EGE ÜNİVERSİTESİ

EÜ FEN BİLİMLERİ ENSTİTÜSÜ

YÜKSEK LİSANS TEZİ

CUDA İLE PARALEL PROGRAMLAMA

Pelin KARAGÖZOĞLU

Tez Danışmanı :Prof. Dr. Aylin KANTARCI

Bilgisayar Mühendisliği Anabilim Dalı

Sunuş Tarihi : 16.05.2018

Bornova-İZMİR
2018

EGE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

(YÜKSEK LİSANS TEZİ)

CUDA İLE PARALEL PROGRAMLAMA

Pelin KARAGÖZOĞLU

Tez Danışmanı : Prof. Dr. Aylin KANTARCI

Bilgisayar Mühendisliği Anabilim Dalı

Sunuş Tarihi : 16.05.2018

Bornova-İZMİR

2018

Pelin KARAGÖZOĞLU tarafından yüksek lisans tezi olarak sunulan "CUDA ile Paralel Programlama" başlıklı bu çalışma EÜ Lisansüstü Eğitim ve Öğretim Yönetmeliği ile EÜ Fen Bilimleri Enstitüsü Eğitim ve Öğretim Yönergesi'nin ilgili hükümleri uyarınca tarafımızdan değerlendirilerek savunmaya değer bulunmuş ve 16/05/2018 tarihinde yapılan tez savunma sınavında aday oybirliği/oyçokluğu ile başarılı bulunmuştur.

Jüri Üyeleri: **İmza**

Jüri Başkanı : Prof. Dr. Aylin KANTARCI .....

Raportör Üye: Doç. Dr. Ayşegül ALAYBEYOĞLU .....

Üye : Yrd. Doç. Dr. Şebnem BORA .....

EGE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

ETİK KURALLARA UYGUNLUK BEYANI

EÜ Lisansüstü Eğitim ve Öğretim Yönetmeliğinin ilgili hükümleri uyarınca Yüksek Lisans Tezi olarak sunduğum "CUDA ile Paralel Programlama" başlıklı bu tezin kendi çalışmam olduğunu, sunduğum tüm sonuç, doküman, bilgi ve belgeleri bizzat ve bu tez çalışması kapsamında elde ettiğimi, bu tez çalışmasıyla elde edilmeyen bütün bilgi ve yorumlara atıf yaptığımı ve bunları kaynaklar listesinde usulüne uygun olarak verdiğimi, tez çalışması ve yazımı sırasında patent ve telif haklarını ihlal edici bir davranışımın olmadığını, bu tezin herhangi bir bölümünü bu üniversite veya diğer bir üniversitede başka bir tez çalışması içinde sunmadığımı, bu tezin planlanmasından yazımına kadar bütün safhalarda bilimsel etik kurallarına uygun olarak davrandığımı ve aksinin ortaya çıkması durumunda her türlü yasal sonucu kabul edeceğimi beyan ederim.

16 / 05 / 2018

İmzası

Pelin Karagözoğlu

ÖZET

CUDA İLE PARALEL PROGRAMLAMA

KARAGÖZOĞLU, Pelin

Yüksek Lisans Tezi, Bilgisayar Mühendisliği Anabilim Dalı
Tez Danışmanı: Prof. Dr. Aylin KANTARCI
Nisan 2018, 40 sayfa

Son zamanlarda GPU'ların büyük gelişme göstermesi ile birlikte CPU'ların yardımcı işlemcisi olarak farklı alanlarda da kullanılmaya başlamıştır. Özellikle, NVIDIA'nın CUDA paralel programlama platformunu geliştiricilere sunmasıyla birlikte GPU kullanımı görüntü işleme gibi büyük hesaplama gerektiren uygulamaların paralelleştirilmesinde büyük kolaylıklar sağlamıştır.

Bu tez kapsamında da NVIDIA CUDA kullanımı ile GPU üzerinde paralel programlama geliştirme incelenmiş ve görüntü işleme uygulamalarının en çok kullanılan yöntemlerinden histogram hesaplama ve eşitleme algoritmaları CUDA programlama yardımı ile paralelleştirilerek algoritmaların seri ve paralel versiyonları iki farklı CPU ve GPU üzerinde çalıştırılmış, performans karşılaştırmaları yapılmıştır.

Anahtar sözcükler: CUDA, histogram, histogram eşitleme, paralel programlama, CPU, GPU.

ABSTRACT

PARALLEL PROGRAMMING WİTH CUDA

KARAGÖZOĞLU, Pelin

MSc in Computer Eng.
Supervisor: Prof. Dr. Aylin KANTARCI
April 2018, 40 pages

Recently, GPUs show great improvements and they are being used in different areas as a CPU coprocessor. Especially, after that NVIDIA's CUDA parallel programming platform is presented to developers, the GPU usage provided great convenience in parallelizing application such as image processing which requires large computation.

In this thesis, parallel programming development on GPU with CUDA was investigated. Also histogram computation and equalization, the most commonly used methods of image processing application, have been parallelized by using CUDA parallel programming. In addition, to compare the execution times of performance, both serial and parallel versions of these algorithms were executed on two different CPU and GPUs.

Keywords: CUDA, histogram, histogram equalization, parallel programming, CPU, GPU.

TEŞEKKÜR

Tez çalışması boyunca önerileri, rehberliği, değerli yorumları ve destekleri için danışmanım Prof. Dr. Aylin Kantarcı'ya teşekkürlerimi sunarım. Son olarak sevgili aileme her daim bana verdikleri destek ve sabır için en içten dileklerimle teşekkür ederim.

İÇİNDEKİLER

Sayfa

İÇİNDEKİLER (devam)

Sayfa

ŞEKİLLER DİZİNİ

Şekil Sayfa

ŞEKİLLER DİZİNİ (devam)

Şekil Sayfa

ÇİZELGELER DİZİNİ

Çizelge Sayfa

SİMGELER VE KISALTMALAR DİZİNİ

Kısaltmalar

ALU	Aritmetik mantık birimi
CDF	Kümülatif dağıtım fonksiyonu
CPU	Merkezi işlem birimi
CUDA	Birleştirilmiş hesaplama aygıt mimarisi
DRAM	Dinamik rastgele erişim belleği
FPU	Kayan nokta birimi
GPU	Grafik işlem birim
JPEG	Birleşik fotoğraf uzmanları grubu
LD/ST	Yükleme/depolama birimi
PCI	Çevresel bileşen bağlantısı
PGM	Taşınabilir gri harita dosya formatı
PPM	Taşınabilir imge harita dosya formatı
RGB	Kırmızı, yeşil, mavi
SFU	Özel fonksiyon birimi
SM	Akış işlemcisi
SMM	Akış çok işlemcisi
SIMT	Tek talimat çoklu iş parçacığı

SİMGELER VE KISALTMALAR DİZİNİ (devam)

Kısaltmalar

YUV Luminance (siyah-beyaz), chrominance1 (mavi), chrominance2 (kırmızı)

1. GİRİŞ

Günümüzde GPU'lar bilgisayar sistemleri için vazgeçilmez birimler haline gelmiştir. Bunun nedeni CPU'ların tek bir görev için hızlı yanıt süresi sağlayan, az işlemci çekirdeğine sahip olarak tasarlanmış olmasıdır. Özellikle görüntü işleme gibi yoğun hesaplama ve paralellik gerektiren durumlarda CPU'nun performansı yetersiz kalmaktadır. GPU'ların ise, içerisindeki yüzlerce çekirdeği kullanması ile birlikte paralel bir şekilde grafiksel ve büyük hesaplamaları kolayca yapabilme kabiliyetine sahip olması ve temel görüntü işlemlerini yerine getiren donanımlar içermesi, problemlerin çözümünde büyük kolaylık sağlamıştır (Lee et al., 2010).

GPU kullanımına olan ihtiyacın artmasıyla birlikte, NVIDIA tarafından 2006 yılında, paralel programlama mimarisi ve platformu olan birleştirilmiş hesaplama cihaz mimarisi CUDA piyasaya sürülmüştür. CUDA, CPU ve GPU'nun birlikte çalışmasını destekleyen bir paralel programlama mimarisidir. CUDA ile GPU kullanımı çok daha verimli ve kolay hale gelmiştir (Yang et al., 2008).

Bu tez çalışması kapsamında NVIDIA CUDA mimarisi kullanılarak, görüntü işlemenin en temel fonksiyonları olan ve görüntülerin kontrastının arttırılmasını sağlayan histogram hesaplama ve eşitleme algoritmaları paralelleştirilmiştir. GPU üzerinde gri tonlamalı ve RGB renkli resimler kullanılarak deneyler yapılmış ve bu sayede GPU'ların nasıl programlandığı hakkında deneyim sahibi olunması amaçlanmıştır. Histogram hesaplama ve eşitleme algoritmalarının hem seri hem de paralel kodu geliştirilerek CPU ve GPU üzerinde performans ölçümleri yapılmıştır. Geliştirilen algoritmalar iki farklı GPU hesaplama mimarisi olan Fermi ve Maxwell üzerinde denenmiş ve farklı mimarilerin performansa etkisi incelenmiştir. Son olarak da çok iş parçacıklı ortamlarda karşımıza çıkan senkronizasyon probleminin GPU'lar üzerinde nasıl ele alındığı incelenmiştir.

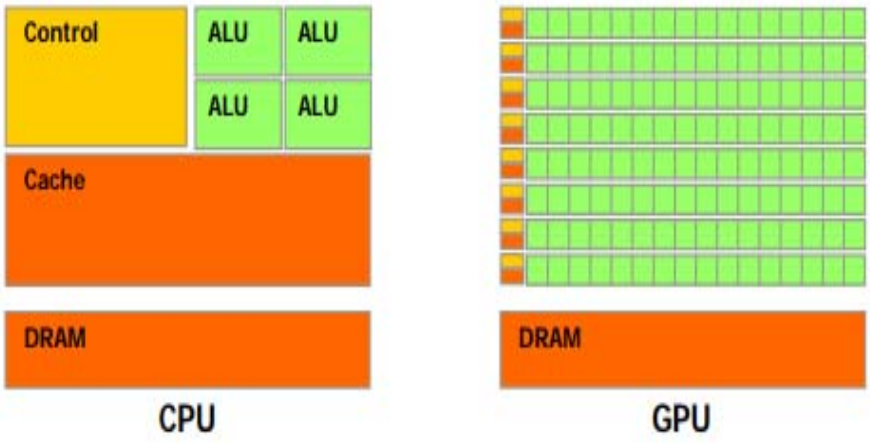
Bu tez çalışmasının diğer bölümlerinde sırasıyla, farklı GPU mimarileri ve CUDA mimarisi, histogram hesaplama ve eşitleme algoritmaları üzerinde yapılan paralelleştirme ve paralel programlamada yaygın bir problem olan senkronizasyonun GPU'lar üzerinde işlem yapılırken nasıl çözüldüğü hakkında bilgilere yer verilmiştir.

Tezin deney ve sonuçlar bölümünde CPU ve GPU üzerinde yapılan deneylerden çıkan sonuçlar gösterilmiş ve yorumlanmıştır. Son olarak çalışmanın sonuçları ve gelecekte yapılabilecek çalışmalarla ilgili öneri sunulmuştur.

2. GPU MİMARİSİ

İlk GPU NVIDIA tarafından 1999 yılında üretilmiştir ve 2003 yılından itibaren de grafiksel işlemlerin dışında yüksek derecede paralellik gerektiren görüntü işleme vb. uygulamalarda, yüksek performans ve verimlilik elde etmek için kullanılmaya başlanmıştır. Önceleri uzun yürütme sürelerine sahip olan görüntü uygulamaları üzerinde çalışma yapılması büyük problem oluştururken, şimdilerde GPU'nun yüzlerce paralel işlemci çekirdeğine sahip olması ile bu problem çözüme kavuşmuştur (NVIDIA, 2007).

Şekil 2.1' de GPU ve CPU mimarileri gösterilmektedir. Bu mimarilere bakıldığında, CPU mimarisinin çok az sayıda matematiksel ve mantıksal işlemleri destekleyen Aritmetik Mantık Birimi'nden (ALU) oluşurken büyük oranda kontrol ve önbellek alanlarına sahip olduğu, GPU'nun ise çok sayıda küçük ALU'ya sahip olduğu ve önbellek alanlarına daha az yer ayırdığı görülmektedir. GPU, bu sayede bellek erişimlerinden kaynaklanan gecikmelere maruz kalmadan çok daha fazla işlemci çekirdeğini kullanarak hızlı bir şekilde paralel hesaplamaları yapabilmektedir (NVIDIA, 2007).



Şekil 2.1 CPU'nun donanımsal mimari karşılaştırması (NVIDIA, 2007)

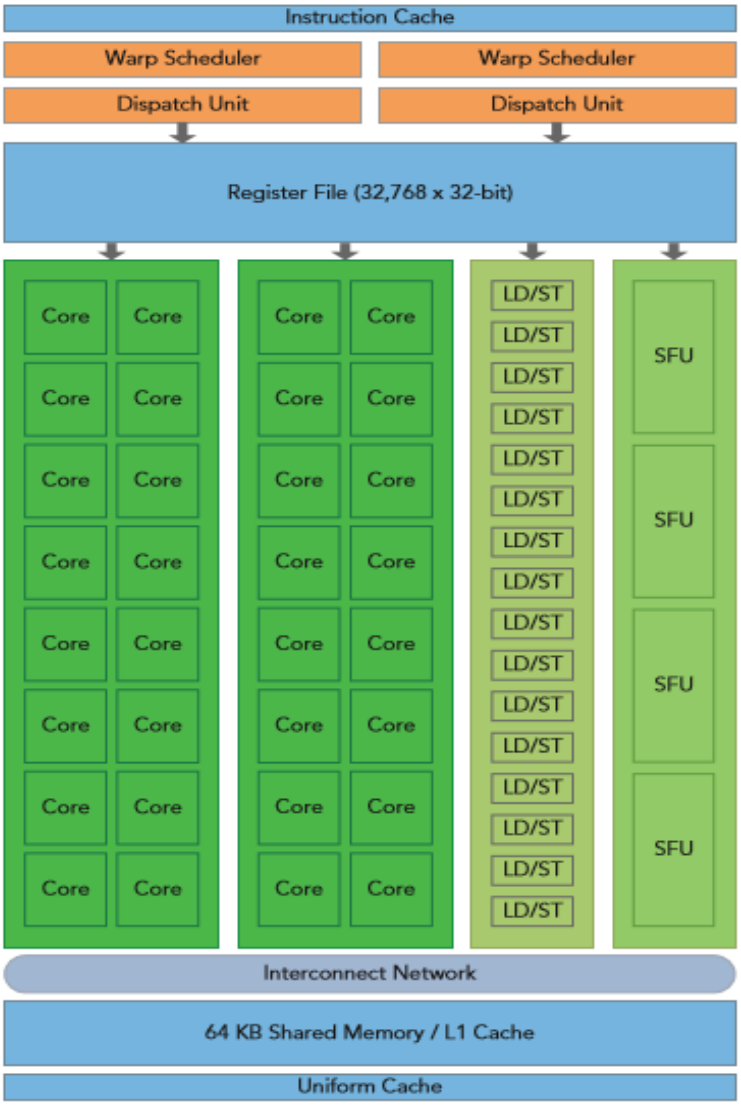
GPU'nun paralel hesaplama yeteneğinden doğru ve verimli bir şekilde yararlanılabilmesinin en önemli gereksinimlerinden biri de kullanılan GPU ve CUDA mimarisi hakkında gerekli bazı temel bilgilere sahip olunmasıdır.

2.1 Genel GPU Mimarisi

GPU mimarisinde birden fazla bağımsız iş parçacığı (thread) tek bir talimat (SIMT) kullanılarak aynı anda yürütülebilme kabiliyetine sahiptir. Her GPU

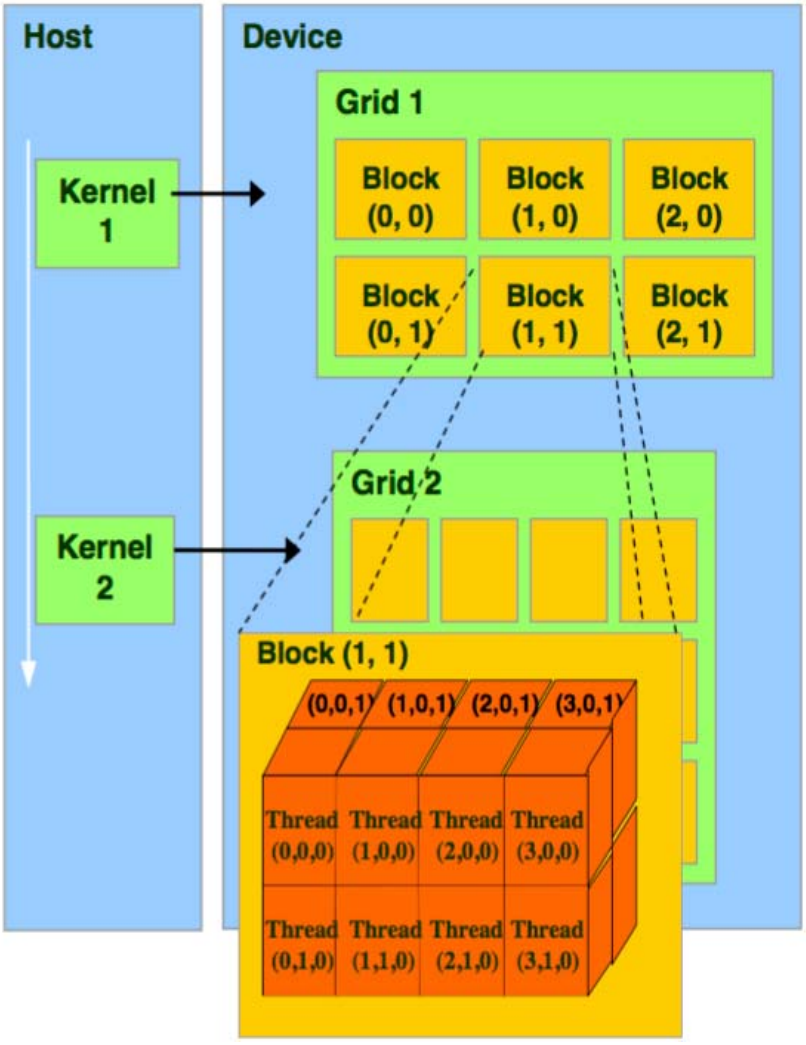
ölçeklenebilir sayıda akış işlemcisinden (SMs) oluşur. GPU içerisinde onlarca SM vardır, bu sayı GPU mimarisine ve versiyonuna göre değişiklik gösterir. Genel GPU mimarisinde, her SM'in içerisinde akış işlemcileri (SP) bulunur. SM'ler SP üzerinde çalışan yüzlerce iş parçacığının aynı anda yürütülmesini sağlar. Her SM aynı anda birden fazla iş parçacığı bloğunu tutabilir ve her biri paylaşılan belleğe ve kayıtçıya (register) sahiptir. Paylaşılan bellek ve kayıtçının boyutu küçük, fakat bu belleklere erişim çok hızlıdır. Bu kaynaklar aynı SM üzerinde bulunan iş parçacıkları arasında bölünmektedir. (Clua and Zamith, 2015).

SM'ler CUDA çekirdeklerinden, paylaşılan bellek, kayıtçı, yükleme ve saklama birimleri, sin(), cos() ve exp() gibi matematiksel işlemleri yapan özel fonksiyon birimi (SFU) ve çözgü zamanlayıcısı (warp schedular)'dan oluşur. Şekil 2.2' deki örnekte bir SM mimarisi gösterilmektedir.



Şekil 2.2 SM mimarisi (Cheng et al., 2014)

SM içerisindeki iş parçacıklarının birleşmesi ile oluşan gruba iş parçacığı blok'ları adı verilir. Bir blok'un içerisinde ki iş parçacıklarının sayısı sınırlıdır. Bu sayı mimarilere göre değişiklik gösterebilir (Clua and Zamith, 2015). İş parçacığı blok'larının birleşimi ile oluşan blok gruplarına ise grid adı verilir. Şekil 2.3' de bu grupların birleşmesiyle oluşan iş parçacığı, blok ve grid yapısı gösterilmektedir.



Şekil 2.3 İş parçacıkları, blok'lar ve grid arasındaki ilişki (New York University., 2017)

Her iş parçacığının ve blok'un kendilerine ait birer ID 'si vardır. Bu ID'ler, threadIdx, blockIdx'dir, bir boyutlu, iki boyutlu ve üç boyutlu olabilir. CUDA blok'ları ve grid'leri üç boyutta düzenler. Her bir boyuta vektör yapısının x, y, z bileşenleri ile erişilebilir. (Cheng et al., 2014)

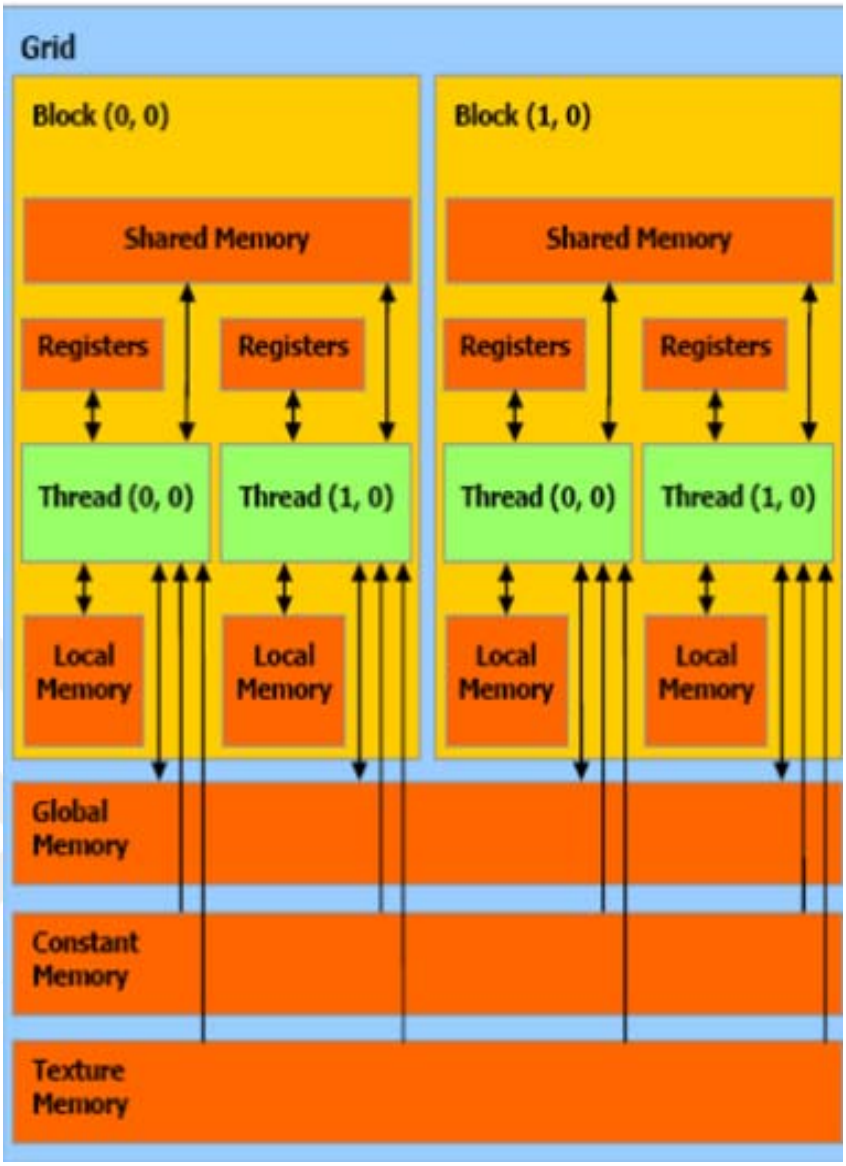
- **ThreadIdx,** iş parçacığının blok içerisindeki yerini belirtir.
- **BlockIdx**, blok'un grid içerisindeki yerini belirtir.

- **BlockDim**, bir blok'un boyutunu ve içerisinde çalışan iş parçacığı sayısını belirtir, bu boyut içerisindeki iş parçacığı sayısına göre değişir.
- **GridDim,** bir grid'in boyutunu ve blok sayısını belirtir. (Cheng et al., 2014)

Blok'lar warp schedular tarafından her biri 32 adet iş parçacığından oluşan warp'a bölünürler. Warp içerisindeki tüm iş parçacıkları aynı kodu yürütür ve bir warp içerisindeki her iş parçacığı kendi komut sayacına, durum kayıtçısına ve bağımsız yürütme yoluna sahiptir. Warp'lar dağıtıcı biriminin (Dispatch Unit) temelidir ve farklı iş parçacığı blok'larından oluşan iki warp eş zamanlı çalışabilme kabiliyetine sahiptir. SM üzerinde bir çok warp programlanabilir fakat SM'in kaynak kullana bilirliğine bağlı olarak tüm warp'lar aktif olmaz (NVIDIA, 2007; Paravecino, 2017).

Bir blok'daki tüm iş parçacıkları aynı SM üzerinde çalışır, bu nedenle aynı blok'da bulunan iş parçacıkları birbirleriyle işbirliği halindedir ve paylaşılan bellek aracılığı ile iletişim sağlarlar. Blok'lar ise birbirleriyle global belleği kullanarak iletişimi sağlarlar. (Paravecino, 2017).

GPU farklı belleklerden oluşur. Bu belleklerin hiyerarşik gösterimi şekil 2.4'de gösterilmektedir. Sırasıyla bu bellekler; kayıtçı (register), paylaşılan bellek (shared memory), yerel bellek (local memory), global bellek (global memory), sabit bellek (constant memory) ve doku belleği (texture memory)'dir.



Şekil 2.4 GPU içerisindeki bellekler (Ding., 2014)

- **Kayıtçı:** En hızlı bellek alanıdır ve her iş parçacığı için özeldir. Kernel bu bellek alanını sık erişilen iş parçacığına özel değişkenleri tutmak için kullanır. Kernel'in yürütülmesi tamamlandıktan sonra buradaki değişkenlere tekrardan ulaşılamaz (Cheng et al., 2014).

- **Paylaşılan Bellek:** Her SM için ayrılmıştır. Sadece aynı blok'da ki iş parçacıkları bu belleğe erişebilir. Paylaşılan belleğe erişim kayıtçı bellek alanına erişim kadar hızlıdır. Bu belleğin kullanımı ile global bellek erişiminden kaçınılabilir, paylaşılan belleğe erişim 4 saat döngüsü sürerken, genel belleğe erişim 400-600 saat döngüsü sürer. Bir blok bittiğinde paylaşılan bellek alanı silinir. (Clua and Zamith, 2015; Yang et al., 2008).

- **Yerel Bellek:** Bu belleğe sadece ait olduğu iş parçacığı tarafından ulaşılabilir. Buradaki veriler iş parçacığının çalışması sırasında saklanıp sonrasında silinir. Belleğe erişim, yüksek gecikme ve düşük bant genişliğindedir (Cheng et al., 2014; Clua and Zamith, 2015).

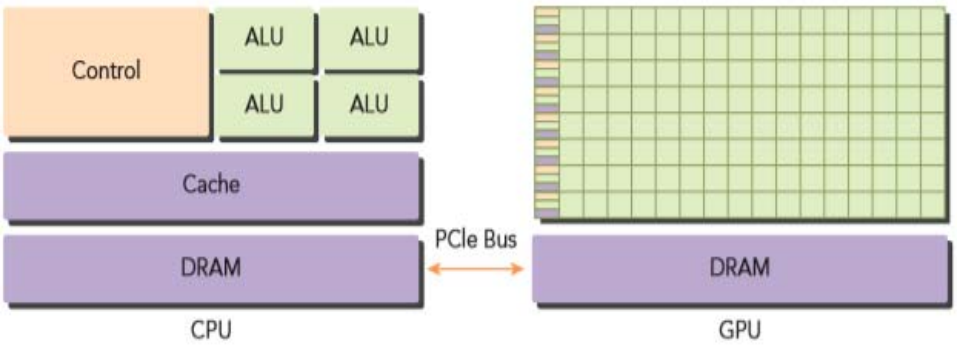
- **Global Bellek:** GPU'nun en çok kullanılan belleğidir, tüm iş parçacıkları tarafından bu belleğe erişim vardır. Yüksek gecikme süresine ve düşük bant genişliğine sahiptir. Yaşam süresi uygulamanın ömrü kadardır (Cheng et al., 2014; Clua and Zamith, 2015).

- **Sabit Bellek:** 64KB'lik küçük bir bellektir, sadece okuma işlemlerini gerçekleştirir. Aynı warp içerisindeki tüm iş parçacıklarının aynı bellek adresinden okuma yaptığı durumlarda en iyi performansı sağlar (Cheng et al., 2014; Clua and Zamith, 2015).

- **Doku Belleği:** Bir çeşit global bellektir. Sadece okuma işlemlerini gerçekleştirir. Önbellektedir, doku ve veri yumuşatılması gibi grafiksel işlemlerde kullanılır (Cheng et al., 2014; Clua and Zamith, 2015).

2.2 Heterojen Hesaplama

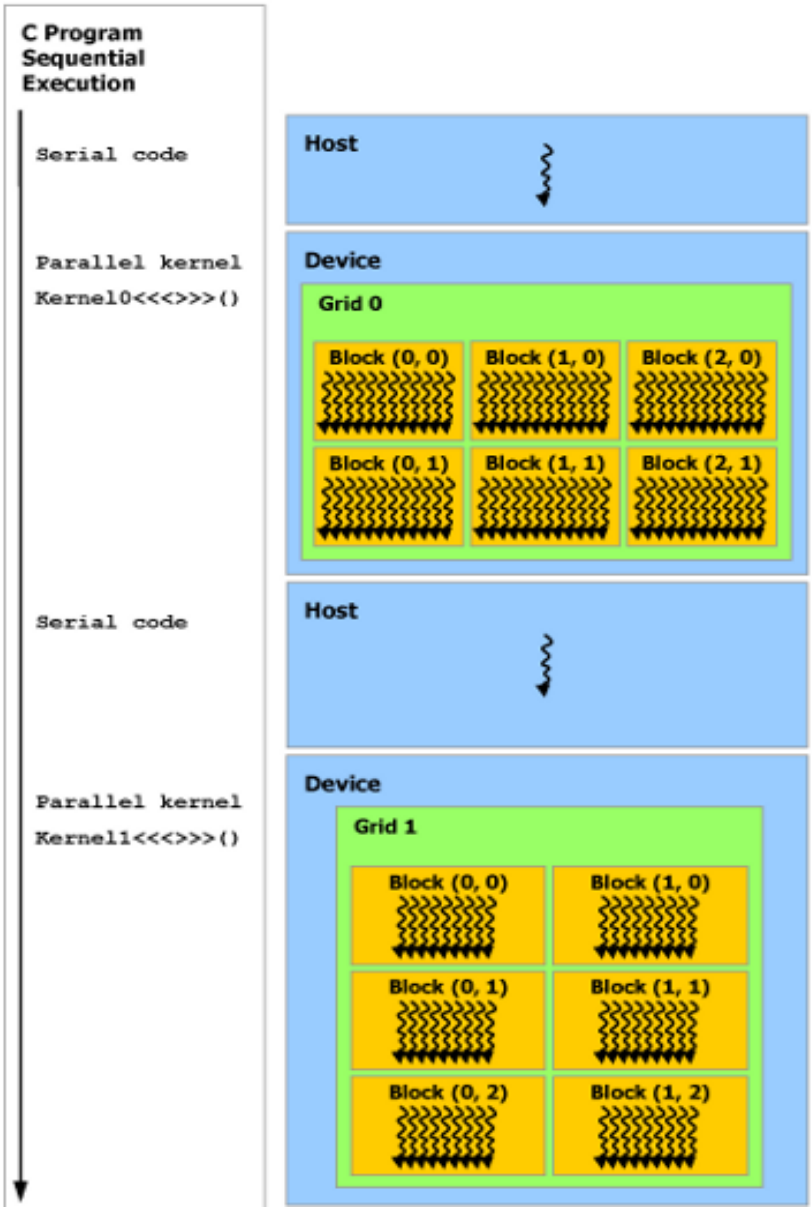
GPU tek başına çalışan bir işlemci değildir, CPU'nun yardımcı işlemcisi olarak çalışmaktadır. Hem CPU'nun hem de GPU'nun farklı program türleri için avantajları vardır. Bu nedenle, birbirlerini tamamlayıcı özelliklere sahiptirler. En iyi performansa birlikte kullanılmalarıyla ulaşılır. Bu kullanıma heterojen hesaplama adı verilmiştir. Şekil 2.5'de görüldüğü gibi CPU ve GPU donanımları arasındaki bağlantı PCI-Express veri yolu ile sağlanır. Bu hesaplamada, CPU ana bilgisayar (host), GPU ise aygıt (device) olarak adlandırılır (Cheng et al., 2014).



Şekil 2.5 CPU ve GPU'nun birleşimi (Cheng et al., 2014)

CPU ve GPU'nun heterojen bir şekilde kullanımında, uygulama CPU tarafından başlatılır. CPU, ortamın, kodların ve verilerin yönetiminden sorumludur. Veri büyüklüğü küçük ve düşük seviye paralellik gerektiren seri algoritmalar yürütülür. GPU'lar ise, hesaplamanın yoğun olduğu ve büyük oranda paralellik gerektiren bölümlerin hızlandırılması için kullanılır. Bu durumda, GPU'lar donanım hızlandırıcısı olarak düşünülebilir (Cheng et al., 2014).

Heterojen programlamanın kullanımını kolaylaştıran en önemli etkenlerden biri de NVIDIA'nın GPU'larda ki paralel hesaplamayı kullanan CUDA programlama modelini çıkartmasıdır. CUDA programı ana makine kodu (host) ve cihaz kodundan (device) oluşur. Şekil 2.6'da heterojen programlama modelinde bir kernel fonksiyonunu çağırdığımızda sırasıyla yapılan işlemler gösterilmektedir. Şekilde görüldüğü gibi derleme sırasında device kodu host kodundan ayrılır. GPU üzerinde çalışan paralel koda çekirdek (kernel) adı verilmiştir ve C programlama kullanılarak yazılabilir, kernel kodu CPU tarafından çağırılır ve kernel'in çağrılmasının ardından device çalışmaya başlar. Kernel GPU üzerinde yürütülürken programın geriye kalan C kodları CPU üzerinde çalıştırılır. Device'ın kernel fonksiyonunu çalıştırmasını bitirmesinin ardından kontrol tekrar CPU'ya geçer ve CPU yeni kernel fonksiyonunu çağırır (NVIDIA, 2018; Cheng et al., 2014).



Şekil 2.6 Heterojen programlama modeli (NVIDIA, 2018)

Kernel'ler bir dizi iş parçacığı tarafından çalıştırılır ve her biri aynı kodu paralel olarak çalıştırır. İş parçacığı bir döngüdeki bir yineleme gibi düşünülebilir (Cheng et al., 2014). Heterojen hesaplamada iş akışı genel olarak aşağıdaki gibidir;

- İlk olarak program CPU'daki verilerin hazırlanmasıyla başlar.
- Veriler CPU'dan GPU'nun genel belleğine kopyalanır.
- İş parçacıkları global bellekte ki verileri okuyarak yerel belleklerine yazarlar. GPU, bu veriler üzerinde çalışıp, hesaplamaları tamamlar.
- Son olarak sonuç global belleğe geri yazılır. (Cheng et al., 2014; Eklund et al., 2013)

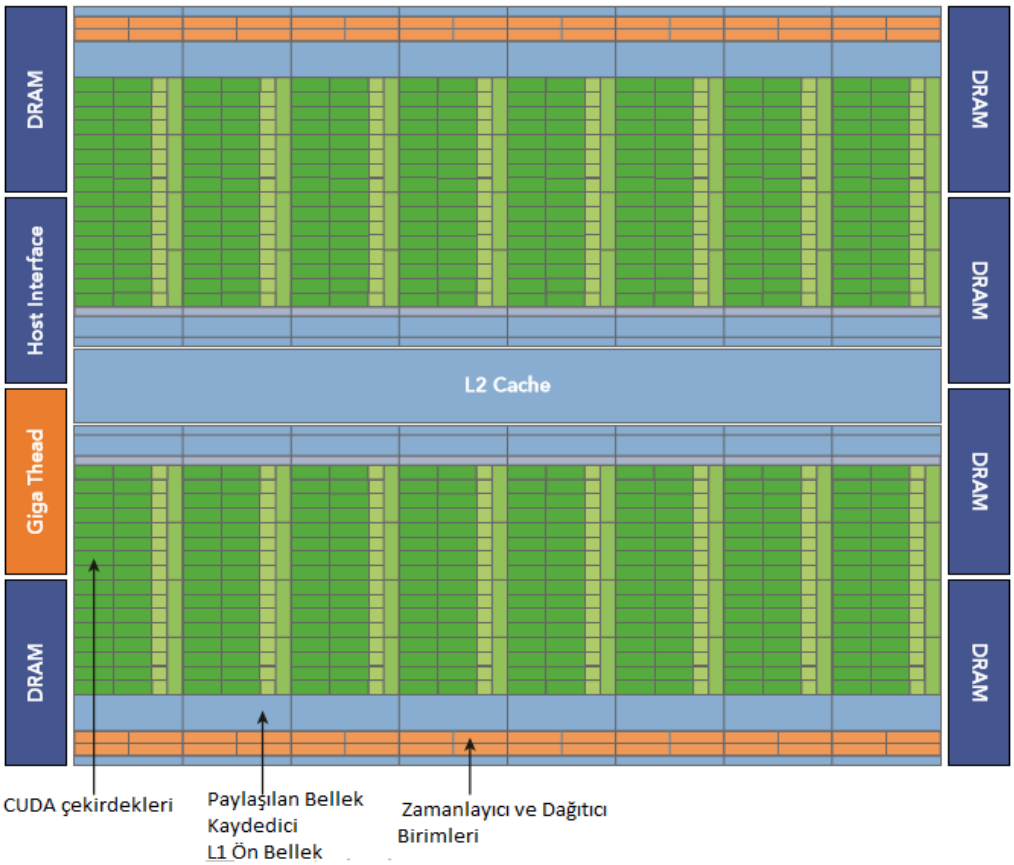
2.3 Fermi ve Maxwell Mimarisi

Bu tez kapsamında iki farklı GPU mimarisi kullanılmıştır bunlar Fermi ve Maxwell Mimarileri'dir. NVIDIA, GPU donanım versiyonlarını tanımlamak için hesaplama kapasitesi terimini kullanmaktadır. Bu çalışmada kullanılan mimarilerden Fermi küçük hesaplama kapasitesine (2.x), Maxwell ise yüksek hesaplama kapasitesine (5.x) sahiptir.

2.3.1 Fermi mimarisi

Fermi Mimarisi 16 SM'den oluşur ve her SM'de 32 CUDA çekirdeği olacak şekilde toplam 512 hızlandırıcı çekirdek SM üzerine yerleştirilmiştir. Her bir CUDA çekirdeği ardışık olarak dizilmiş ALU'ya ve FPU'ya sahiptir. Bir SM'de 16 yük/depo (load/store) birimi vardır, bu birim bir warp içerisindeki iş parçacığı sayısının yarısı kadar yani 16 iş parçacığı için kaynak ve hedef adreslerinin hesaplanmasını sağlar. Ayrıca SM'de 4 SFU bulunur, SFU'lar sinüs, cosinüs ve karekök gibi talimatları yönetirler (Cheng et al., 2014).

Aşağıda şekil 2.7'de görüldüğü gibi bu mimaride SM'ler L2 önbelliği etrafında konumlandırılmışlardır ve tüm SM'ler bu belleği paylaşırlar. Ayrıca her SM'in 2 adet çözgü zamanlayıcısı (warp schedular) ve 2 dağıtıcı birimi (dispatcher unit), paylaşılan belleği, 64 KB'lik kayıtçısı ve L1 önbelleği vardır. Zamanlayıcı ve dağıtıcı birimler SM'e bir iş parçacığı bloğu atandığında 32'lik warp'lara bölünür ve çözgü zamanlayıcıları 2 warp seçer ve her warp'tan bir talimatı 16 CUDA çekirdeğine, yük/depo birimine ve 4SFU'ya bildirir. Yine şekilde 2.7'de görüldüğü gibi bu mimaride 1 GigaThread motoru (engine) ve toplam 6GB'lık global bellek desteği sağlayan 6 adet DRAM bulunmaktadır. GigaThread motoru iş parçacığı bloklarını SM'deki warp zamanlayıcılarına dağıtmada görev alır (Cheng et al., 2014).

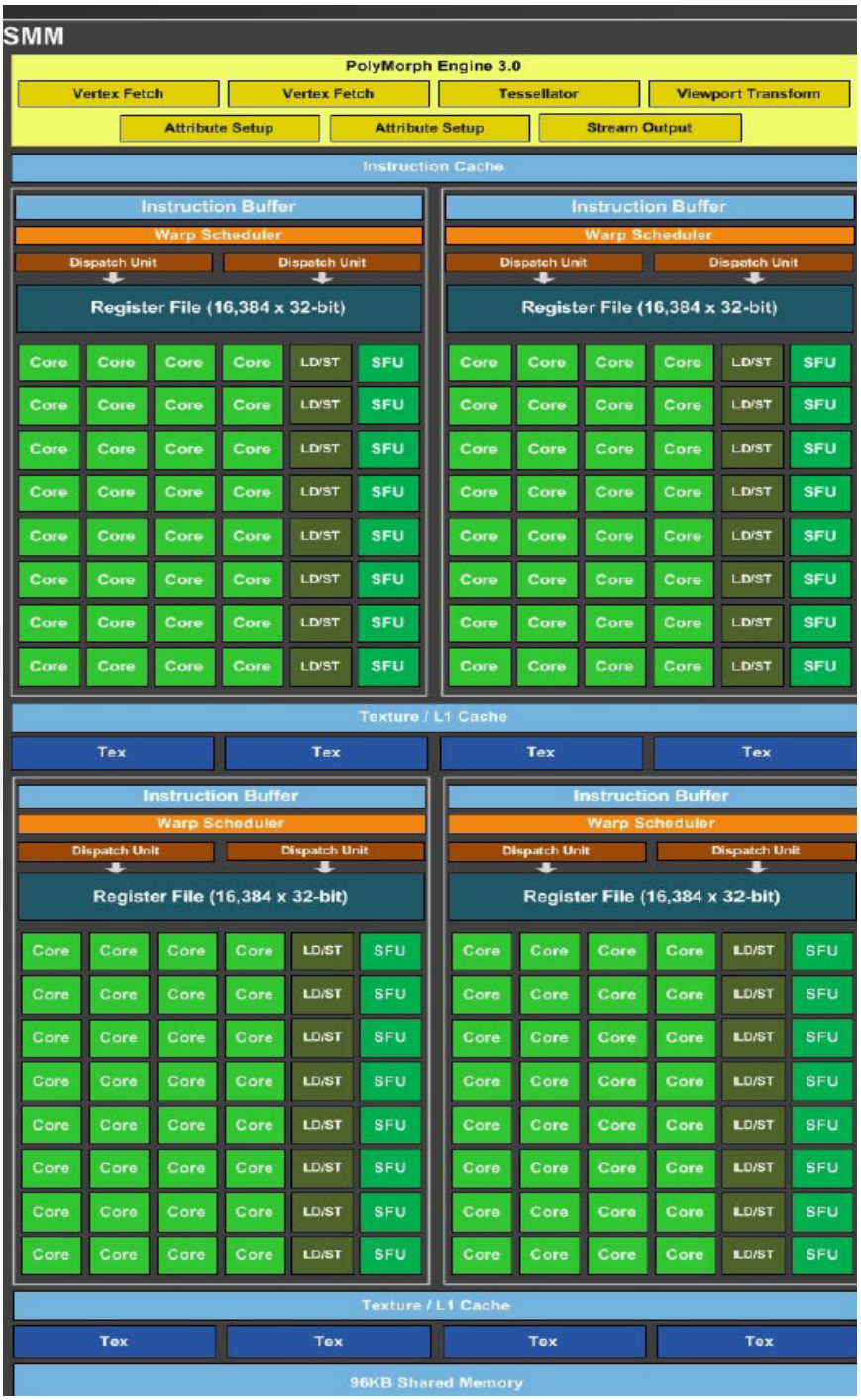


Şekil 2.7: Fermi Mimarisi (Cheng et al., 2014)

2.3.2 Maxwell mimarisi

Maxwell Mimarisi'nde önceki mimarilere göre güç verimliliğini daha çok arttırmak için yeni bir SM tasarımı yapılmıştır. Diğer mimarilere göre her SM daha az sayıda CUDA çekirdeğinden oluşur. Bu yeni SM mimarisi daha küçük akış çok işlemcisi (SMM) olarak adlandırılmıştır.

Şekil 2.7'de görüldüğü gibi her SMM yapısı 4 özdeş alt yapıdan oluşur. Her birinde 32 çekirdek, 8LD/ST birimi, 8SFU ve 16K kayıtçı bulunur ve SMM başına düşen CUDA çekirdeği 128'dir, her saat dögüsünde iki talimat gönderen 4 warp zamanlayıcısı, 8 doku birimi (texture unit) ve 8 talimat dağıtıcı birimden oluşur. Bu mimarideki en önemli değişimlerden biri de bellek hiyerarşisidir. Paylaşılan bellek için özel bir alan ayrılmıştır ve L1 önbelleği doku ön belleği ile birleştirilmiştir. Diğer mimarilerde bu alan L1 önbelleği ve paylaşılan bellek arasında bölünüyordu (Harris, 2014; Paravecino, 2017).



Şekil 2.8:Maxwell SMM Mimarisi (Clua and Zamith, 2015)

Çizelge 2.1'de tez çalışmasında kullanılan GPU mimarilerinin özellikleri gösterilmiştir. Küçük hesaplama kapasitesine sahip mimarilerin farklı versiyonlarında güncellemelere bağlı olarak bazı değişiklikler bulunmaktadır. Bu nedenle yukarıda bahsedilen genel Fermi Mimarisi'ne göre kullanılan GPU mimarisinde, örneğin; CUDA çekirdek sayısı farklılık göstermektedir.

	GeForce GT 540M (Fermi)	GeForce GTX 960M (Maxwell)
Cuda Çekirdeği	96	640
Bellek Hızı (Mhz)	900	2500
Bellek Arayüzü	128-bit DDR3	128-bit GDDR5
Bellek Bant Genişliği	28.8	80
CUDA Kapasite Versiyonu	2.1	5.0
Toplam Global Bellek	2.0 Mbytes	4.0 Mbytes
L2 Cache Boyutu	131072 bytes	2097152 bytes
Toplam Sabit Bellek	65536 bytes	65536 bytes
Blok Başına Paylaşılan Bellek Toplamı	49152 bytes	49152 bytes
Blok Başına Toplam Kayıtçı Sayısı	32768	65536
Warp Boyutu	32	32
Her Bir Çok İşlemcili Başına Maksimum İş Parçacığı Sayısı	1536	2048
Her Blok Başına Maksimum İş Parçacığı Sayısı	1024	1024
Bir Bloğun Her Boyutunun Maksimum Büyüklüğü	1024 x 1024 x 64	1024 x 1024 x 64
Bir Grid'in Her Boyutunun Maksimum Büyüklüğü	65535 x 65535 x 65535	2147483647 x 65535 x 65535
Maksimum Bellek Aralığı (pitch)	2147483647 bytes	2147483647 bytes

Çizelge 2.1 Kullanılan Fermi ve Maxwell Mimarisi'nin özellikleri

3. CPU VE GPU'DA GÖRÜNTÜ İŞLEME

Görüntü uygulamalarının genellikle büyük boyutta görüntüleri kullanması ve hesaplama yoğunluğunun yüksek olması nedeniyle geleneksel görüntü işleme yöntemleri istenilen performansı sağlayamamaktadır. Yüksek veri paralelliğinin gerektiği bu durum için GPU kullanımı en ideal çözümlerden biridir (Zhang et al., 2010).

Bu tez çalışmasında da görüntü işlemenin en temel fonksiyonlarından histogram hesaplama ve eşitleme kullanılarak CPU üzerinde seri ve GPU üzerinde paralel görüntü kontrastı iyileştirme programı geliştirilmiştir. Geliştirilen seri ve paralel görüntü işleme programları 2 farklı işlemci üzerinde çalıştırılarak deney sonuçları elde edilmiştir.

3.1 Histogram Hesaplama ve Eşitleme

Histogram hesaplama ve eşitleme görüntü işlemenin birçok alanında kullanılan en temel fonksiyonlardandır. Histogram ile bir görüntüde farklı yoğunluklardaki piksel sayısını elde ederiz. Histogram eşitleme ile de görüntülerin histogramından elde ettiğimiz piksel yoğunluklarını kullanarak görüntü tonlarının tüm görüntü boyunca eşitlenmesi ile kontrastın iyileştirilmesini sağlarız. Histogram eşitleme birçok görüntü ve video işleme uygulamasında ön işleme basamağı olarak kullanılır (Wawud et al., 2017). Şekil 3.1'de geliştirdiğimiz görüntü işleme uygulaması ile yapılmış kontrast iyileştirme örneği gösterilmektedir.

(a) (b)

Şekil 3.1 a) Orijinal resim, b) Orijinal resme histogram eşitletme uygulandıktan sonra

3.2 CPU'da Histogram Hesaplama ve Eşitleme

Bu tez çalışmasında seri algoritma kodlarının geliştirilmesi için C programlama dili kullanılmıştır. Histogram bir dizi (array) olarak temsil edilir ve histogram dizisi her bir öğesi bin olarak adlandırılan kutulardan birine karşılık gelir ve bu bin'lerden her birine düşen piksel sayısını içerir. Her renk kanalı için 256 bin tanımlanır (Sakharnykh 2015; Milic et al. 2013). Bir görüntünün her pikselinin 0 ile 255 arasında bir değere sahip olduğu varsayılır, piksel değeri bu aralığa uyduğu zaman bin değeri bir arttırılır.

Seri histogram hesaplama, paralel histogram hesaplamaya göre oldukça basittir. Şekil 3.2'de basit olarak kullandığımız seri histogram algoritması gösterilmektedir. Kullandığımız bu algoritmada hist değeri sonuç histogramımızı tutan bir dizidir, img değeri görüntü girdisidir, img_size görüntümüzün boyutunu tutar, nbr_bin değeri ise renk kanalı için kullandığımız bin değerini tutar.

```
void histogram_cpu(int * hist, unsigned char *img, int img_size,
int nbr_bin){
    int i;
    for ( i = 0; i < nbr_bin; i ++){
        hist[i] = 0;
    }

    for ( i = 0; i < img_size; i ++){
        hist[img[i]] ++;
    }
}
```

Şekil 3.2 Seri histogram hesaplama algoritması

Şekil 3.3'de seri olarak bir girdi görüntüsünün histogram eşitleme fonksiyonu gösterilmektedir. Bu fonksiyon hem gri ölçekli görüntülerde hem de RGB (kırmızı, yeşil, mavi) renk görüntülerde kullanılmıştır. Fakat, RGB renk görüntüleri üç renk kanalından oluşmaktadır, ve eğer görüntüyü üç ayrı renk kanalına bölerek her bir kanal için ayrı histogram eşitleme uygulanırsa doğru sonuçlar elde edilemez. Bunun için renk bileşenlerini yoğunluk değerlerinden ayıran bir renk uzayı kullanılmalıdır. Bu tez çalışmasında YUV renk uzayı kullanımı tercih edilmiştir. YUV uzayının Y bileşeni rengin parlaklığını belirlerken U ve V bileşenleri ise rengin doygunluğunu ve tonunu belirler.

Şekil 3.3.'de RGB renkli görüntü için kullandığımız kontrast iyileştirme fonksiyonumuz gösterilmektedir. Burada ilk olarak giriş görüntümüzü RGB renk uzayından YUV uzayına dönüştürülür, sonrasında Y bileşeninin histogramı çıkarılır ve elde edilen Y bileşenin histogramı ile histogram eşitleme yapılır bu işlem sonucunda yeni Y bileşenimiz ile U ve V bileşenleri birleştirilir. Son olarak da YUV uzayından tekrar RGB uzayına dönüştürme yapılarak sonuç görüntümüz elde edilir.

```
PPM_IMG contrast_enhancement_rgb(PPM_IMG img_in)
{
    PPM_IMG result;
    YUV_IMG yuv_med;
    int hist[256];
    unsigned char * y_equ;

    yuv_med = rgb2yuv(img_in);
    y_equ = (unsigned char *)malloc(yuv_med.h*yuv_med.w*sizeof(unsigned char));
    histogram(hist, yuv_med.img_y, yuv_med.h * yuv_med.w, 256);
    histogram_equalization(y_equ,yuv_med.img_y,hist,yuv_med.h * yuv_med.w, 256);

    free(yuv_med.img_y);
    yuv_med.img_y = y_equ;
    result = yuv2rgb(yuv_med);

    free(yuv_med.img_y);
    free(yuv_med.img_u);
    free(yuv_med.img_v);

    return result;
}
```

Şekil 3.3 CPU'da RGB renk resim için kontrast iyileştirme fonksiyonu

Şekil 3.4'de gösterilen histogram eşitleme fonksiyonumuzda önceden hesaplanmış olan giriş görüntüsünün histogram değeri hist_in ile alınır ve bu fonksiyon içerisinde elimizdeki girdi görüntüsünün histogramını kullanarak görüntünün yeni parlaklık değerleri elde edilir.

```
void histogram_equalization(unsigned char * img_out, unsigned char * img_in,
                            int * hist_in, int img_size, int nbr_bin){
    int *lut = (int *)malloc(sizeof(int)*nbr_bin);
    int i, cdf, min, d;
    cdf = 0;
    min = 0;
    i = 0;

    while(min == 0){
        min = hist_in[i++];
    }

    d = img_size - min;
    for(i = 0; i < nbr_bin; i ++){
        cdf += hist_in[i];
        lut[i] = (cdf - min)*(nbr_bin - 1)/d;
        if(lut[i] < 0){
            lut[i] = 0;
        }
    }

    /* sonuc resmini elde etme */
    for(i = 0; i < img_size; i ++){
        if(lut[img_in[i]] > 255){
            img_out[i] = 255;
        }
        else{
            img_out[i] = (unsigned char)lut[img_in[i]];
        }

    }
}
```

Şekil 3.4 CPU'da görüntü histogram eşitleme fonksiyonu

Görüntünün yeni parlaklık deperinin elde edilmesi için öncelikle şekil 3.5'deki formülü kullanılarak histogramın kümülatif dağılım fonksiyonunu hesaplarız.

$$cdf(i) = \sum_{j=0}^{i} n_j$$

Şekil 3.5 Kümülatif Dağılım Fonksiyonu Hesaplama Formülü (Gaura, 2016)

Kümülatif dağılım fonksiyon (cdf) değerinin hesaplanmasının ardından ise yeni parlaklık değerimizi şekil 3.6.'de ki formülü kullanarak hesaplarız.

$$\frac{cdf - cdf_{\min}}{(\text{width} \times \text{height}) - cdf_{min}} \times (L-1)$$

Şekil 3.6 Parlaklık Değeri Hesaplama Formülü (Gaura, 2016)

Buradaki cdf_{min} değeri kümülatif dağılım fonksiyonun sıfır olmayan en küçük değeridir. L değeri parlaklık seviyesinin sayısını (nbr_bin) belirtir. Son olarak oluşturduğumuz arama tablosunda (lut) hesapladığımız yeni parlaklık değerleri saklanır ve en son bu tabloya göre görüntü tekrardan güncellenerek sonuç görüntümüz elde edilir.

3.3 GPU'da Histogram Hesaplama ve Eşitleme

GPU üzerinde çalıştırılacak olan paralel algoritma kodları geliştirilirken C programlamanın yanında NVIDIA CUDA platformu kullanılmıştır. Paralel kodlama yapılırken dikkat edilmesi gereken kısımlar seri kodlamaya göre çok daha fazladır. Bu nedenle paralel algoritmaların implementasyonu seri algoritmalara göre oldukça zordur. Bir paralel algoritma implementasyonunda kullanılacak iş parçacığı sayısı, blok sayısı ve paylaşılan bellek kullanımı gibi birçok konu göz önünde bulundurulmalıdır.

Şekil 3.7'de gri ölçekli görüntülerde kullandığımız GPU üzerinde çalıştırılan kontrast iyileştirme fonksiyonu gösterilmektedir. Buradaki fonksiyonun aynısı RGB renk görüntüler içinde aynı şekilde uygulanır fakat bölüm 3.2'de anlatılan CPU üzerinde kontrast iyileştirmede kullanılan YUV uzayına dönüştürme aynı şekilde burada da yapılır ve Y bileşeni üzerinden histogram hesaplama ve eşitleme işlemleri uygulanır. Şekil 3.7'de ki GPU üzerinde CUDA C ile kontrast iyileştirme algoritmamızın implementasyonu yapılırken şu adımlar izlenilmiştir; öncelikle aygıt tarafında paralel olarak yürütülecek kernel fonksiyonlarımız başlatılmadan önce ana makina (host)'dan aygıta (device) kullanılacak girdi verileri kopyalanır. Bu işlem için CUDA'nın cudaMemcpy() fonksiyonunu kullanılır. Sonrasında C programlamada kullanılan memset() fonksiyonu ile aynı işleve sahip olan cudaMemset() ile cdf ve histogram'a 0 değerini veririz. Ardından kernel fonksiyonları çağırılmadan önce yürütülme konfigürasyonları yaparak, GPU'da iş parçacıklarının nasıl çalıştırılacağını belirleriz.

<<<blok sayısı , iş parçacığı sayısı>>> kernel fonksiyonlarının çağrılma şeklidir, buradaki ilk değer grid içerisinde başlatılacak blok sayısını, ikinci değer ise her bir blok içerisindeki iş parçacığı sayısını belirtir. Bu tez çalışmasında iş parçacığı ve blok sayıları belirlenirken blok başına 256 iş parçacığı kullanılmıştır bunun nedeni histogram hesaplamada bin değerinin 256 olmasıdır. İş parçacığı, blok ve grid sayılarının uygun olarak verilmesi büyük öneme sahiptir, aksi durumda uygulama doğru sonuçlar vermemektedir. Bu işlemlerin ardından

kernellerin çalışması bittiğinde son olarak cudaMemcpy() ile çıkış verileri aygıttan ana makinaya tekrardan kopyalanır.

```
PGM_IMG gpu_contrast_enhancement_g(PGM_IMG img_in)
{
   PGM_IMG result;
   unsigned char * d_result;
   unsigned int * histo;
   float * cdf;

   result.w = img_in.w;
   result.h = img_in.h;
   result.img = (unsigned char *)malloc(result.w * result.h * sizeof(unsigned char));

   cudaMalloc(&d_result,result.w * result.h * sizeof(unsigned char));
   cudaMalloc((void **)&histo, 256*sizeof(unsigned int));
   cudaMalloc((void **)&cdf, 256*sizeof(float));

   cudaMemcpy(d_result, img_in.img, result.w * result.h * sizeof(unsigned char), cudaMemcpyHostToDevice);
   cudaMemset(histo, 0, HISTOGRAM_LENGTH*sizeof(unsigned int));
   cudaMemset(cdf, 0, HISTOGRAM_LENGTH*sizeof(float));

   int numThreads = 256;
   int blockSize = (result.w * result.h - 1)/256 + 1;

   histogram_gpu<<<blockSize,numThreads>>> ( histo, d_result, result.w * result.h);
   calcCDF << <blockSize,numThreads >> >(cdf, histo, result.w * result.h, 256);
   histEqualize << <blockSize,numThreads >> >(d_result, cdf, result.w * result.h);

   cudaMemcpy(result.img, d_result, result.w * result.h * sizeof(unsigned char), cudaMemcpyDeviceToHost);
   cudaFree(d_result);
   return result;
}
```

Şekil 3.7 GPU'da Kontrast İyileştirme Fonksiyonu

3.4. GPU Üzerinde Paylaşılan Bellek Ve Global Bellek Kullanımı

Bölüm 2.1'de bahsedildiği gibi paylaşılan bellek SM'de bulunan en hızlı belleklerdendir erişim gecikmesi çok düşüktür (4 saat çevrimi kadar) ve yüksek bant genişliğine sahiptir. Global bellek ise GPU'nun ana ve en çok kullanılan belleğidir. Bu belleğe erişim gecikmesi çok daha yüksektir (400-600 saat çevrimi). Paylaşılan belleğin ömrü iş parçacığı bloğu kadardır ve sadece aynı blok içerisindeki iş parçacıkları ulaşabilirken, global belleğin ömrü uygulamanın ömrü kadardır ve tüm çekirdeklerin (kernel) iş parçacıkları tarafından erişilebilir. Global belleğe erişimin olabildiğince az yapılması istenir, CUDA programlamada aynı blok içerisindeki iş parçacıkları tarafından birden fazla defa global bellekten çekilmiş bir veriye erişilmesi gereken durumlarda aynı verilere yeniden global bellekten erişilmesindense veriler paylaşılan belleğe kopyalanarak kullanılır. Böylece uzun gecikme süresinden kaçınılır (Nickolls et al., 2008;Cheng, 2014).

Bu tez çalışmasında da GPU üzerinde histogram hesaplama yapılırken hem global bellek hem de paylaşılan bellek kullanılmış ve performansları incelenmiştir.

3.4.1 Histogram hesaplamada paylaşılan bellek kullanımı

Paylaşılan bellek ile histogram hesaplama yapılırken öncelikle her bloğun histogramını tutmak için paylaşılan arabellek alanı ayrılır. Paylaşılan bellek dizisinde ki her bir eleman değeri için 0 verilir. Bu işlem sonrasındaki adımda paylaşılan bellekten değerler okunup değiştirileceği için bir bloktaki tüm iş parçacığı için işlemin tamamlandığından emin olunması gerekir. Bunun için _syncthreads() çağrısını kullanırız.

Senkronizasyon çağrısının kullanım nedeni iş parçacıkları arasında veri paylaşımı yapıldığı zamanlarda yarış koşulları (race condition) meydana gelmesidir. Yarış koşullarının oluşma nedeni bir blok içerisindeki iş parçacıklarının mantıksal olarak paralel çalışmasına rağmen aslında tüm iş parçacıklarının fiziksel olarak aynı anda çalıştırılamamalarıdır. Paylaşılan bellek aynı anda erişilebilen eşit boyutlu 32 bellek modülüne (bank) ayrılır. 32'ye bölünme nedeni bir warp içerisinde 32 iş parçacığı bulunmasıdır.

Bir bloktaki iş parçacıkları aynı paylaşılan bellek adresine erişebilir. Ortaya çıkan bu durum bellek konumlarında tanımlanmamış davranışlara sebep olabilir. İş parçacıklarının düzgün bir şekilde çalışmasını ve doğru sonuçların elde

edilmesini sağlamak için CUDA'nın sağladığı en basit senkronizasyon biçimi olan ve bir çeşit bariyer görevi gören _syncthreads() kullanılır. Bir iş parçacığı bu çağrıya ulaştığında bloğun içindeki tüm iş parçacıklarının aynı senkronizasyon noktasına erişmesini bekler. _syncthreads()'ın aynı zamanda bir işlevi de aynı blok içerisindeki iş parçacıkları arasındaki iletişimi koordine etmektir (Cheng et. al., 2016; Gupta 2013).

Paylaşılan bellek ayrıldıktan sonra görüntünün her pikseli için uygun bin değeri bulunduğunda ilgili sayaç arttırılır. Fakat paralel olarak histogram hesaplama yapılırken aynı histogram bin değerini birden fazla iş parçacığı aynı anda arttırmak isteyebilir buna engel olmak için CUDA'nın atomik fonksiyonlarından atomicAdd() kullanılmıştır.

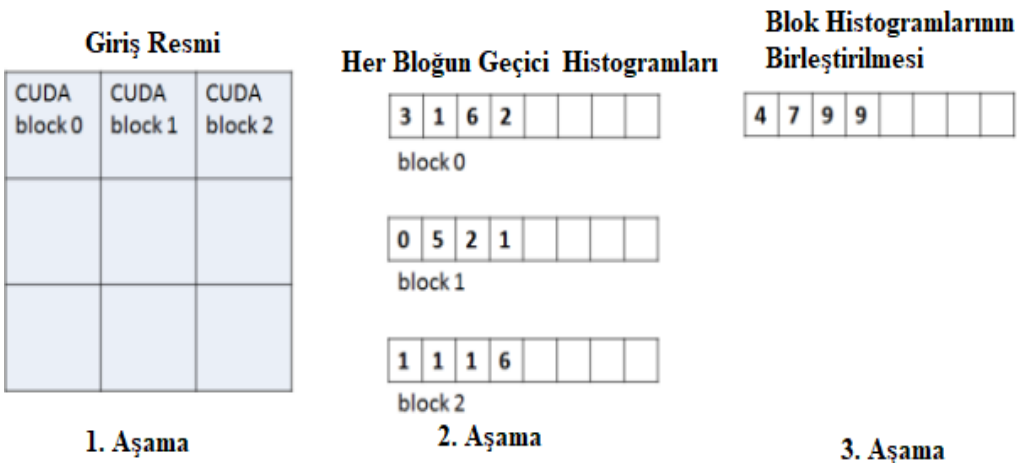
Atomik fonksiyonlar, bir iş parçacığının, diğer iş parçacıklarının müdahalesi olmadan bellek işlemini kesintisiz bir şekilde gerçekleştirebildiği matematiksel işlemlerdir. Yüzlerce iş parçacığı tarafından paylaşılan değerlerle çalışılması için güvenli bir yoldur. Paralel iş parçacıkları arasındaki bellek erişiminin senkronizasyonunu sağlayarak iş parçacıklarının birbirine müdahalesini engellediğinden yarış koşulunun meydana gelmesini önler (Cheng et. al., 2016).

Şekil 3.8'de görüldüğü gibi histogram_priv değeri atomicAdd() fonksiyonu ile başka bir iş parçacığı tarafından kesintiye uğramadan bellek adresindeki değeri okuyup değeri bir arttırır ve sonucu tekrar bellek adresine yazar. Bloktaki tüm iş parçacıkların aynı işlemi bitirdiğinden emin olunabilmesi için tekrardan _syncthread() fonksiyonu kullanılır. Tüm bu işlemlerin sonunda paylaşılan bellekteki sonuç değerleri global belleğe kopyalanır.

```
__global__ void histogram_kernel (unsigned int *histogram, unsigned char
*grayimg, unsigned int size ) {

  __shared__ unsigned int histogram_priv[256];

  int tx = threadIdx.x;
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int offset = blockDim.x * gridDim.x;

  if (tx < 256) {
    histogram_priv[tx] = 0;
  }
  __syncthreads();

  while (i < size) {
    atomicAdd( &(histogram_priv[grayimg[i]]), 1);
     i += offset;

  }
  __syncthreads();

  if (tx < 256){
    atomicAdd( &(histogram[tx]), histogram_priv[tx]);
  }
 }
```

Şekil 3.8 GPU'da paylaşılan bellek kullanımı ile histogram hesaplama

Şekil 3.9' de paralel histogram algoritmasının bir görüntünün histogramını hesaplama adımlarını şematik olarak göstermektedir.



Şekil 3.9 GPU'da bir resmin paralel hesaplama ile histogramını hesaplama aşamaları (Sakharnykh, 2015)

3.4.2 Histogram hesaplamada global bellek kullanımı

Şekil 3.10'de gösterilen global bellekte histogram hesaplama yapan kernel fonksiyonu, seri histogram hesaplama fonksiyonu ile oldukça benzerdir. Buradaki fonksiyonun seri versiyonundan farkı atomicAdd() kullanımıdır. Diğerinden farklı olarak atomicAdd() kullanım nedeni ise bir önceki bölümde bahsedilen birden fazla iş parçacığının aynı bin değerini arttırmak istemesiyle ortaya çıkacak problemleri önlemek içindir.

```
__global__ void histogram_kernel_gmem (unsigned int *histogram, unsigned char *grayimg,
                                       unsigned int size ) {

  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * gridDim.x;
  while (i < size)
  {
          atomicAdd( &(histogram[grayimg[i]]), 1 );
              i += stride;
  }
}
```

Şekil 3.10 GPU'da global bellek kullanımı ile histogram hesaplama

Şekil 3.11 ' de histogram eşitleme de yeni renk değerinin hesaplanması için kullanılan cdf hesaplama fonksiyonu gösterilmektedir. Bu fonksiyonda daha önceden elde ettiği görüntünün histogram dizisi içerisinde tarama yaparak doğru değeri bulduğunda cdf'e yazar.

```
__global__ void  calcCDF (float * cdf, unsigned int * histo,int img_size, int length){
    __shared__ float scan[SCAN_SIZE];
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    if (i<SCAN_SIZE && i<length)
       scan[i] = (float)histo[i] / (float)(img_size);
     __syncthreads();

    for (unsigned int stride = 1; stride <= BLOCK_SIZE; stride *= 2) {
        unsigned int index = (threadIdx.x + 1)*stride * 2 - 1;
        if (index<SCAN_SIZE && index < length)
            scan[index] += scan[index - stride];
        __syncthreads();
    }

    for (unsigned int stride = BLOCK_SIZE / 2; stride>0; stride /= 2) {
        __syncthreads();
        unsigned int index = (threadIdx.x + 1)*stride * 2 - 1;
        if (index + stride < SCAN_SIZE && index + stride < length) {
            scan[index + stride] += scan[index];
        }
    }

    __syncthreads();
    if (i<SCAN_SIZE && i<length)
        cdf[i] += scan[threadIdx.x];
}
```

Şekil 3.11 GPU'da kümülatif dağılım fonksiyonun hesaplanması

Şekil 3.12'de gösterilen histogram eşitleme kernel fonksiyonumuzda önceden hesaplanmış olan giriş görüntüsünün histogram değeri ve cdf değeri alınarak, görüntünün yeni parlaklık değerleri elde edilir. Yeni değerin elde edilmesi işleminde seri versiyonunda, şekil 3.6'da gösterilen aynı formül kullanılır. Seri histogram eşitleme fonksiyonumuzdan tek farkı döngülerin yerine, iş parçacıklarının koordinat değişkenlerinin kullanılmasıdır.

```
__global__ void histogram_equalization_gpu(unsigned char * image, float * cdf,
int size) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    float cdfmin = cdf[0];
    if (i<size) {
        float x = 255.0F * (cdf[image[i]] - cdfmin) / (1.0F - cdfmin);
        float start = 0.0F;
        float end = 255.0F;
        if (start > x)
           x = start;
        if (end < x)
           x = end;
        image[i] = (unsigned char)x;
    }
   }
```

Şekil 3.12 GPU'da görüntü histogramını eşitleme fonksiyonu

4. DENEYLER VE SONUÇLAR

Bu tez çalışmasında görüntülerin kontrastını iyileştirmede kullanılan histogram hesaplama ve eşitleme algoritmalarının paralel ve seri versiyonları geliştirilerek CPU ve GPU üzerinde yürütülme performansları test edilerek incelenmiştir. Aşağıdaki bölümlerde sırasıyla testlerimizi yaptığımız platformlar, girdi görüntülerimiz, elde ettiğimiz sonuçlar ve analizlerimiz sunulmaktadır.

4.1 Test Ortamı

Çalışmamızda CUDA C'yi kullanarak GPU üzerinde paralel kod ve C programlama ile CPU üzerinde seri kod derlemek için şekil 4.1'de gösterilmekte olan araçlar ve platformlar kullanılarak geliştirme ve test ortamımız hazırlanmıştır.

ASUS K53SJ	**ASUS N552VW**
• Windows 10 işletim sistemi • İşlemci; Intel Core i7 – 26300Q 2.00GHz • Grafik Kartı; NVIDIA GeForce GT 540M (Fermi Mimarisi)	• Windows 10 işletim sistemi • İşlemci; Intel Core i7 – 6700HQ 2.60GHz • Grafik Kartı; NVIDIA GeForce GTX 940M (Maxwell Mimarisi)
Kullanılan Araçlar: • Visual Studio 2015 • NVIDIA Aygıt Sürücüsü • CUDA Geliştirme Araçseti (CUDA Development Toolkit Version 8.0)	

Çizelge 4.1 Tez Test Platformları ve Araçları

Tez çalışmasında deneyler çizelge 4.1'deki özelliklere sahip 2 farklı işlemci ve grafik kartında yapılmıştır. Kullanılan grafik kartlarının mimarileri Fermi ve Maxwell'dir. Bölüm 2.3 'de bahsedildiği gibi 2.1 hesaplama kapasitesine sahip Fermi Mimarisi 16 SM'den oluşur, 16LD/ST birim, 64KB toplam önbelleği L1 önbelleği ve paylaşılan bellek tarafından kullanılır ve toplam 96 çekirdek içerir. 5.0 hesaplama kapasitesine sahip Maxwell Mimarisi ise SMM olarak adlandırılan enerji verimliliğini büyük oranda arttıran farklı bir SM dizaynına sahiptir. SMM'in geliştirilmiş alan verimliliği ile GPU başına düşen CUDA çekirdeği

Fermi Mimarisi'ne kıyasla önemli ölçüde yüksektir. Kullandığımız bu mimarinin toplam çekirdek sayısı 640'dır. 32 LD/ST birimi, 32SFU ve Fermi'den farklı olarak özel olarak ayrılmış 96 KB'lik paylaşılan bellek alanı vardır. (Harris, 2014)

Deneylerden elde edilen tüm sonuçlarda programımızın hem seri hem paralel versiyonu için de kullandığımız iki farklı bilgisayarda ful şarjlı haldeyken 5'er kez çalıştırılmıştır. Her bir sonuç incelenerek en düşük ve yüksek değerler çıkartıldıktan sonra geriye kalan 3 değerin ortalaması alınarak sonuç değerlerimiz elde edilmiştir.

Histogram eşitleme yöntemi daha çok gri ölçekli sisli ve bulanık görüntülerde iyi sonuç vermektedir fakat deneylerimizdeki amacın uygulamaların performans sürelerini karşılaştırılması olması nedeniyle deneylerde kullanılacak olan görüntü seçimi yapılırken tekdüze renklerden çok farklı renkleri içinde bulunduran büyük boyutlu olmasına özen gösterilmiştir. Yapılan tüm deneyler aynı görüntünün 3 farklı boyutu (534 x 356, 2136 x 1424 ve 4272 x 2848) için denenmiştir. Kullanılan JPEG formatındaki RGB ve gri ölçekli görüntülerde RGB renkteki, PPM gri ölçekli görüntü ise PGM formatına dönüştürülerek girdi olarak kullanılmıştır. PPM ve PGM görüntü verilerinin kaydedilmesini sağlayan en düşük görüntü formatlarıdır. Farklı uygulamalarda bu formatlarının kullanımı ile görüntüler kolay bir şekilde okunabilir. Bu nedenle uygulamamızda bu formatların kullanımı tercih edilmiştir.

Şekil 4.1 kullandığımız görüntünün RGB renkte ki orijinal halini gösterilmektedir.

Şekil 4. 1 RGB renkte orjinal görüntü

Şekil 4.2 RGB renkteki orijinal resme histogram eşitleme uygulandıktan sonra çıkan PPM formatındaki sonucun JPEG formatına dönüştürülmüş hali gösterilmektedir.

Şekil 4. 2 RGB renkte histogram eşitleme uygulanan görüntü

Şekil 4.3'de deneyde kullanılan gri ölçekli orijinal resim ve şekil 4.4 'de orijinal resme histogram eşitleme uygulandıktan sonra çıkan PGM formatındaki sonucun JPEG formatına dönüştürülmüş hali gösterilmektedir.

Şekil 4. 3 Gri ölçekli orjinal görüntü

Şekil 4. 4 Gri ölçekli histogram eşitleme uygulanan görüntü

Şekil 4.5'de uygulamamızın CPU ve GPU üzerinde çalıştırılması ile elde edilen sonuç ekranı gösterilmektedir.

```
D:\yeni kodlar>nvcc -o cpu.exe commandwithCpu.cu
commandwithCpu.cu
   Creating library cpu.lib and object cpu.exp

D:\yeni kodlar>cpu.exe
 Running contrast enhancement for gray-scale and rgb images
================================================
Gray-scale image size: 534 x 356
[Compute] 1.1172345679 Performing CUDA gray histogram.
RGB color image size: 534 x 356
[Compute] 11.5160493827 Performing CUDA rgb histogram.
Press any key...d

D:\yeni kodlar>gpu.exe
Gray-scale image size: 534 x 356
[Compute] 2.2510617284 Performing on GPU histogram gray equ.
Rgb image size: 534 x 356
[Compute] 4.3725432099 Performing on GPU histogram equ.
Press any key...sj
```

Şekil 4. 5 Uygulamanın örnek sonuç ekranı

4.2 CPU ve GPU Üzerinde Hesaplama Hız Karşılaştırmaları

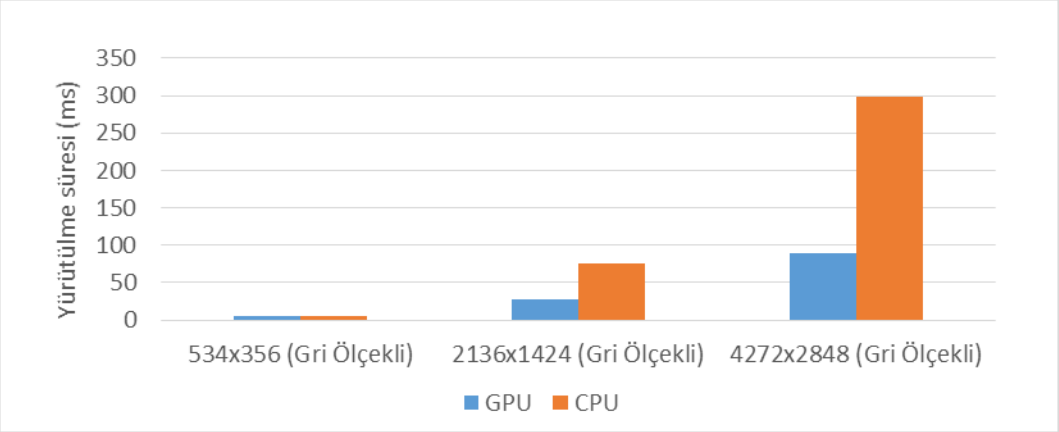
CPU için geliştirdiğimiz seri ve GPU için geliştirdiğimiz paralel kodlarımız iki farklı işlemci ve grafik kartı üzerinde çalıştırılmış ve çizelge 4.2'de ve çizelge 4.3'de elde ettiğimiz tüm sonuçlar sunulmuştur. Çizelge 4.2'de gösterilen daha düşük bir işlemci ve grafik kartı üzerinde elde edilen sonuçlara göre küçük boyuttaki resimlerde CPU'nun az bir farkla olsa daha GPU'ya göre performansı daha yüksek çıkmıştır. GPU üzerinde yapılan hesaplamalarda veri transferi için gereken zamandan dolayı küçük boyutlu resimlerde çok fazla performansı artışı

elde edilememektedir. Buna ek olarak görüntü boyutu arttıkça GPU'nun CPU'ya göre hesaplama performansında artış gözlenmektedir. Çizelge 4.2'de görüldüğü gibi küçük boyutlu resimlerde CPU az bir farkla olsa da daha iyi performans gösterirken, orta boyutlu resimde GPU performansının CPU'nun 2,8 katına kadar çıktığı, büyük boyutlu resimlerde ise bu farkın 3,5 katına kadar çıktığı görülmektedir.

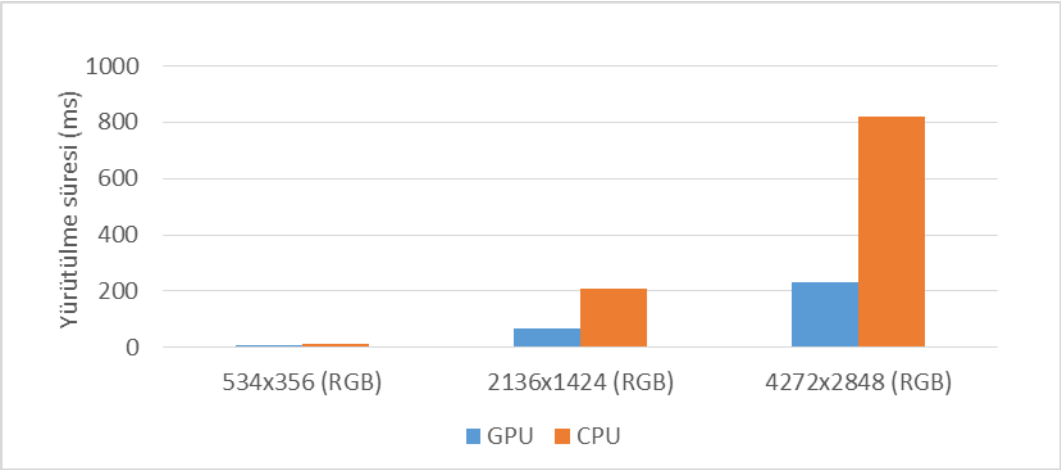
Girdi Resim Boyutu	GPU Süresi (ms)	CPU Süresi (ms)	Hızlanma (kat sayısı)
534x356 (Gri Ölçekli)	5.40	4,76	⬇0,88
534x356 (RGB Ölçekli)	7,72	13,60	⬆1,76
2136x1424 (Gri Ölçekli)	26,94	75,48	⬆2,80
2136x1424 (RGB Ölçekli)	66,10	210,55	⬆3,18
4272x2848 (Gri Ölçekli)	88,40	297,84	⬆3,37
4272x2848 (RGB Ölçekli)	231,79	819,23	⬆3,53

Çizelge 4.2 Intel Core i7 – 2.00GHz ve Fermi Mimarisi ile elde edilen sonuçlar

Şekil 4.6 ve 4.7'da 2.00Ghz CPU ve Fermi Mimarisi'ne sahip GPU üzerinde elde edilmiş sonuçları grafiksel olarak gösterilmektedir.



Şekil 4. 6 Intel Core i7 – 2.00GHz ve Fermi Mimarisi ile gri ölçekli resimlerde elde edilen GPU ve CPU sonuçlarının karşılaştırması



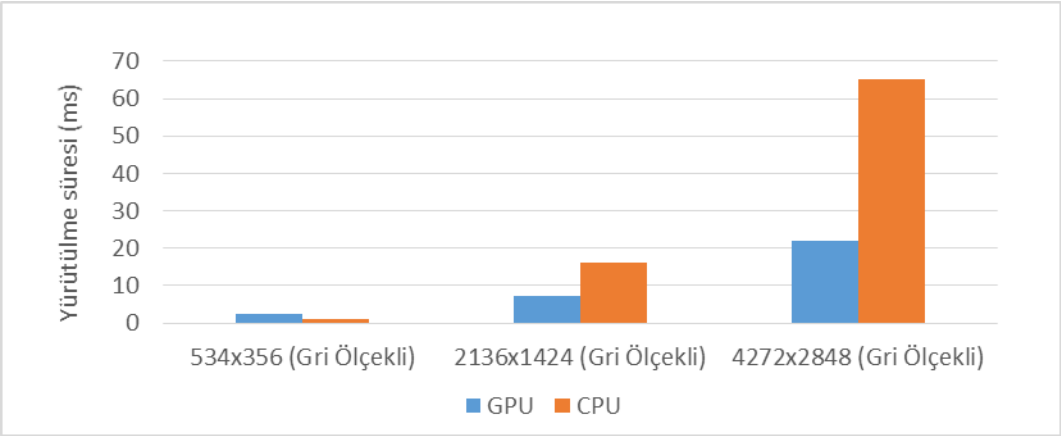
Şekil 4. 7 Intel Core i7 – 2.00GHz ve Fermi Mimarisi ile RGB renkte resimlerde elde edilen GPU ve CPU sonuçlarının karşılaştırması

Çizelge 4.3'de daha güçlü bir işlemci ve Maxwell Mimarisi 'ne sahip grafik kartından elde ettiğimiz sonuçlar gösterilmektedir. Çizelge 4.3'de görüldüğü gibi küçük boyutlu resimlerde CPU az bir farkla olsa da daha iyi performans gösterirken, orta boyutlu resimde GPU performansının CPU'nun 3,7 katına kadar çıktığı, büyük boyutlu resimlerde ise bu farkın 4,5 katına kadar çıktığı görülmektedir.

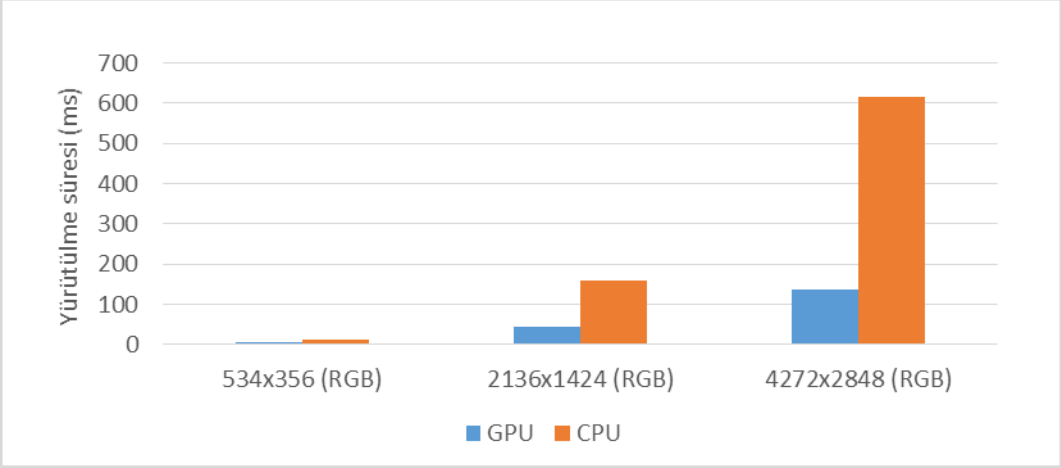
Girdi Resim Boyutu	**GPU Süresi (ms)**	**CPU Süresi (ms)**	**Hızlanma (kat)**
534x356 (Gri Ölçekli)	2.25	1,13	⬇ 0,50
534x356 (RGB Ölçekli)	4,33	10,31	⬆ 2,38
2136x1424 (Gri Ölçekli)	7,30	16,20	⬆ 2,22
2136x1424 (RGB Ölçekli)	42,75	159,29	⬆ 3,72
4272x2848 (Gri Ölçekli)	22,04	65,21	⬆ 2,96
4272x2848 (RGB Ölçekli)	136,19	615,60	⬆ 4,52

Çizelge 4.3 Intel Core i7 – 2.60GHz ve Maxwell Mimarisi ile elde edilen sonuçlar

Şekil 4.8 ve 4.9'da 2.60Ghz CPU ve Maxwell Mimarisi 'ne sahip GPU üzerinde elde edilmiş sonuçları grafiksel olarak gösterilmektedir.

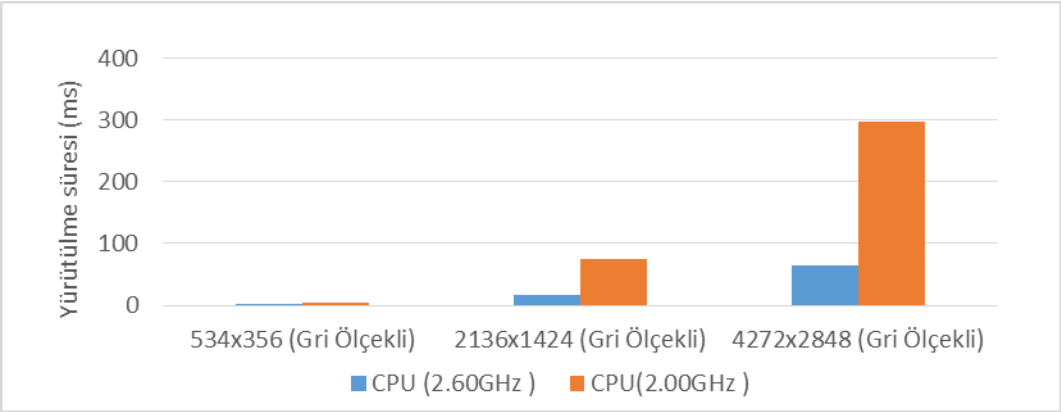


Şekil 4. 8 Intel Core i7 – 2.60GHz ve Maxwell Mimarisi ile gri ölçekli resimlerde elde edilen GPU ve CPU sonuçlarının karşılaştırması

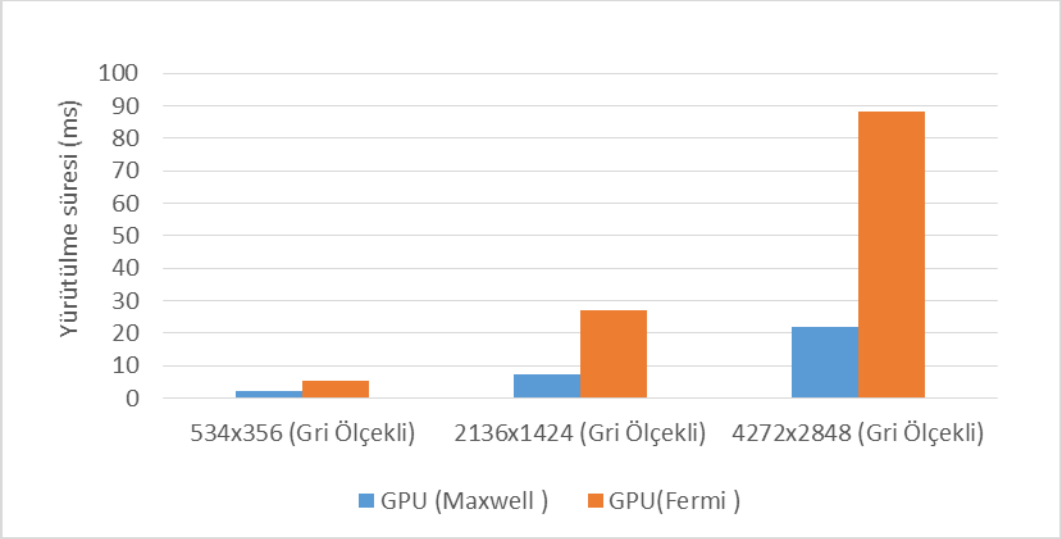


Şekil 4. 9 Intel Core i7 – 2.60GHz ve Maxwell Mimarisi ile RGB renkte resimlerde elde edilen GPU ve CPU sonuçlarının karşılaştırması

Şekil 4.10 2 farklı CPU üzerinde elde edilen sonuçların grafiksel olarak karşılaştırması, şekil 4.11' de ise 2 farklı GPU'dan elde edilen sonuçların grafiksel olarak karşılaştırması görülmektedir. Şekil 4.11' de Intel Core i7-2.60Hz ve 2.00Ghz işlemcilerin karşılaştırılması yapılmış ve 2.60Ghz işlemcinin performansı gri ölçekli resimlerde 4,7 kata kadar hızlandığı görülmektedir. Şekil 4.11'de Fermi ve Maxwell Mimarisi 'ne sahip GPU sonuçları karşılaştırıldığında 4,6 kata kadar Maxwell'in performans artışı sağladığı görülmektedir.



Şekil 4. 10 Intel Core i7 – 2.60GHz ve 2.00Ghz ile Gri ölçekli resimlerde elde edilen CPU sonuçlarının karşılaştırması



Şekil 4. 11 Fermi ve Maxwell Mimarisi ile Gri ölçekli resimlerde elde edilen GPU sonuçlarının karşılaştırması

4.2.1 Global ve paylaşılan bellek hızlarının karşılaştırılması

Çizelge 4.4'de ve 4.5'de Fermi ve Maxwell Mimarileri'nde yapmış olduğumuz histogram hesaplama işlemini CPU'da ve GPU üzerinde global bellek ve paylaşılan bellek kullanımı ile elde edilen performans karşılaştırılmaları gözükmektedir. Bu deneyde performans farklarının daha belirgin şekilde incelenebilmesi için en büyük boyuttaki görüntü kullanılarak yürütülme sürelerini hesaplanmıştır ve ana bellekten aygıt belleğine yapılan veri transfer süreleri hesaba katılmamıştır. 2 farklı işlemci üzerinde yapılan deneylerden elde edilen sonuçlara CPU'ya oranla GPU kullanımı ile büyük oranda performans artışı sağlandığı görülmektedir. Çizelge 4.4'de Fermi Mimarisi üzerinde global bellek ve paylaşılan bellek kullanımlarının performans hızları karşılaştırıldığında sonuç değerlerinin arasında büyük oranda fark görülmemektedir. Paylaşılan bellek

gücünden tam olarak yararlanılabilmesi için verimli bir donanım gerekmektedir. Bu donanım ihtiyacı Kepler ve sonraki mimari versiyonlarında karşılanmaktadır. Kullanılan GPU'nun eski bir sürüm olan Fermi Mimari'si olmasından dolayı bu deney sonucunda paylaşılan bellek kullanımından istenilen performans artışı elde edilememiştir.

İşlemciler (ASUS K53SJ)	Yürütülme Süresi (ms)
Intel Core i7 – 2.00GHz (CPU)	129,92
NVIDIA GeForce GT 540M (Fermi GPU, Global Bellek)	17,12
NVIDIA GeForce GT 540M (Fermi GPU, Paylaşılan Bellek)	18,53

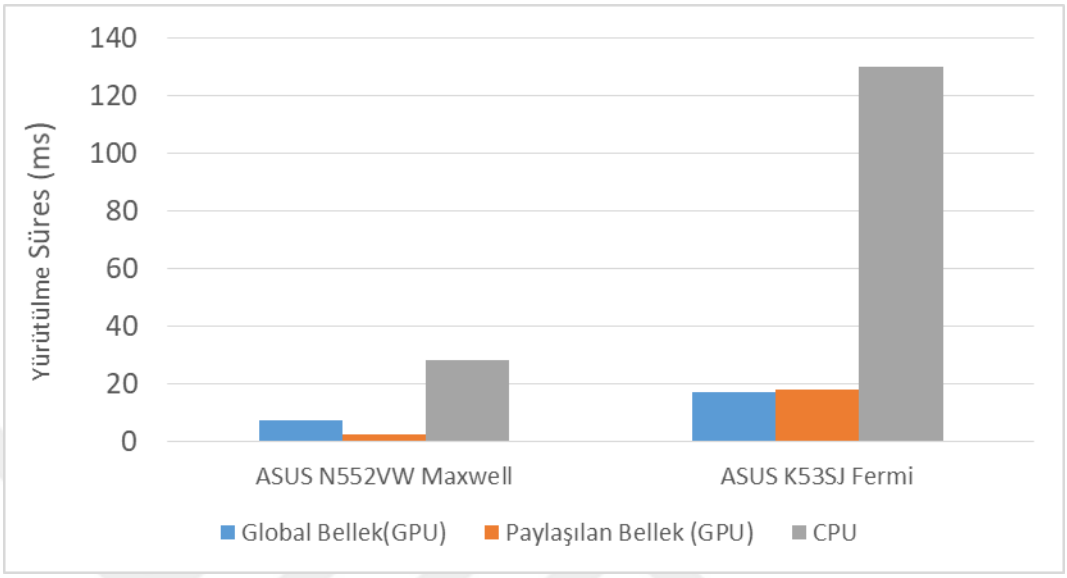
Çizelge 4.4 Fermi Mimarisi'nde histogram hesaplama yürütülme süreleri

Çizelge 4.5'de Maxwell Mimarisi üzerinde global bellek ve paylaşılan bellek kullanımlarının performans hızlarını karşılaştırdığımızda paylaşılan bellek kullanımını global bellek kullanımına göre yaklaşık olarak 3 kat daha hızlı olduğu görülmektedir. Global bellek CUDA bellekleri arasında en büyük ama aynı zamanda en yavaş olan bellektir. Paylaşılan bellek ise global belleğe göre oldukça küçüktür fakat çip üzerinde bulunduğundan dolayı bu belleğe erişim çok daha hızlıdır. Paylaşılan bellek kullanımında bir blok içerisindeki global bellekten çekilen verileri aynı blok içerisindeki diğer iş parçacıklarında kullanmak için bu verileri paylaşılan belleğe koyulur. Böylece tekrar kullanmamız gerektiğinde global bellekten yeniden bu verileri çekmek zorunda kalmayız (Harris, 2013). Bu durumda global bellek yerine paylaşılan belleğin doğru bir şekilde kullanımı ile global belleğin yoğun trafiğinden kaçınıp önemli ölçüde performans artışı sağlanmaktadır.

İşlemciler (ASUS N552VW)	Yürütülme Süresi (ms)
Intel Core i7 – 2.60GHz (CPU)	28,43
NVIDIA GeForce GTX 940M (Maxwell GPU, Global Bellek)	7,22
NVIDIA GeForce GTX 940M (Maxwell GPU, Paylaşılan Bellek)	2,41

Çizelge 4.5 Maxwell Mimarisi'nde histogram hesaplama yürütülme süreleri

Şekil 4.12'de 2.60Ghz ve 2.00Ghz CPU'da histogram hesaplama, Fermi ve Maxwell Mimarisi'ne sahip 2 farklı GPU üzerinde histogram hesaplama da global bellek ve paylaşılan bellek kullanımı ile elde edilen yürütülme sonuçları grafiksel olarak gösterilmektedir.



Şekil 4. 12 Maxwell ve Fermi Mimarilerinde global bellek, paylaşılan bellek kullanımı ve 2.60Ghz -2.00Ghz CPU sürelerinin karşılaştırılması

5. SONUÇ

Çalışmamızda görüntü işleme alanında kontrast iyileştirmede ve bir çok uygulamanın ön basamağı olarak en çok kullanılan yöntemlerden histogram eşitleme algoritması kullanılmıştır. Histogram eşitleme algoritmasının C programlama ile seri ve CUDA paralel programlama ile paralel versiyonları geliştirilerek 2 farklı GPU ve CPU üzerinde çalıştırılmış ve deney sonuçları gözlemlenmiştir.

Yapılan tüm çalışmaların sonucunda uygun durumlarda GPU kullanımının CPU'ya göre büyük oranda performans kazanımı sağladığı görülmüştür. CUDA programlamanın GPU üzerinde çalışabilen paralel uygulamalar geliştirilmesine sağladığı kolaylık ile büyük oranda paralel hesaplama gerektiren görüntü işleme problemleri çözüme kavuşabilmektedir.

Bu tez çalışmasında ayrıca, GPU ile CPU'ya göre performans kazanımının sağlandığının görülmesinin ardından GPU üzerinde global bellek ve paylaşılan bellek kullanımı incelenmiştir. Paylaşılan belleğin çip üzerinde bulunması ve erişim hızının oldukça yüksek olması, global belleğin ise CUDA bellekleri arasındaki en çok kullanılan fakat en yavaş bellek olmasından dolayı CUDA programlamada kullanıma uygun belleğin seçilmesi performans kazanımı açısından oldukça önemli olduğu deney sonuçlarından gözlemlenmiştir. Eğer bir blok içerisindeki iş parçacıkları tarafından birden fazla defa global bellekten çekilmiş bir veriye erişilmesi gerekiyorsa paylaşılan bellek kullanımının yaklaşık 3 kata kadar hız kazandırabildiği görülmüştür. Gelecekteki çalışmalarımızda da, daha ileri düzey problemlerde CUDA kullanımını inceleyeceğiz.

KAYNAKLAR DİZİNİ

Borke P., 1997, "PPM/ PGM / PBM image files" http://paulbourke.net/dataformats/ppm/, (Erişim Tarihi: 5 Mart 2018)

Cheng, J., Grossman, M., and McKercher, T., 2014, Professional CUDA C Programming, John Willey & Sons, Indianapolis, 8-264p.

Clua, E. G. W., Zamith, M., 2015, Programming in CUDA for Kepler and Maxwell Architecture, Revista de Informatica Te ´ orica e Aplicada, 22 (2): 233-257 pp.

Ding C., "CUDA Tutorial", http://geco.mines.edu/tesla/cuda_tutorial_mio/, (Erişim Tarihi: 10 Mart 2018)

Eklund, A., Dufort, P., Forsberg, D. and LaConte, S. M., 2013, Medical image processing on the GPU–past, present and future, Medical Image Analysis, 17(8): 1073–1094 pp.

Gaura J., 2016, "Histogram Equalization", http://mrl.cs.vsb.cz/people/gaura/dzo/hist_en.pdf, (Erişim Tarihi: 25 Ocak 2018)

Ghorpade, J., Parande, J., Kulkarni, M. and Bawaskar, A., 2012, GPGPU processing in CUDA architecture, Advance Computing: An International Journal (ACIJ), 3(1):105-120p.

Gupta N., 2013, "Optimization in Histogram in CUDA", http://cuda-programming.blogspot.com.tr/2013/03/optimizing-histogram-cuda-code.html, (Erişim Tarihi: 15 Ocak 2017)

Gupta N., 2013, "Fast Implementation of Histogram Cuda", http://cuda-programming.blogspot.com.tr/2013/03/further-optimization-in-histogram-cuda.html, (Erişim Tarihi: 20 Ocak 2017)

Harris M., 2012, "How to Optimize Data Transfers in CUDA C/C++", https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc, (Erişim Tarihi: 28 Şubat 2018)

Harris M., 2013, "Using Shared Memory in CUDA C/C++", https://devblogs.nvidia.com/using-shared-memory-cuda-cc/ , (Erişim Tarihi: 30 Ocak 2018)

KAYNAKLAR DİZİNİ (devam)

Harris, M., 2014, "5 Things You Should Know About the New Maxwell GPU Architecture", https://devblogs.nvidia.com/parallelforall/5-things-you-should-know-about-new-maxwell-gpu-architecture/, (Erişim Tarihi: 23 Aralık 2017)

Lee, V., W., Kim C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A., D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R. and Dubey, P., 2010, Debunking the 100x GPU vs. CPU Myth: an Evaluation of Throughput Computing on CPU and GPU, In Proceedings of the 37th Annual International Aymposium on Computer Architecture, 451-460p.

Milic U., Gelado, I., Puzovic, N., Ramirez, A. and Tomasevic, M., 2013, Parallelizing General Histogram Application for CUDA Architectures, In Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 11-18p.

New York University, 2017, "Introduction to GPUs CUDA", https://nyu-cds.github.io/python-gpu/02-cuda/, (Erişim Tarihi 10 Mart 2018)

Nickolls J., Buck I., Garland M. and Skadron K., 2008, Scalable Parallel Programming with CUDA, ACM Digital Library, 6(2): 42-53p.

NVIDIA, 2007, "NVIDIA's Next Generation CUDA Compute Architecture Fermi", https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, (Erişim Tarihi: 20 Aralık 2017)

NVIDIA, 2018, "CUDA C Programming Guide – NVIDIA Developer Documentation", http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#heterogeneous-programming__heterogeneous-programming, (Erişim Tarihi: 20 Şubat 2018)

Paravecino, F. N., 2017, Characterization and Exploitation of Nested Parallelism and Concurrent Kernel Execution to Accelerate High Performance Applications, Dissertation, Northeastern University.

Sakharnykh, N., 2015, "GPU Pro Tip: Fast Histograms Using Shared Atomics on Maxwell", https://devblogs.nvidia.com/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/, (Erişim Tarihi: 20 Ocak 2018)

KAYNAKLAR DİZİNİ (devam)

Wadud M. A., Kabir M. H., M. Dewan A. A., and Chae O., 2017, A Dynamic Histogram Equalization for Image Contrast Enhancement, IEEE Transactions on Consumer Electronics, 53(2):593-600p.

Yang, Z.,Zhu, Y. and Pu. Y., 2008**,** Parallel image processing based on CUDA, In International Conference on Computer Science and Software Engineering 3:198-201p.

Zhang, N., Wang, J. L. and Chen. Y. shan, 2010**,** Image parallel processing based on GPU, Advanced Computer Control (ICACC), 3:367-370p.

ÖZGEÇMİŞ

Ad Soyad : Pelin KARAGÖZOĞLU
Doğum Tarihi : 09.07.1992
Doğum Yeri : Balıkesir

Telefon : (+90) 538 974 51 18
E-posta : pelin.karagozoglu@gmail.com

Eğitim :

- 2010 – 2015 Işık Üniversitesi Yazılım Mühendisliği Bölümü 3,21/4
- 2006 – 2010 Balıkesir Cumhuriyet Anadolu Lisesi 85/100

İş Tecrübeleri :

- 2015- 2018 Freelance Android Geliştiricisi
- 2018- Halen Android Geliştircisi (HUAWEI)