y, left0.z,
    left1.x, left1.y, left1.z, left2.x, left2.y, left2.z );
    output.y= find_median( left0.y, left0.z, left0.w,
    left1.y, left1.z, left1.w, left2.y, left2.z, left2.w );
    output.z= find_median( left0.z, left0.w, right0.x,
    left1.z, left1.w, right1.x, left2.z, left2.w, right2.x );
    output.w= find_median( left0.w, right0.x, right0.y,
    left1.w, right1.x, right1.y, left2.w, right2.x, right2.y );

    ((__globaluint4*)od)[ idx_4 ] = output;
}
```

Şekil 4.2: Medyan Filtre OpenCL Kernel 2

4.1.2 Yerel(Local) Bellek Kullanımı

OpenCL'de yerel bellek, her bir işlemci üzerinde yer alan bir bellek türüdür ve bu belleğin üzerindeki veriler, yalnızca aynı iş grubu üzerindeki iş parçacıkları ile paylaşılır. Yerel bellek ve yazmaçlar arasında veri aktarımı, genel bellek ve yazmaçlar arasında veri aktarımından daha hızlıdır. Veriler host üzerinde yazılamaz ve okunamaz. İşlemlerin yalnızca device üzerinde gerçekleştirilmesine izin verilir. Yerel bellek üzerindeki veriler ancak aynı iş grubu tarafından paylaşılabilir. Genel bellek üzerindeki bir elemana birden fazla erişmeniz gerektiğinde, bu elemanı bir kez yerel belleğe kopyalamak ve birden çok kez yerel bellekten erişmek, bant genişliği üzerinde hızlanmanızı sağlar. Bu koşullarda, yerel bellek oldukça kullanışlıdır.

Önceki bölümde verilen OpenCL kernelının yerel bellek kullanılarak gerçeklenmesi sonucu yeni kernel Şekil 4.3' deki gibidir. Bu kernelda ise 8x8 lik bir yerel bellek kullanılmış olup ilgli piksele ait 9x9 komşuluk değerleri bu belleğe kopyalanmıştır. Daha sonra okuma işlemleri genel bellek yerine, yerel bellek kullanılarak yapılmıştır. Diğer kernellara göre daha karmaşık bir indeksleme gerektiren bu kernel sonucunda ilk kernela göre %35'lik bir performans kazanımı görülmüştür.

```
__kernel void medianFilter(__global uchar *src,__global uchar *dst,const int cols,const int rows,const int depth){
    int srcOffset = 0;
    int dstOffset = 0;
    int srcStep = cols*depth;
    int dstStep = rows*depth;
    __local uchar data[9][9];
    __global uchar* source=src + srcOffset;

    int dx = get_global_id(0) - get_local_id(0) -1;
    int dy = get_global_id(1) - get_local_id(1) -1;

    const int id = min((int)(get_local_id(0)*8+get_local_id(1)), 5*10-1);

    int dr=id/10;
    int dc=id%10;
    int r=clamp(dy+dr, 0, rows-1);
    int c=clamp(dx+dc, 0, cols-1);

    data[dr][dc] = source[r*srcStep + c];
    r=clamp(dy+dr+5, 0, rows-1);
    data[dr+5][dc] = source[r*srcStep + c];

    barrier(CLK_LOCAL_MEM_FENCE);

    int x =get_local_id(0);
    int y =get_local_id(1);
    uchar p0=data[y][x], p1=data[y][x+1], p2=data[y][x+2];
    uchar p3=data[y+1][x], p4=data[y+1][x+1], p5=data[y+1][x+2];
    uchar p6=data[y+2][x], p7=data[y+2][x+1], p8=data[y+2][x+2];
    uchar mid;

    sw(p1, p2); sw(p4, p5); sw(p7, p8); sw(p0, p1);
    sw(p3, p4); sw(p6, p7); sw(p1, p2); sw(p4, p5);
    sw(p7, p8); sw(p0, p3); sw(p5, p8); sw(p4, p7);
    sw(p3, p6); sw(p1, p4); sw(p2, p5); sw(p4, p7);
    sw(p4, p2); sw(p6, p4); sw(p4, p2);

    if(get_global_id(1)<rows && get_global_id(0)<cols)
        dst[dstOffset + get_global_id(1)*dstStep + get_global_id(0)]=p4;
}
```

Şekil 4.3: Medyan Filtre OpenCL Kernel 3

4.1.3 Dar Boğaz Oluşturan Bölümleri Belirlemek

Belirli mimarilerde, vektör veri tiplerini kullanarak verimlilik elde etmek mümkündür. Kütüphanede bazı algoritmalar vektör veri tiplerini kullanma kabiliyetine büyük ölçüde sahipken, diğerleri bu veri tiplerini kullanamaz. Matris işlemleri ve filtre fonksiyonları gibi bazı fonksiyonlar, vektör veri tipleri kullanılarak uygulanır. Gömülü platformlarda, vektör dışı veri tiplerinin üzerinde uygulandığında, bu fonksiyonlarda hızlanma elde edilmiştir. Perspektif dönüşüm gibi diğer bazı fonksiyonlar ve dönüşüm afin fonksiyonları, vektör veri tipi kullanma kapasitesine sahip olmadığından, bu fonksiyonları, vektör veri tiplerini kullanarak

uygulamadık. Bu durumda, vektör veri tiplerinin kullanılmasının performansı nasıl etkilediği de algoritmaya bağlıdır.

Yerel bellek, genel bellek üzerinde aynı veri unsuruna iki veya daha fazla kez erişim sağlamamızın gerektiği algoritma türleri karşısında avantaj sağlar. Bu durum, kütüphane üzerinde bulunan, konvolüsyon tabanlı algoritmalar ve matris çarpımı gibi bazı algoritmalar üzerinde karşımıza çıkar. Ancak, gömülü platform üzerinde genel bellek uygulaması karşısında herhangi bir hızlanma gerçekleşmemiştir.

4.1.4 Genel Yöntemler

OpenCL kernel performansını artırabilmek yapılabilecek işlemler genel olarak paralel yürütmenin artırılması, maksimum bellek bant genişliğine ulaşabilmek için bellek kullanımını minimuma indirmek ve maksimum buyruk çıkışını sağlamak için buyruk kullanımındaki optimizasyonlardır. Bu optimizasyonlardan yola çıkarak herhangi bir OpenCL kernelı yazarken yapılacak ana işlemler önem sırasına göre şu şekilde gruplandırılabilir.

4.1.4.1 Yüksek Öncelikli:

İlk iş olarak host ve device arasındaki veri transferi minimize edilmelidir. İş-grubu başına düşen iş parçacığı sayısı warp büyüklüğünün tam bir katı olmalıdır. Bu sayede kaynaklar tam kapasite ile kullanılabilir ve bu da optimum hesaplama verimliliği kazandır. Bellek kullanımını iyileştirmek için yani bellek bant genişliğini maksimize edebilmek için genel bellek erişimleri olabildiğince az olmalıdır. Bunun yanında genel bellek kullanımı minimize edilmeli ve daha sıkça erişilen bellek transferleri için yerel bellek kullanımı maksimize edilmelidir.

4.1.4.2 Orta Öncelikli:

Yerel belleğe erişimlerin, yığın çatışması(bank conflict) nedeniyle meydana gelen, taleplerin serileştirilmesini önleyecek şekilde tasarlanması ve aynı zamanda, yerel belleğin genel bellekten fazlalık aktarımları önlemek amacıyla da kullanılması gerekir. Yazmaç bağımlılıklarından kaynaklanan gecikmeyi gizlemek için, çoklu işlemci başına yeterli sayıda etkin iş parçacığının sürdürülmesi gerekir. Daha yavaş ve genel matematik fonksiyonlarının karşısında, mümkün olduğunca daha hızlı ve özel matematik fonksiyonlarının tercih edilmesi ve hızın kesinlikten daha önemli olduğu durumlarda OpenCL matematik kütüphanesinin kullanılması gerekir.

4.1.4.3 Düşük Öncelikli:

Uzun parametre listeli çekirdekler için, yerel bellekten tasarruf sağlamak amacıyla, bazı parametrelerin değişmez belleğe yerleştirilmesi daha verimli olabilir. Değişmez belleğin hızlı olmasının nedeni, önbellekte olmasıdır. Kaydırma işlemleri kullanmak ve döngü sayacı olarak işaretsiz tam sayılar yerine işaretli tam sayılar kullanmak verimlilik açısından daha iyi yaklaşımlardır.

4.1.5 Kernel Füzyon

Geliştirdiğimiz bu kütüphanenin ana kullanım alanlarından olan ve önemli bir performans kazanımı sunması olası olan bir uygulama alanı da gerçek zamanlı görüntü işleme uygulamarıdır diyebiliriz. Bu tarz uygulamalarda genel olarak bir çok görüntü işleme filtresi herhangi bir videodaki her bir imgeye sıralı bir biçimde uygulanmaktadır. Alışılagelmiş GPGPU uygulamalarında OpenCL kernelları olarak implement edilmiş ve sıralı olarak çalışan bu filtrelerde her bir kernel her seferinde tüm veriyi kopyalayıp onun üzerinde işlem yapmaktadır. Örnek olarak bir imge üzerinde yapılacak olan birden çok filtre için her seferinde giriş ve çıkış verileri host ve device arasında transfer edilmektedir. Bunun üstesinden

gelebilmek için olası bir çözüm bu verileri kopyalamak yerine device üzerinde bir sonraki kernelın verisini göstermek ve ulaşmasını sağlayabilmek olabilir. Buna ek olarak bir diğer çözüm ise çoklu kernelları tek bir kernel üzerinden gönderebilmek olacaktır. Kernel füzyonu dediğimiz bu işlem veri transfer sürelerini azaltırken device üzerinde veri yerelliğini artırmakta ve güç tüketimini de azaltmaktadır. Bu optimizasyon adımında ana hedef olarak yukarıda belirttiğimiz şekilde çalışan GPGPU uygulamalarının veri yerelliğini artırarak performans kazanmak ve güç tüketimini azaltmak diyebiliriz.

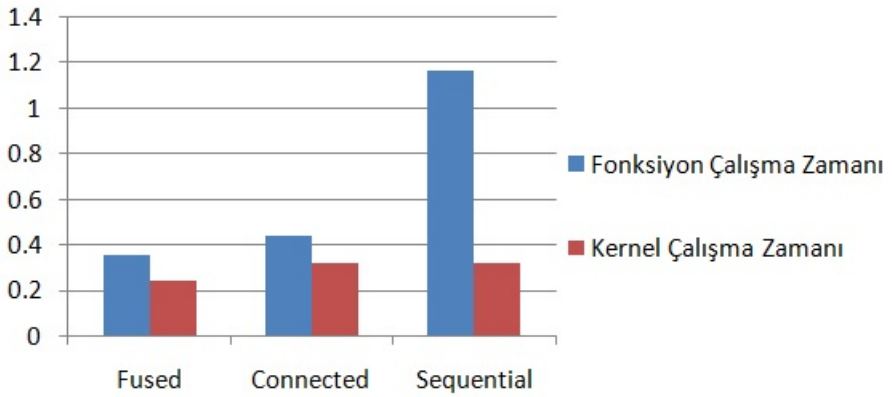
Bu amaç doğrultusunda geliştirdiğimiz kernel füzyon aracı birden farklı işlem sırasında çalışma ihtimali olan birden çok kernelı device tarafında otomatik olarak birbirine bağlamakta ve veri transfer sürelerini azaltmaktadır. Buna ek olarak birden çok kernel ile ardı sıra çalışan uygulamalar tek bir kernel olarak implement edilmiş ve veri yerelliği artırılarak olası performans kazanımları değerlendirilmiştir. Bu çalışmadaki ana amaç olası bir kaynak koddan kaynak koda OpenCL derleyicisinin performans kazanımlarını tahmin etmektir. Bu amaç doğrultusunda bazı OpenCL kernelleri ve bu kernelların manuel olarak füzyon edilmiş versiyonları geliştirilmiştir. Geliştirilen kernellar algoritmik karmaşıklıklarına göre 2 grupta toplanmıştır.

- Düşük Karmaşıklık
  - Piksel Tabanlı Çalışan Kernellar
  - Veri bağımlılığı yok
  - Blok tabanlı bellek erişimleri yok
- Orta Karmaşıklık
  - Blok Tabanlı Çalışan Kernellar
  - Veri bağımlılığı yok
  - Blok tabanlı bellek erişimleri var

Birinci grup kernellarda piksel tabanlı çalışan 4 renk uzayı dönüşümü ve 1 gamma düzeltme fonksiyonu gerçekleştirilmiştir. İkinci grup kernellarda ise bu

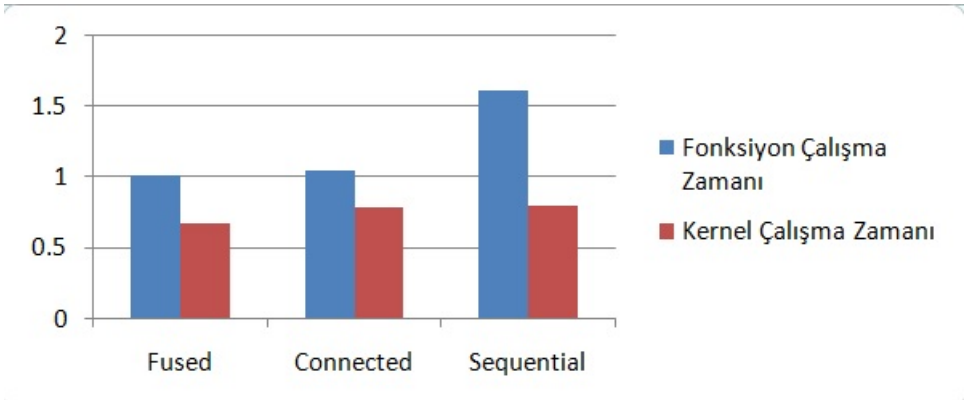
fonksiyonlara ek olarak median,sobel,laplace, gaussian gibi blok tabanlı çalışan filtreleme fonksiyonları bulunmaktadır.

Şekil 4.4' de 1. grup piksel tabanlı çalışan kernellar için her seferinde verinin CPU-GPU arasında kopyalandığı "sequential", verinin sadece ilk kernel için kopyalandıktan sonra, son çalışan kernela kadar GPU tarafında tutulduğu ve bunun ardından CPU'ya kopyalandığı "connected" ve çoklu kernelların tek bir kernel olarak yazılmış versiyonlarının çalıştırıldığı "fused" versiyonlar için fonksiyon ve kernel çalışma zamanlarına ait bilgiler bulunmaktadır.



Şekil 4.4: Kernel Füzyon Düşük Karmaşıklık Grubu Hızlanmalar

Şekil 4.5' de 2. grup blok tabanlı çalışan kernellar için her seferinde verinin CPU-GPU arasında kopyalandığı "sequential", verinin sadece ilk kernel için kopyalandıktan sonra, son çalışan kernela kadar GPU tarafında tutulduğu ve bunun ardından CPU'ya kopyalandığı "connected" ve çoklu kernelların tek bir kernel olarak yazılmış versiyonlarının çalıştırıldığı "fused" versiyonlar için fonksiyon ve kernel çalışma zamanlarına ait bilgiler bulunmaktadır.



Şekil 4.5: Kernel Füzyon Orta Karmaşıklık Grubu Hızlanmalar

Sonuç olarak kernel füzyonu birinci grup olan basit kernellarda ortalamada %32 performans kazanımı sunarken, blok tabanlı çalışan kernellarda ortalamada %17 kazanç sağlamıştır. Buna ek olarak GPU versiyonları skalar CPU kodlarına nazaran ortalamada 42 kat hızlanma sağlarken, skalar versiyonlarında bu hızlanma 10 kat olarak ölçülmüştür.

Geliştirilen fonksiyonların ve bunların manuel olarak füzyon edilmiş versiyonlarının çalışma zamanları Şekil 4.6 ve Şekil 4.7' de verilmiştir. Buna ek olarak güç tüketimlerine ait bilgiler de görülebilir.

		Sistem 1 Çalışma Süresi		Sistem 2 Çalışma Süresi		Güç Tüketimi
Düşük Karmaşıklık	Filters	Fonksiyon	Kernel	Fonksiyon	Kernel	
	YUV420TO444(K1)	***0.8576215***	0.11616	***3.859104***	0.7775651	88.989383
	YUV444TORGB(K2)	***1.00936***	0.14088	***6.135427***	1.25259712	90.653227
	GAMMA(K3)	***1.893285***	1.02	***7.848835***	2.87350787	91.973795
	RGBTOYUV444(K4)	***1.158047***	0.18312	***5.859703***	0.77279467	89.358483
	YUV444TO420(K5)	***0.9238927***	0.12792	***7.217167***	2.79703821	88.664445
		Sistem 1 Çalışma Süresi		Sistem 2 Çalışma Süresi		
Orta Karmaşıklık	Filters	Fonksiyon	Kernel	Fonksiyon	Kernel	
	K1-K2	***0.819026***	0.13632	***4.231395***	1.23594921	88.447201
	K1|K2	***1.21624***	0.24996	***6.44892***	2.01283	91.302843
	K1-K2-K3	***1.692305***	0.99216	***5.610634***	2.3211384	91.025332
	K1|K2|K3	***1.96928***	1.18322	***7.19839***	4.81829	93.392037
	K4-K5	***0.952813***	0.17112	***7.16998***	2.80797289	87.976498
	K4|K5	***1.32929***	0.29083	***9.76822***	3.56823	89.389201
	K3-K4-K5	***1.762893***	1.01736	***7.062394***	2.859241	92.637876
	K3|K4|K5	***2.198332***	1.26928	***10.27925***	6.41439	93.892739
	K1-K2 | K3 | K4-K5	***1.948832***	1.3284	***9.287781***	6.90659872	95.602552
	K1-K2-K3 | K4-K5	***1.703889***	1.1608	***7.29411***	4.947978311	95.706592
	K1-K2 | K3-K4-K5	***1.682392***	1.16088	***6.451223***	4.05333781	95.884825
	K1-K2-K3-K4-K5	***2.19832***	1.59024	***10.22092***	8.41952226	95.557046

Şekil 4.6: Kernel Füzyon Düşük Karmaşıklık Grubu Sonuçları

		Sistem 1 Çalışma Süresi		Sistem 2 Çalışma Süresi		Güç Tüketimi
Düşük Karmaşıklık	Filters	Fonksiyon	Kernel	Fonksiyon	Kernel	
	MEDIAN(K1)	***1.64635***	0.85964	***8.4682***	3.59543	91.175353
	GAUSSIAN(K2)	***1.77453***	0.93586	***9.6359***	4.64821	91.360658
	LAPLACIAN(K3)	***1.52647***	0.82658	***6.98525***	2.15239	89.607325
	SOBEL(K4)	***1.57305***	0.77596	***7.22381***	2.89514	89.876219
	BOX(K5)	***1.54874***	0.7674	***8.26743***	3.48182	89.610288
	NORMALIZEDBOX(K6)	***1.48931***	0.49681	***7.05648***	2.70286	89.55281
Orta Karmaşıklık	Filters	Sistem 1 Çalışma Süresi		Sistem 2 Çalışma Süresi		
		Fonksiyon	Kernel	Fonksiyon	Kernel	
	K1|K4	***1.73943***	0.9845	***9.65846***	4.75642	91.903588
	K1|K3	***1.73168***	0.96872	***10.15917***	5.17256	91.912005
	K1-K4	***1.68541***	0.90456	***9.25646***	4.65241	91.502565
	K1-K3	***1.67828***	0.91584	***9.84561***	5.51668	91.726024
	K2|K4	***1.9128***	0.99522	***10.65863***	5.82294	92.568765
	K2|K3	***1.89727***	0.94251	***10.25886***	5.18213	92.046259
	K2K4	***1.84197***	0.97915	***9.96763***	5.05486	92.008584
	K2K3	***1.84008***	0.93681	***9.92058***	5.02846	91.945642
	K1|K5	***1.74058***	0.97852	***9.105558***	4.75846	91.849821
	K1|K6	***1.73586***	0.95257	***9.00546***	4.48516	91.842022
	K1K5	***1.67576***	0.90025	***8.76216***	4.06482	91.492042
	K1K6	***1.67684***	0.89571	***8.72462***	4.00413	91.897348
	K2|K5	***1.91057***	0.99135	***10.2596***	5.82452	92.546232
	K2|K6	***1.90852***	0.95146	***10.00926***	5.62758	92.254546
	K2K5	***1.82596***	0.95157	***9.90813***	5.24025	92.004568
	K2K6	***1.79917***	0.93581	***9.72272***	5.00845	91.867546

Şekil 4.7: Kernel Füzyon Orta Karmaşıklık Grubu Sonuçları

5. SONUÇLAR

5.1 Performans Sonuçları

Bu bölümde geliştirdiğimiz görüntü işleme kütüphanesinin içerdiği fonksiyonların farklı platformlar üzerinde çalıştırılması sonucu elde edilen performans sonuçları bulunmaktadır.Elde edilen tüm sonuçlar gömülü ve masaüstü olmak üzere iki farklı sistem üzerinde denenmiştir. İlk test ortamımız olan gömülü platformumuz QuadCore ARM Cortex A9 merkezi işlem birimi ile Vivante GC2000 grafik işlem birimi bulundurmaktadır.Diğer test platformumuz ise Intel 2670QM i7 merkezi işlem birimi ile Nvidia GT555M grafik işlem birimi bulundurmaktadır. Test platformlarına ait detaylar Tablo 5.1' de bulunmaktadır.

Tablo 5.1: Test Sistemleri Spesifikasyonları

Gömülü Platform		Masaüstü Sistem	
GPU	GC2000	GPU	GTX560Ti
GPU Çekirdek Saat Frekansı	600	GPU Çekirdek Saat Frekansı	1645
GPU Çekirdek Sayısı	4	GPU Çekirdek Sayısı	384
CPU	Cortex A9	CPU	Intel i7 2600k
CPU Çekirdek Saat Frekansı	1200	CPU Çekirdek Saat Frekansı	3400(Maks:3800)
CPU Bellek Saat Frekansı	533	CPU Bellek Saat Frekansı	1333

Buna ek olarak Vivante GC2000'e ait bazı ek özellikler şu şekildedir. GC2000' de her bir iş grubu 16 veya daha az iş parçacığı barındırabilmektedir. Bu nedenle OpenCL kernellarının iş grubu büyüklüğü 16 ve onun katı olmalıdır diyebiliriz. 16 ve onun katı olmaması durumunda iş grupları tam kapasite ile çalışmayabilir. Örneğin iş grubu sayısını 8 yapmamız durumunda bu iş grubu aynı and 16 iş parçacığını çalıştırabiliyorken 8 tane çalıştıracak ve verimsiz bir

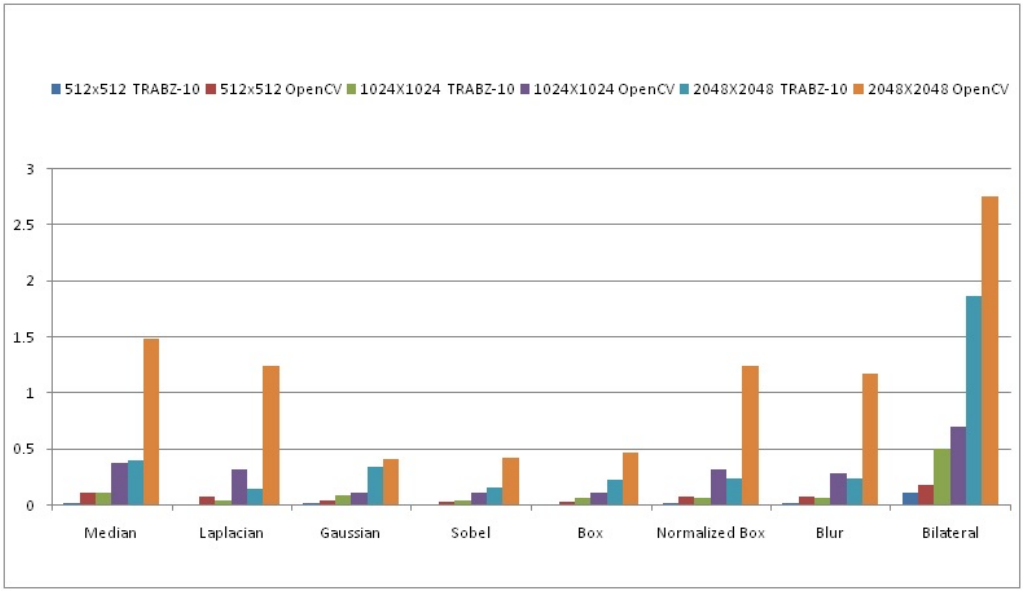
Tablo 5.2: Vivante GC2000 Bellek Hiyerarşisi

Bellek Tipleri	Bellek Yapısı	Tanımı
Özel Bellek(Private Memory)	Geçici registerlar,L1 Cache	İlgili İş Parçacığına Özel, Diğer iş parçacıkları erişemez
Yerel Bellek(Local Memory)	Yerel Registerler, L1 Cache	İş grubuna özel, İlgili iş grubundaki tüm iş parçacıkları erişebilir
Genel Bellek(Global Memory)	Sistem Belleği, Geçici Registerlar, L1 Cache	Tüm iş parçacıkları tarafından erişilebilir. Bunun yanında Host tarafından da erişilebilir.
Değişmez Bellek(Constant Memory)	Sabir Registerlar, L1 Cache	Host tarafından alanı ayrılmış ve ilgili değerler yüklenmiş bellek tipidir. Kernel çalışma süresi boyunca değişmez.
Host Belleği	RAM	Tüm uygulamaların erişebileceği bellek

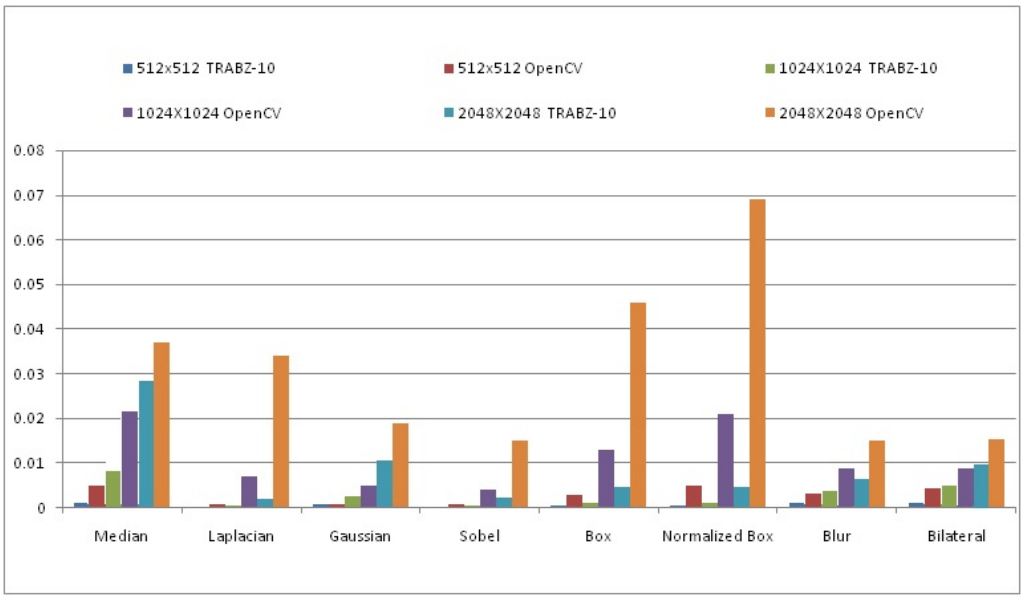
durum oluşacaktır. Donanım tarafında iş grubu büyüklüğü 16 ve onun katı olarak ayarlanmaktadır. Örneğin iş grubu büyüklüğünü 17 verirsek, donanım tarafında bu iki ayrı iş grubu olarak işlenecek ve biri tam kapasite ile çalıştırılabilirken diğerinde tek bir iş parçacığı çalışacaktır. Bu da benzer bir verimsizliğe neden olacaktır. Bunların yanında GC2000'in performansı olumsuz olarak etkileyen bazı kısıtları şu şekildedir. GC2000 her bir işlem çekirdeğinde 64 adet 32 bit register bulundurmakta ve OpenCL kernelı başına maksimum buyruk sayısını 512 ile kısıtlamaktadır. Teoride yerel bellek bulunuyor olmasına rağmen bu yerel belleğin genel bellek üzerinde olması olası kazanımlarını ortadan kaldırmaktadır. GC2000' e ait bellek bellek hiyerarşisi Tablo 5.2' de gösterilmiştir.

Grafiklerde gösterilen tüm sonuçlar kernel çalışma zamanlarının saniye cinsinden karşılıklarıdır. Şekil 5.1 ve Şekil 5.2 farklı filtreleme sonuçlarının farklı platformlar üzerinde kernel çalışma zamanlarını göstermektedir. Şekil 5.1 gösterilen filtreleme sonuçları gömülü test platformumuz olan 1 nolu test ortamızda alınmıştır. OpenCV fonksiyonları ARM Cortex A9 üzerinde çalıştırılmış iken TRABZ-10 fonksiyonları Vivante GC2000 üzerinde çalıştırılmıştır. Şekil 5.2 ise aynı fonksiyonların 2 nolu test ortamında alınması sonucu elde edilmiştir. Bu test ortamında OpenCV fonksiyonları Intel 2670QM i7 merkezi işlem biriminde koşturulurken, TRABZ-10 fonksiyonları Nvidia GT555M grafik işlem birimi üzerinde elde edilmiştir. Sonuçlardan da görüleceği üzere TRABZ-10 fonksiyonların bir çoğunda daha iyi sonuçlar elde etmiştir. Bunun yanında görüntü büyüklüğü arttıkça

performans kazanımında arttığı görülebilir. Bunun nedeni görüntü büyüklüğünün artması sonucu merkezi işlem birimi ile grafik işlem biriminin arasındaki veri transferinin büyümesi ve buradaki bellek bant genişliğinin daha verimli bir şekilde kullanılmasıdır.



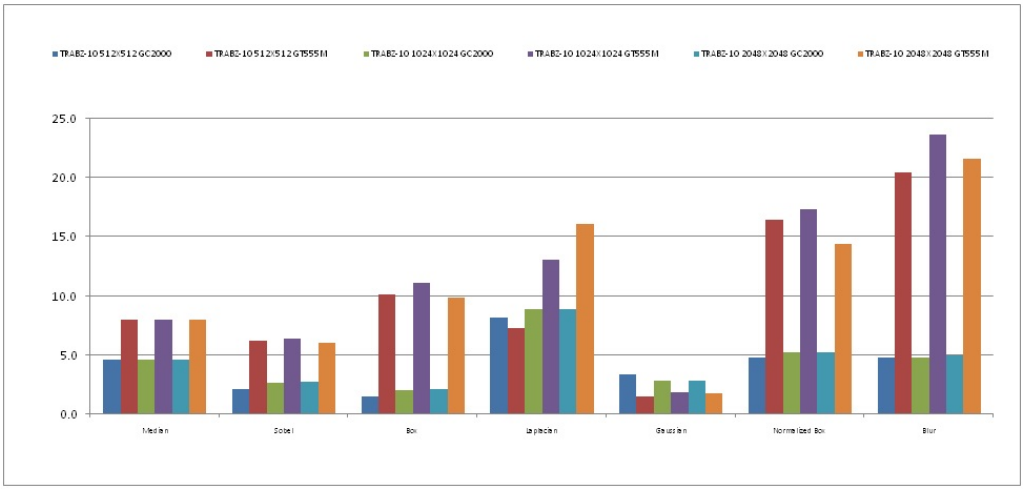
Şekil 5.1: Filtreleme Fonksiyonları Çalışma Zamanları(Gömülü Platform)



Şekil 5.2: Filtreleme Fonksiyonları Çalışma Zamanları(Masaüstü Platform)

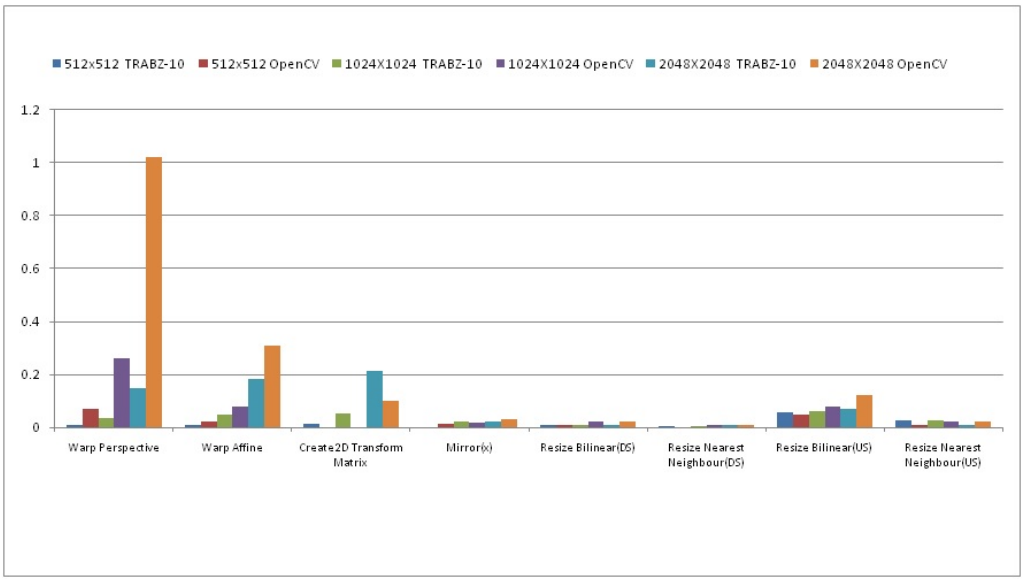
Sonuçlar genel olarak karşılaştırıldığında ise TRABZ-10 görüntü işleme kütüphanesi 1 nolu test ortamında OpenCV'ye göre ortalama 7 kata kadar hızlanma sağlamıştır. Bunun yanında 2 nolu test ortamızda bu hızlanma oranı 20 kata kadar ölçülmüştür. Bunun nedeni ise 2 nolu test ortamızda ki grafik işlem biriminin daha güçlü bir donanıma sahip olması diyebiliriz. Sonuçlar Şekil 5.3' de gösterilmektedir.

Şekil 5.4 ve Şekil 5.5 farklı dönüşüm fonksiyonları sonuçlarının farklı platformlar üzerinde kernel çalışma zamanlarını göstermektedir. Şekil 5.4 gösterilen dönüşüm sonuçları gömülü test platformumuz olan 1 nolu test ortamızda alınmıştır. OpenCV fonksiyonları ARM Cortex A9 üzerinde çalıştırılmış iken TRABZ-10 fonksiyonları Vivante GC2000 üzerinde çalıştırılmıştır. Şekil 5.5 ise aynı fonksiyonların 2 nolu test ortamında alınması sonucu elde edilmiştir. Bu test ortamında OpenCV fonksiyonları Intel 2670QM i7 merkezi işlem biriminde koşturulurken,TRABZ-10 fonksiyonları Nvidia GT555M grafik işlem birimi üzerinde elde edilmiştir. Sonuçlardan da görüleceği üzere TRABZ-10 fonksiyonların bir çoğunda daha iyi sonuçlar elde etmiştir. Şekillerden de görüleceği üzere en

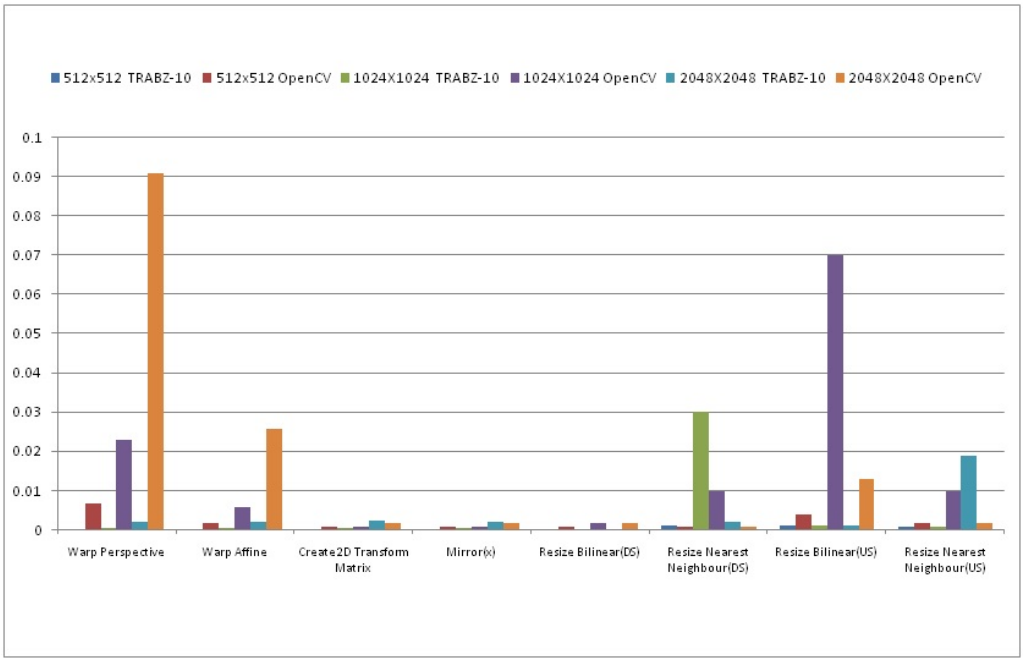


Şekil 5.3: Her iki test platformu üzerinde OpenCV ile karşılaştırma sonucu kazanılan hızlanma oranları

TRABZ-10 görüntü işleme kütüphanesinin OpenCV'ye göre en büyük performans kazanımının olduğu fonksiyonlar afin dönüşüm fonksiyonlarıdır. Bunun önemli bir nedeni bu fonksiyonların bölüm 3'te anlatılan algoritmik yapılarıdır. Bellek işlemlerinin az olması ve veri paralel işlemeye uygun olan bu yapı performans kazanımlarını en optimum seviyeye taşımıştır. Diğer fonksiyonlarda ise OpenCV ile yaklaşık sonuçlar elde edilmiştir.

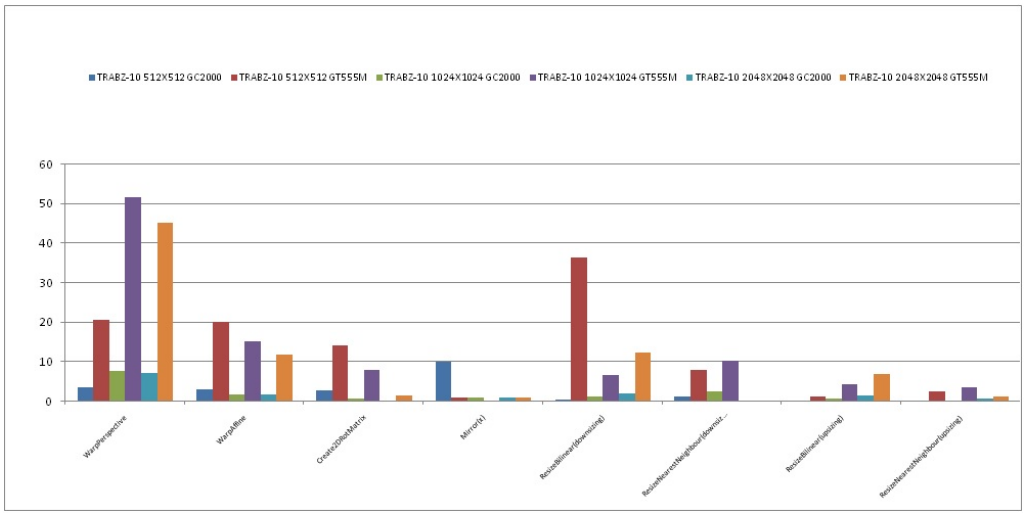


Şekil 5.4: Dönüşüm Fonksiyonları Çalışma Zamanları(Gömülü Platform)



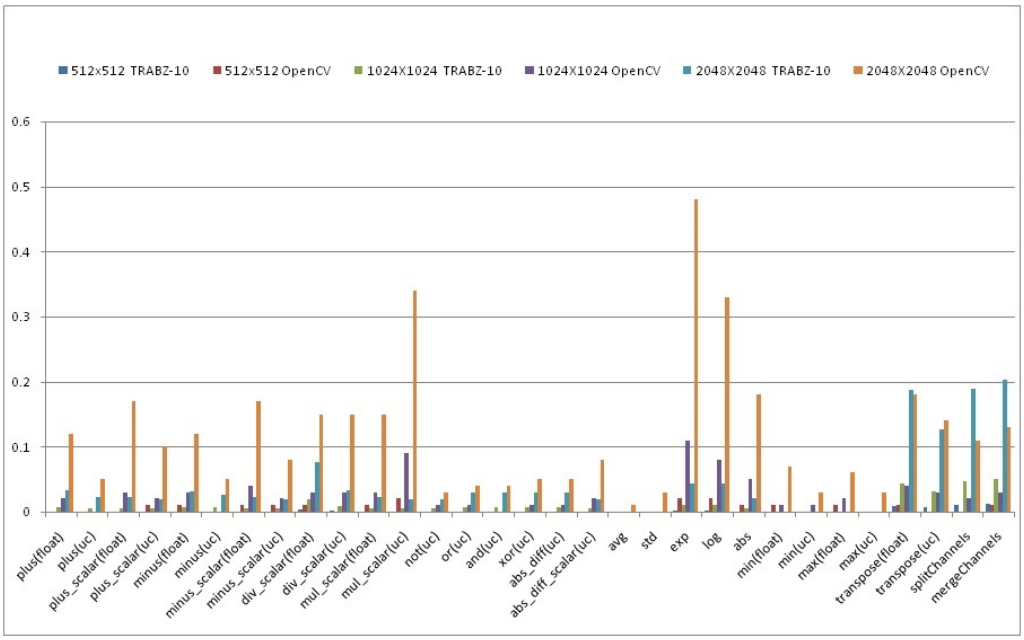
Şekil 5.5: Dönüşüm Fonksiyonları Çalışma Zamanları(Masaüstü Platform)

Sonuçlar genel olarak karşılaştırıldığında ise TRABZ-10 görüntü işleme kütüphanesi 1 nolu test ortamında OpenCV'ye göre ortalama 8 kata kadar hızlanma sağlamıştır. Bunun yanında 2 nolu test ortamızda bu hızlanma oranı 45 kata kadar ölçülmüştür. Sonuçlar Şekil 5.6' de gösterilmektedir.



Şekil 5.6: Her iki test platformu üzerinde OpenCV ile karşılaştırma sonucu kazanılan hızlanma oranları

Şekil 5.7 görüntü işlemede sıklıkla kullanılan bir diğer grup olan matris işlemlerinin bulunduğu fonksiyonların çalışma zamanlarını göstermektedir. Şekilden de görüleceği üzeri TRABZ-10, OpenCV' ye nazaran daha iyi bir performansa sahiptir. Bölüm 3'te de bahsedildiği üzere bazı matris işlemler algoritmik yapısından dolayı özyinelemeli olarak çalıştığı için CPU tarafında implement edilmiştir.



Şekil 5.7: Matris İşlemleri için Çalışma Zamanları(Gömülü Platform)

5.2 Gelecek Çalısmalar

Bu çalışmada gömülü sistemler üzerinde test edilmiş ve görüntü işleme alanında bir standart haline gelmiş olan OpenCV ile performans ve fonksiyonel açıdan karşılaştırılarak sonuçları sunulmuş bir görüntü işleme kütüphanesi geliştirilmiştir. Sonuçlar incelendiğinde OpenCV'ye göre ortalama 7 kat hızlanma sağlanmıştır. Geliştirilen bu kütüphane temel görüntü işleme fonksiyonlarını içermekte olup gelecek çalışmalarda bu fonksiyonlara ek başka fonksiyonlar da eklenebilir. Gelecek çalışmalarda daha detaylı analizler yapabilmek adına, performans sonuçlarına ek olarak geliştirilen bu kütüphanenin detaylı güç tüketimi değerlerinin farklı platformlar üzerinde ölçülmesi planlanmaktadır. Bunların dışında gelişen donanımlar ve OpenCL spesifikasyonlarındaki değişikliklere bağlı olarak algoritmalar üzerinde optimizasyon çalışmaları devam edecektir.

Kaynakça

[1] Khronos Group. 2012. OpenGL Shading Language Specification. http://www.opengl.org/documentation/glsl

[2] Randi J. Rost., ÖpenGL Shading Language (3. Ed.).; 2004 Addison-Wesley Professional

[3] Khronos Group. 2012. OpenGL ES 2.0 Specification, Khronos Group. http://www.khronos.org/opengles

[4] Aaftab Munshi GLES. 2008. OpenGL ES 2.0 Programming Guide. Addison Wesley

[5] Ricardo Marroquim and Andre Maximo. 2009. Introduction to GPU Programming with GLSL. Computer Graphics and Image Processing (SIBGRAPI TUTORIALS), 2009 Tutorials of the XXII Brazilian Symposium on, vol., no., pp.3,16, 11-14 Oct. 2009

[6] John D. Owens and David Luebke and Naga Govindaraju and Mark Harris and Jens Krüger and Aaron E. Lefohn and Timothy J. Purcell. 2007. A Survey of general-purpose computation on graphics hardware. Computer graphics forum, Volume 26. Wiley Online Library

[7] Nitin Singhal and In Kyu Park and Sungdae Cho. 2010. Implementation and optimization of image processing algorithms on handheld gpu. Image Processing (ICIP), 2010 17th IEEE International Conference on , vol., no., pp.4481,4484, 26-29 Sept. 2010

[8] J-P Farrugia and Patrick Horain and Erwan Guehenneux and Yannick Alusse. 2006. GPUCV: A Framework for Image Processing Acceleration with Graphics Processors. Multimedia and Expo, 2006 IEEE International Conference on, vol., no., pp.585,588, 9-12 July 2006.

[9] Pavel Babenko and Mubarak Shah. 2008. MinGPU: a minimum GPU library for computer vision. Journal of Real-Time Image Processing 3.4 (2008): 255-268.

[10] James Fung and Steve Mann. 2005. OpenVIDIA: parallel GPU computer vision. Proceedings of the 13th annual ACM international conference on multimedia. ACM

[11] Erik Lindholm. 2008. NVIDIA Tesla: A unified graphics and computing architecture. Micro, IEEE 28.2 (2008): 39-55

[12] Zhiyi Yang and Yating Zhu and Yong Pu. 2008. Parallel image processing based on CUDA. Computer Science and Software Engineering, 2008 International Conference on. Vol. 3. IEEE, 2008

[13] Yannick Allusse. 2008. GpuCV: an opensource GPU-accelerated framework forimage processing and computer vision. Proceedings of the 16th ACM international conference on Multimedia. ACM, 2008.

[14] Jingfei Kong. 2010. Accelerating MATLAB image processing toolbox functions on GPUs. Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, ACM, 2010.

[15] Yuancheng Luo and Ramani Duraiswami. 2008. Canny edge detection on NVIDIA CUDA. Computer Vision and Pattern Recognition Workshops, CVPRW'08. IEEE Computer Society Conference on. IEEE

[16] Aaftab Munshi CL. 2008. OpenCL Parallel Computing on the GPU and CPU, SIGGRAPH (2008). OpenCL Spec. 2011. OpenCL Specification, Khronos OpenCL Working Group. Retrieved April 25, 2013 from http://www.khronos.org/registry/cl

[17] Tomasz S. Czajkowski. 2012. From OpenCL to high-performance hardware on FPGAs. Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on. IEEE, 2012.

[18] Kamran Karimi and Neil G. Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. arXiv preprint arXiv:1005.2581 (2010).

[19] Jianbin Fang and Ana Lucia Varbanescu and Henk Sips. 2011. A comprehensive performance comparison of CUDA and OpenCL. Parallel Processing (ICPP), 2011 International Conference on. IEEE, 2011.

[20] Peng Du. 2011. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing (2011).

[21] Gary Bradski and Adrian Kaehler. 2008. Learning OpenCV: Computer vision with the OpenCV library. O'Reilly Media, Incorporated, 2008.

[22] Aaftab Munshi and Benedict Gaster and Timothy G. Mattson. 2011. OpenCL programming guide. Addison-Wesley Professional, 2011.

[23] Jyrki Leskela and Jarmo Nikula and Mika Salmela. 2009. OpenCL embedded profile prototype in mobile device. Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on. IEEE, 2009.

[24] Kwang-Ting Cheng and Yi-Chu Wang. 2011. Using mobile GPU for general-purpose computing–a case study of face recognition on smartphones. VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on. IEEE, 2011.

[25] Miguel Bordallo López. 2011. Accelerating image recognition on mobile devices using GPGPU. IS&T/SPIE Electronic Imaging. International Society for Optics and Photonics, 2011.

[26] Vivante. Vivante Corporation, http://www.vivantecorp.com, May 2013. FreeScale. ï.MX 6q Fact Sheet:

[27] N. T. Prosser. 2010. Medical image segmentation using GPU-accelerated variational level set methods (Doctoral dissertation, Rochester Institute of Technology).

[28] Tomasz Kornuta and Mateusz Pruchniak. 2010. Utilization of GPU for real-time vision in robotics. Signal Processing Algorithms, Architectures, Arrangements, and Applications Conference Proceedings (SPA), 2010. IEEE, 2010.

[29] BT.601. 2011. Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios, http://www.itu.int/rec/R-REC-BT.601-7-201103-I/en, March 2013

ÖZGEÇMİŞ

Kişisel Bilgiler

Soyadı, Adı : Hakkı Doğaner Sümerkan
e-mail : hdsumerkan@gmailcom

Eğitim

Derece	Eğitim Birimi	Mezuniyet Tarihi
Y. Lisans	TOBB Ekonomi ve Teknoloji Üniversitesi	2014
Lisans	TOBB Ekonomi ve Teknoloji Üniversitesi	2011

İş Deneyimi

Yıl	Yer	Görev
2011-2014	TOBB Ekonomi ve Teknoloji Üniversitesi	Burslu Y.L. Öğrencisi

Yabancı Dil

İngilizce (İleri Seviye)
Almanca (Başlangıç Seviyesi)

Yayınlar

Mustafa Cavus, Hakki Doganer Sumerkan, Osman Seckin Simsek, Hasan Hassan, Abdullah Giray Yaglikci, Oguz Ergin: GPU based Parallel Image Processing Library for Embedded Systems. VISAPP (1) 2014