
GAZİ ÜNİVERSİTESİ
Gazi
1926

PARALEL VERİ İŞLEME TEKNİKLERİ KULLANILARAK SİLİNDİRİK METAL NESNELERİN YÜZEY HATALARININ GERÇEK ZAMANLI OLARAK BELİRLENMESİ

Semra AYDIN

DOKTORA TEZİ
ELEKTRONİK VE BİLGİSAYAR EĞİTİMİ ANA BİLİM DALI

GAZİ ÜNİVERSİTESİ
BİLİŞİM ENSTİTÜSÜ

OCAK 2019

Semra AYDIN tarafından hazırlanan "PARALEL VERİ İŞLEME TEKNİKLERİ KULLANILARAK SİLİNDİRİK METAL NESNELERİN YÜZEY HATALARININ GERÇEK ZAMANLI OLARAK BELİRLENMESİ" adlı tez çalışması aşağıdaki jüri tarafından OY BİRLİĞİ ile Gazi Üniversitesi Elektronik-Bilgisayar Eğitimi Ana Bilim Dalında DOKTORA TEZİ olarak kabul edilmiştir

Danışman: Prof. Dr. Ömer Faruk BAY
Elektrik-Elektronik Mühendisliği, Gazi Üniversitesi
Bu tezin, kapsam ve kalite olarak Doktora Tezi olduğunu onaylıyorum.

İkinci Danışman: Prof. Dr. Refik SAMET
Bilgisayar Mühendisliği, Ankara Üniversitesi
Bu tezin, kapsam ve kalite olarak Doktora Tezi olduğunu onaylıyorum.

Başkan: Prof. Dr. Şeref SAĞIROĞLU
Bilgisayar Mühendisliği, Gazi Üniversitesi
Bu tezin, kapsam ve kalite olarak Doktora Tezi olduğunu onaylıyorum.

Üye: Prof. Dr. Fırat HARDALAÇ
Elektrik-Elektronik Mühendisliği, Gazi Üniversitesi
Bu tezin, kapsam ve kalite olarak Doktora Tezi olduğunu onaylıyorum.

Üye: Doç. Dr. Hacer KARACAN
Bilgisayar Mühendisliği, Gazi Üniversitesi
Bu tezin, kapsam ve kalite olarak Doktora Tezi olduğunu onaylıyorum.

Üye: Doç. Dr. Hakan MARAŞ
Bilgisayar Mühendisliği, Çankaya Üniversitesi
Bu tezin, kapsam ve kalite olarak Doktora Tezi olduğunu onaylıyorum.

Üye: Dr. Öğretim Üyesi Atila BOSTAN
Bilgisayar Mühendisliği, Atılım Üniversitesi
Bu tezin, kapsam ve kalite olarak Doktora Tezi olduğunu onaylıyorum.

Tez Savunma Tarihi: 23/01/2019

Jüri tarafından kabul edilen bu tezin Doktora Tezi olması için gerekli şartları yerine getirdiğini onaylıyorum.

.....

Doç. Dr. Aslıhan TÜFEKCİ
Bilişim Enstitüsü Müdürü

ETİK BEYAN

Gazi Üniversitesi Bilişim Enstitüsü Tez Yazım Kurallarına uygun olarak hazırladığım bu tez çalışmasında;

- Tez içinde sunduğum verileri, bilgileri ve dokümanları akademik ve etik kurallar çerçevesinde elde ettiğimi,
- Tüm bilgi, belge, değerlendirme ve sonuçları bilimsel etik ve ahlak kurallarına uygun olarak sunduğumu,
- Tez çalışmasında yararlandığım eserlerin tümüne uygun atıfta bulunarak kaynak gösterdiğimi,
- Kullanılan verilerde herhangi bir değişiklik yapmadığımı,
- Bu tezde sunduğum çalışmanın özgün olduğunu,

bildirir, aksi bir durumda aleyhime doğabilecek tüm hak kayıplarını kabullendiğimi beyan ederim.



Semra AYDIN

23/01/2019

PARALEL VERİ İŞLEME TEKNİKLERİ KULLANILARAK SİLİNDİRİK METAL NESNELERİN YÜZEY HATALARININ GERÇEK ZAMANLI OLARAK BELİRLENMESİ

(Doktora Tezi)

Semra AYDIN

GAZİ ÜNİVERSİTESİ

BİLİŞİM ENSTİTÜSÜ

Ocak 2019

ÖZET

Mühimmat üretimi savunma sanayinin önemli üretim kollarından biridir. Hafif ateşli silahların mühimmatının her birine fişek adı verilir. Kovan üretimi fişek üretiminin bir aşamasıdır ve kovan üretim aşamasında hatalı olan kovanlar ayrılarak üretim bandından çıkarılmaktadır. Bu çalışmanın konusu kovan hatalarını gerçek zamanlı olarak tespit ederek, hatalı kovanları üretim bandından ayıklamaktır. Endüstride otomasyon sistemlerinin kullanımı giderek artmaktadır. Bilgisayarlı görmede görüntü işleme teknikleri kullanılarak karar verme işlemi otomatik olarak yapılmaktadır. Hatalı parçaların bulunması bilgisayarlı görmenin endüstride kullanım alanlarından birisidir. Bu çalışmada hatalı kovanların otomatik olarak ayıklanmasını sağlayan bir sistem geliştirilmiştir. Hataların tespit edilmesinde görüntü işleme teknikleri kullanılmıştır. Gerçekleştirilen sistemde gerçek zamanlı olarak kovanın ağız, kapsül ve yan yüzey olmak üzere farklı bölgelerindeki hataları tespit edilmektedir. Bunun için her bir kovandan bir adet ağız, bir adet kapsül ve sekiz adet yan yüzey olmak üzere toplam 10 adet görüntü alınmaktadır. Bu görüntülerde kovan hatalarının tespit edilmesi için farklı hataların tespiti amacıyla algoritmalar geliştirilmiş ve gerçekleştirilen sistem üzerinde çalıştırılmıştır. Gerçekleştirilen kovan muayene sisteminin kapasitesi dakikada 300 kovan olarak hedeflendiğinden 200 ms'lik süre içerisinde bir kovan ile ilgili tüm işlemlerin tamamlanması gerekmektedir. Alınan görüntü sayısının fazla olması ve hedeflenen hız nedeniyle işlem zamanının kısaltılması için paralel programlama teknikleri kullanılmıştır. Otomatik kovan muayene sisteminin gerçek zamanlı olarak çalıştırılması amacıyla hem CPU çekirdeklerini hem de GPU'nun işlem gücünü kullanan hibrit bir sistem önerilmiştir. CPU çekirdeklerinin paralel çalıştırılmasında OpenMP, GPU çekirdeklerinin paralel çalıştırılmasında ise CUDA platformu kullanılmıştır. Böylece hem CPU'nun hem de GPU'nun gücünü kullanan hibrit bir sistem geliştirilmiştir. Kovan hatalarını bulma algoritması seri algoritmalarla 583 ms'de tamamlanırken, paralelleştirilmiş algoritmalarla 99 ms'de tamamlanmıştır. Hata ayıklama için geliştirilmiş ve paralelleştirilmiş algoritmalarla her bir kovanın işlem süresi seriye göre 5,9 kat hızlandırılmıştır.

Bilim Kodu	:	92422
Anahtar Kelimeler	:	GPU programlama, paralel programlama, görüntü işleme, kovan muayene
Sayfa Adedi	:	132
Danışman	:	Prof. Dr. Ömer Faruk BAY
İkinci Danışman	:	Prof. Dr. Refik SAMET

REAL-TIME DETERMINATION OF THE SURFACE ERRORS OF CYLINDRICAL METAL OBJECTS USING PARALLEL DATA PROCESSING TECHNIQUES

(Ph. D. Thesis)

Semra AYDIN

GAZİ UNIVERSITY

INFORMATICS INSTITUTE

January 2019

ABSTRACT

The production of ammunition is one of the important production lines of the defense industry. Each of the ammunition of light firearms is called a cartridge. Cartridge case production is a stage of cartridge production and cases which are defected in the case production stage are separated and removed from the production line. The subject of this study is to detect defect errors in real-time and to extract faulty casings from production line. The use of automation systems in industry is increasing. In computer vision, decision making is performed automatically by using image processing techniques. The presence of defected parts is one of the industrial applications of computer vision. In this study, a system has been developed for automatic determination and removal of defected casings. Image processing techniques are used to detect defects. In real-time system, it is necessary to determine the defects of the different parts of case which are mouth, capsule and surface. For this purpose, a total of 10 images are taken for each case, one mouth, one capsule and eight surfaces. Algorithms have been developed and run on the system to detect different defects for detection of case defects in these images. Since the capacity of the case inspection system is targeted at 300 cases per minute, all operations related to a case should be completed within 200 ms. Parallel programming techniques have been used to reduce the processing time due to the high number of images received and the targeted speed. In order to operate the automatic case inspection system in real time, a hybrid system has been proposed that uses both CPU cores and GPU's processing power. OpenMP is used in parallel operation of CPU cores and CUDA platform is used in parallel operation of GPU cores. The hybrid system has been developed that uses the power of both the CPU and the GPU. The algorithm for detecting case defects was completed in 583 ms by using serial algorithms and 99 ms by using parallel algorithms. The processing time of each case, for developed parallel algorithms for determination and removal of defective casing, was improved by 5.9 times compared to the those of series algorithm.

Science Code	:	92422
Key Words	:	GPU programming, parallel programming, image processing, ammunition inspection
Page Number	:	132
Supervisor	:	Prof. Dr. Ömer Faruk BAY
Co-Supervisor	:	Prof. Dr. Refik SAMET

TEŞEKKÜR

Çalışmalarım boyunca değerli yardım ve katkılarıyla beni yönlendiren tez danışmanım Prof. Dr. Ömer Faruk BAY'a, yine kıymetli tecrübelerinden faydalandığım ikinci danışmanım Prof. Dr. Refik SAMET'e, verdikleri desteklerden dolayı TİK üyeleri Prof. Dr. Şeref SAĞIROĞLU ve Prof. Dr. Fırat HARDALAÇ'a, tez çalışmamla birlikte yürüttüğüm SANTEZ projesinde görevli tüm çalışma arkadaşlarıma, manevi destekleriyle beni hiçbir zaman yalnız bırakmayan eşim Zekeriya AYDIN'a teşekkürü bir borç bilirim.

Bu tezimi 0018.STZ.2013-1 numaralı SANTEZ Projesi ile destekleyen Sanayi ve Teknoloji Bakanlığına ve SANTEZ projesinin sanayi ortağı olan MKE Gazi Fişek Fabrikasına da ayrıca teşekkür ederim.

İÇİNDEKİLER

Sayfa

ÖZET iv

ABSTRACT v

TEŞEKKÜR vi

İÇİNDEKİLER vii

ÇİZELGELERİN LİSTESİ xi

ŞEKİLLERİN LİSTESİ xii

RESİMLERİN LİSTESİ xvi

SİMGELER VE KISALTMALAR xvii

1. GİRİŞ 1

2. LİTERATÜR TARAMASI 9

2.1. Görüntü İşleme Teknikleri ile Hata Bulma 9

2.2. Görüntü İşlemede Paralelleştirme 11

3. GÖRÜNTÜ İŞLEME 17

3.1. Görüntü Geriçatma 19

3.2. Görüntü İyileştirme 19

3.3. Görüntü Bölütleme 19

3.4. Görüntü Çakıştırma 20

3.5. Görüntü Sınıflandırma 20

4. PARALEL PROGRAMLAMA 21

4.1. Ortak Bellekli Sistemler 23

4.1.1. Tekdüze bellek erişimi 24

4.1.2. Tekdüze olmayan bellek erişimi 25

4.1.3. Ortak Bellek Modellerinin Karşılaştırılması 26

Sayfa

4.1.4. OpenMP 27
4.2. Dağıtık Bellekli Sistemler 40
4.2.1. PVM 42
4.2.1. MPI 43
5. GPU PROGRAMLAMA VE CUDA 47
5.1. Genel Amaçlı GPU Programlama Tarihi 47
5.2. GPU Üreticileri 48
5.3. GPU Programlama 49
5.4. GPU ve CPU Mimarisi Arasındaki Farklar 51
5.4.1. Transistör kullanımı 51
5.4.2. Ön belleğin rolü 52
5.4.3. Gecikme yönetimi 52
5.4.4. Çoklu iş parçacığı 52
5.4.5. SIMD ve SIMT 52
5.4.6. Bellek denetleyici 52
5.5. İşlem ve İş Parçacığı Farkı 53
5.6. CUDA (Compute Unified Device Architecture) 54
5.6.1. CUDA aygıtları 55
5.6.2. CUDA mimarisi 55
5.6.3. CUDA hafıza modeli 58
5.6.4. Hafıza optimizasyonu 60
5.6.5. Bir CUDA programının yapısı 66
5.6.6. CUDA araçları ve donanım test araçları 67
5.6.7. CUDA platformunda kod çalıştırma 69

Sayfa

5.6.8. CUDA kütüphaneleri 72

6. SİSTEM TASARIMI, HATA BULMA ALGORİTMALARI GELİŞTİRME VE PARALELLEŞTİRME 79

6.1. Sistem Tasarımı 79

6.2. Kovan Ağız Hatalarının Bulunması 87

6.2.1. Arka zemin çıkarma 90

6.2.2. Görüntünün kesilmesi 92

6.2.3. Dairelerin bulunması 93

6.2.4. Eşikleme işlemi 94

6.2.5. Kenarların belirlenmesi 95

6.2.6. Yarık hatasının tespit edilmesi 96

6.2.7. Yeni görüntüde merkez noktasının tespit edilmesi 96

6.2.8. Varyansların toplamının hesaplanması 97

6.2.9. Sınıflandırma 98

6.2.10. Performans ölçütleri 99

6.2.11. Değerlendirme 101

6.3. Kapsül Hatalarının Bulunması 102

6.4. Yan Yüzey Hatalarının Bulunması 102

6.4.1. Algoritmanın GPU ile Paralelleştirilmesi 103

6.4.2. Gaussian filtresi 105

6.4.3. Şablon eşleme 107

6.4.4. İki yanlı filtre 109

6.4.5. Sobel filtresi 113

6.4.6. Eşikleme 115

6.4.7. Morfolojik operasyonlar 117

Sayfa

6.4.8. GPU'da koşturulan algoritmaların değerlendirilmesi 120

6.5. OpenMP ile Paralelleştirme 122

7. SONUÇ VE ÖNERİLER 123

KAYNAKLAR 125

ÖZGEÇMİŞ 131

ÇİZELGELERİN LİSTESİ

Çizelge **Sayfa**

Çizelge 4.1. OpenMP ile elde edilen hızlanma 36

Çizelge 4.2. OpenMP ile çekirdek sayısına göre elde edilen hızlanma 40

Çizelge 6.1. Karışıklık matrisi 99

Çizelge 6.2. Karışıklık matrisi sonuçları 101

Çizelge 6.3. Uygulamada test edilen NVIDIA ekran kartları ve çekirdek sayıları 105

Çizelge 6.4. 5X5'lük Gaussian filtresi işlem süreleri 107

Çizelge 6.5. Şablon eşleme işlem süreleri 108

Çizelge 6.6. İki yanlı filtre işlem süreleri 112

Çizelge 6.7. Sobel filtresi işlem süreleri 114

Çizelge 6.8. Eşikleme işlem süreleri 116

Çizelge 6.9. Morfolojik operasyonlardan kapama işlem süreleri 119

Çizelge 6.10. Tüm algoritmaların GPU işlem süreleri 120

Çizelge 6.11. Tesla K40 sonuçları 121

ŞEKİLLERİN LİSTESİ

Şekil **Sayfa**

Şekil 3.1. Görüntü işleme işlem basamakları 18

Şekil 4.1. Seri kod işlemenin şematik gösterimi 21

Şekil 4.2. Paralel kod işlemenin şematik gösterimi 21

Şekil 4.3. Ortak bellekli bilgisayar mimarisi şematik gösterimi 24

Şekil 4.4. Tekdüze bellek erişimi mimarisi 24

Şekil 4.5. Tekdüze olmayan bellek erişimi mimarisi 25

Şekil 4.6. OpenMP,Cilk ve TBB karşılaştırması 26

Şekil 4.7. Seri programın çok çekirdekli mimaride çalışması 28

Şekil 4.8. OpenMP örnek program çıktısı 29

Şekil 4.9. OpenMP iş parçacığı oluşumu 30

Şekil 4.10. *Schedule (static, 250)* için iş parçacıklarının çalışması 32

Şekil 4.11. *Schedule (static, 10)* için iş parçacıklarının çalışması 33

Şekil 4.12. İşlem zamanı ve veri bloğu büyüklüğü arasındaki ilişki 34

Şekil 4.13. İş parçacığı sayısı ve işlem zamanı arasındaki ilişki 34

Şekil 4.14. Kovan görüntüleri 35

Şekil 4.15. İşlem zamanı ile veri bloğu büyüklüğü arasındaki ilişki 37

Şekil 4.16. Hızlanma ile veri bloğu büyüklüğü arasındaki ilişki 38

Şekil 4.17. OpenMP'de çekirdek sayısı ile hızlanma arasındaki ilişki 40

Şekil 4.18. Dağıtık bellekli bilgisayar mimarisi şematik gösterimi 41

Şekil 4.19. Dağıtık bellek yapısı 41

Şekil 5.1. CPU, GPU bant genişliğinin yıllara göre karşılaştırması 48

Şekil 5.2. GPU programlama gelişimi 50

Şekil **Sayfa**

Şekil 5.3. OpenCL ve CUDA karşılaştırması 51

Şekil 5.4. İşlem ve iş parçacığı çalışması 54

Şekil 5.5. İş parçacıklarının ızgara ve bloklarda çalıştırılması 56

Şekil 5.6. GPU mimarisi 57

Şekil 5.7. SM'nin yapısı 57

Şekil 5.8. CUDA Hafıza Modeli 58

Şekil 5.9. GPU'da bellekler ve işlemcilerin yapısı 59

Şekil 5.10. Bütünleşmiş bellek 62

Şekil 5.11. Yığın çatışması olmayan bellek adresleme 63

Şekil 5.12. Yığın çatışması gerçekleşen bellek adresleme 63

Şekil 5.13. Ayrılma ile *warp* içerisinde işlemler 64

Şekil 5.14. Bir CUDA programının işlemesi 66

Şekil 5.15. CUDA platformunda yazılım donanım kavramlarının eşleştirilmesi 67

Şekil 5.16. Nsight menüsü ekran görüntüsü 68

Şekil 5.17. Visual Studio'da CUDA projesi oluşturma 68

Şekil 5.18. Ekran kartı modeli ve bant genişlikleri 69

Şekil 5.19. Ekran kartının teknik özellikleri 70

Şekil 5.20. CUDA program çıktısı 72

Şekil 5.21. NPP'de görüntü parametreleri 76

Şekil 6.1. 7.62X51 mm fişek 80

Şekil 6.2. Fişeğin bölümleri 80

Şekil 6.3. Kovanın bölümleri 83

Şekil 6.4. Geliştirilen sistemin şematik gösterimi 84

Şekil 6.5. Geliştirilen sisteme ait blok diyagramı 85

Şekil **Sayfa**

Şekil 6.6. Kovan ağız görüntüleri 88

Şekil 6.7. Ağız hatası bulma algoritması blok diyagramı 89

Şekil 6.8. Bekleme istasyonu ile alınan görüntü 90

Şekil 6.9. Arka zemin çıkarma 91

Şekil 6.10. Kesilen görüntü örneğinin önceki ve sonraki görüntüsü 92

Şekil 6.11. Hough algoritması sonucu işaretlenmiş örnek ağız görüntüleri 93

Şekil 6.12. Eşiklenmiş ve morfoloji uygulanmış örnek görüntüler 94

Şekil 6.13. Kovan ağzı iç çeperinin tespiti 96

Şekil 6.14. Bulunan minimum dikdörtgenin görüntü üzerinde işaretlenmesi 97

Şekil 6.15. Dairesel varyansların hesaplanması 98

Şekil 6.16. Geliştirilen arayüz yazılımında ağız hataları 99

Şekil 6.17. Geliştirilen arayüz yazılımında kapsül hataları 102

Şekil 6.18. Yan yüzeyin bölgelere ayrılması 104

Şekil 6.19. Orijinal görüntü 105

Şekil 6.20. 5X5 Gaussian filtresi sonucu 107

Şekil 6.21. Şablon eşleme kaynak görüntü 108

Şekil 6.22. Kullanılan şablon 108

Şekil 6.23. Tipik görüntü filtresi 109

Şekil 6.24. İki yanlı filtre şeması 110

Şekil 6.25. 3x3 boyutlarındaki filtre maskesi 110

Şekil 6.26. İki yanlı filtre sonucu 112

Şekil 6.27. 3X3 Sobel filtresi sonucu 115

Şekil 6.28. Genişletme işlemi 117

Şekil 6.29. Aşındırma işlemi 117

Şekil **Sayfa**

Şekil 6.30. Kapama işlemi sonucu 118

Şekil 6.31. Geliştirilen arayüz yazılımında yan yüzey hataları 119

RESİMLERİN LİSTESİ

Resim **Sayfa**

Resim 6.1. Gözle muayene tezgahları 81

Resim 6.2. Geliştirilen sistemin görüntüsü 86

SİMGELER VE KISALTMALAR

Bu çalışmada kullanılmış simgeler ve kısaltmalar, açıklamaları ile birlikte aşağıda sunulmuştur.

Simgeler	**Açıklamalar**
s	Saniye
ms	Milisaniye
GB	Gigabayt
GHz	Gigahertz
MHz	Megahertz

Kısaltmalar	**Açıklamalar**
CPU	Merkezi İşlem Birimi (Central Processing Unit)
CUDA	Birleşik Hesaplama Aygıt Mimarisi (Compute Unified Device Architecture)
DRAM	Dinamik Rasgele Erişimli Bellek
FFT	Hızlı Fourier Dönüşümü
FPGA	Alan Programlanabilir Kapı Dizileri (Field Programmable Gate Array)
GPU	Grafik İşlem Birimi (Graphical Processing Unit)
GPGPU	Genel amaçlı GPU programlama (General Purpose Computing on GPU)
ID	Kimlik Numarası
IPC	Süreçler Arası İletişim (Inter Process Communication)
KNN	K En yakın komşu (K Nearest Neighbourhood)
LAN	Yerel alan ağı (Local Area Network)
MIMD	Çoklu komut çoklu veri (Multiple Instruction Multiple Data)

Kısaltmalar	Açıklamalar
MISD	Çoklu komut tekli veri (Multiple Instruction Single Data)
MKE	Makine Kimya Endüstrisi
MPI	Mesaj Geçiş Arayüzü (Message Passing Interface)
MPP	Büyük Ölçekli Paralel İşlemci (Massively Parallel Processor)
MR	Manyetik Rezonans
NASA	Ulusal Havacılık ve Uzay Dairesi
NATO	Kuzey Atlantik Antlaşması Örgütü
NPP	NVIDIA Performans Temelleri (NVIDIA Performance Primitives)
OpenCL	Açık Hesaplama Dili (Open Computing Language)
OpenMP	Açık Çoklu İşleme (Open Multi Processing)
PVM	Paralel Sanal Makine (Parallel Virtual Machine)
RAM	Rasgele Erişimli Bellek (Random Access Memory)
ROI	İlgi Bölgesi (Region of Interest)
RISC	Azaltılmış Komut Kümeli Bilgisayarlar
SART	**S**ürekli Cebirsel Geriçatma Tekniği (Simultaneous Algebraic Reconstruction Technique)
SIMD	Tekli Komut Çoklu Veri (Single Instruction Multiple Data)
SIRT	Sürekli İteratif Geriçatma **(**Simultaneous Iterative Reconstruction Technique)
SISD	Tekli Komut Tekli Veri (Single Instruction Single Data)
SM	Çoklu İşlemci Modülü (Streamimg Multiprocessor)
SP	Skaler İşlemci (Scalar Processor)
SVM	Destek Vektör Makineleri (Support Vector Machine)
TBB	İş Parçacıklı İşleme Blokları (Threading Building Blocks)
2B	İki Boyutlu
3B	Üç Boyutlu

1. GİRİŞ

Ülkelerin ekonomik ve siyasal alandaki gücünü savunma sanayi ve ülkelerin bu alandaki harcamaları belirlemektedir. Teknolojik açıdan gelişmiş bir savunma sanayi ülkelerin bu alanda dış pazardaki gücünü artırmaktadır. Ülkemizde savunma sanayinin gücünün artması ekonomik ve siyasal alandaki gücümüzü de artıracaktır.

Türkiye'de savunma sanayinin en önemli üretim kollarından biri de mühimmat üretimidir [1]. Mühimmatlar iki grupta sınıflandırılır. Bunlar hafif ateşli ve ağır ateşli silah mühimmatlarıdır. Makine Kimya Endüstrisi (MKE) Gazi Fişek Fabrikası Türkiye'de hafif silah mühimmatı üreten bir fabrikadır. Hafif ateşli silahlarda kullanılan mühimmatın her birine fişek adı verilir. Gazi Fişek Fabrikasında farklı boyutlarda fişek üretilmektedir. Üretim kapasitesi en yüksek olan mühimmat 7,62 mm çaplı fişeklerdir.

Fişek; mermi çekirdeği, kapsül, sevk barutu ve kovan olmak üzere dört bölümden oluşmaktadır. Fişek üretimi çeşitli aşamalarda gerçekleştirilmektedir. Üretilen fişekler üretimin çeşitli aşamalarında kontrol edilmektedir. Bunlardan biri de kovan üretimidir. Kovan üretimi sürecinde hatalı olan kovanlar ayrılmakta ve üretim bandından çıkarılmaktadır. Kovanlar üzerindeki hataların tespit işlemi gözle muayene tezgâhlarında yapılmaktadır. Gözle muayene işlemi [2] özel olarak eğitilen elemanlar tarafından yapılmaktadır. Gözle muayene işleminde insan faktörü devreye girdiğinden gözle muayene yapan operatörlerin dikkatine, özenine, psikolojik ve fizyolojik durumuna bağlı olarak ürün kalite standardı değişmektedir. Bu ise ürün kalite standardını bozan bir unsur olarak karşımıza çıkmaktadır Uygulamada çalışanların dikkatlerini sürekli olarak yaptıkları işe vermeleri çok zor olduğundan belli aralıklar ile gözlerini dinlendirmeleri gerekmektedir.

MKE Gazi fişek fabrikasında kovan üretiminin kapasitesi her bir bantta dakikada 250 kovandır. Ancak gözle muayene tezgâhlarında üretim kapasitesi dakikada 125 kovandır. Yani her bir üretim bandı için iki adet gözle muayene tezgâhı kullanılmaktadır. Gözle muayene tezgâhlarında çalışanların sayısı ve çalışma zamanı dikkate alındığında fişek üretiminde gözle muayene işlemi üretim bandında maliyeti artıran bir unsur olarak da karşımıza çıkmaktadır.

Endüstriyel uygulamalarda otomasyon sistemleri çok yoğun bir şekilde kullanılmaktadır. İnsan emeği ile yapılan işlerin bilgisayarlar, mekanik aletler ve kendi kendine çalışan makineler ile yapılmasına otomasyon sistemi denmektedir. Başka bir deyişle; mekanik, elektronik ve bilgisayar tabanlı sistemleri kullanarak imalatın gerçekleştirilmesi ve kontrol edilmesidir. Otomasyon sistemlerinin kalitenin artırılması, maliyetin düşürülmesi, rekabet gücünün artırılması, üretimin standardizasyonu gibi pek çok avantajı bulunmaktadır.

Otomasyon sistemlerinde bilgisayarlı görme pek çok alanda farklı uygulamalar ile karşımıza çıkmaktadır. Bilgisayarlı görme endüstride hatalı parça bulma, ayıklama, ölçme, robotik, barkod okuma, karakter tanıma, nesne yakalama, takip etme, sayma, sınıflandırma, kalite kontrol gibi pek çok kullanım alanı bulmaktadır.

Bilgisayarlı görme, çeşitli kanallardan alınan verilerden bilgi çıkarma işlemidir. Günümüzde bilgisayarlı görme ile genellikle çeşitli kameralardan alınan veriler bilgisayarlar yardımıyla işlenerek istenen sonuçlar elde edilmektedir. İstenen sonuçların elde edilmesi yapılan işlemlerin karmaşıklığına bağlı olarak uzun zaman alabilmektedir. Endüstriyel uygulamaların pek çoğunda sistemler gerçek zamanlı olarak çalışmakta, bu nedenle de işlem süresi en önemli parametrelerden biri haline gelmektedir. Her ne kadar bilgisayarlardaki işlem gücü gün geçtikçe artsa da alınan verilerin miktarı ve kullanılan teknikler için gerekli işlem zamanı da artmaktadır.

Görüntü işleme teknolojilerinin gelişmesi ile bu teknolojilerin üretim hatalarının tespit edilmesinde kullanılması yaygınlaşmaktadır. Hataların tespitinde görüntü işleme tekniklerinin gerçek zamanlı uygulamalarda kullanılabilmesi için yüksek işlem gücü gerekmektedir. Bunun nedeni görüntü işleme uygulamalarının bazılarında işlem karmaşıklığının fazla olması ve dolayısıyla uzun işlem zamanı gerektirmesidir. Tabii ki donanımsal olarak bilgisayarların işlem gücünün artması işlem zamanını belli bir oranda kısaltmaktadır. Ancak bazı durumlarda işlem zamanını kısaltmak amacıyla bilgisayarların işlem gücünün artması yeterli olmamaktadır. Problemin durumuna göre farklı yazılım çözümlerine ve yaklaşımlarına ihtiyaç duyulmaktadır. İşlem zamanını kısaltmak ve uygulamanın gerçek zamanlı olarak gerçekleştirilmesini sağlamak amacıyla paralel programlama bir çözüm olarak karşımıza çıkmaktadır.

Bu çalışmanın amacı, MKE Gazi Fişek Fabrikasında üretilen ve üretim kapasitesi en fazla olan 7.62 mm çapındaki kovanların hatalı olanlarının otomatik olarak ayıklanmasını hızlı ve düşük maliyetli olarak gerçekleştirmektir. Bunun için, görüntü işleme teknikleri ve paralel veri işleme algoritmaları kullanarak hatalı olan kovanları ayıran hızı yüksek ve maliyeti düşük otomatik bir sistem geliştirmek hedeflenmektedir.

Önerilen sistemde üretim kapasitesi hedefi hat kapasitesinin de üzerinde olan dakikada 300 kovandır. Kuzey Atlantik Antlaşması Örgütü (NATO) standardında [3, 4] belirtilmiş olan kovan üzerindeki hatalar gözle görülür hatalardır. Bu nedenle kameralar ile alınan kovan görüntüleri kullanılmaktadır. Alınan kovan görüntüleri görüntü işleme teknikleri ve paralel veri işleme algoritmaları kullanılarak işlenmekte ve hatalı olan kovanlar belirlenmektedir.

Günümüzde sanayi ürünleri gelişen teknolojinin sonucu olarak ancak ürün kalitesini yükselterek ve üretim maliyetini düşürerek kendilerine pazar bulabilmektedirler. Gelişmekte olan savunma sanayimiz de ülkemiz ihtiyaçlarının karşılanmasının yanı sıra, dış pazarda kendisine pay bulmak zorundadır. Bu da ancak teknolojik olarak kendini sürekli geliştirerek mümkün olabilir. Bu çalışma ile hafif mühimmat üretiminde maliyetin düşürülmesi, kalitenin artırılması ve kalite standardının sağlanması amaçlanmaktadır. Üretim hattında hata bulma işlemi insan faktörüne bağlı olmayan, hata bulmayı ve hatalı olanları ayıklamayı otomatik olarak gerçekleştiren bir sistem geliştirilmektedir. Böylece güncel muayene sistemi uygulayan, NATO standartlarına uygun üretim yapan, yurtiçi ihtiyaçları karşılayan ve yurtdışına ihraç edilebilecek bir ürünün geliştirilmesine katkı verilmiş olacaktır.

Kovan hataları ağız hataları, kapsül hataları ve yan yüzey hataları olmak üzere üç kısımdan oluşmaktadır. Bu hataların bulunabilmesi için oluşturulan sistemde kameralar ile her bir kovan için bir adet ağız, bir adet kapsül ve sekiz adet yan yüzey olmak üzere toplam 10 adet görüntü alınmaktadır. Gerçek zamanlı sistemde alınan kovan görüntüleri görüntü işleme teknikleri kullanılarak işlenmekte ve hatalı olanlar tespit edilmektedir.

Kovanlar silindirik bir yapıya sahip metal yüzeyli nesnelerdir. Üretim sırasında kovanlar üzerinde çizik, vuruk, delik, renk değişimi, şekil bozukluğu gibi hatalar olabilmektedir. Bu tür hatalara sahip kovanlar, hatalı kovan olarak ayıklanmalıdır. Alınan görüntülerde kovanın silindirik şekli ve malzemesinden kaynaklanan parlama, yansıma gibi durumlar hatalı kovanların tespit edilmesini zorlaştırmaktadır. Bu sebeple alınan görüntülerde hatanın tespit

edilmesi için yapılan işlemler artmaktadır. Belirli bir sürede belirli sayıda kovan muayenesinin yapılma zorunluluğundan dolayı ihtiyaç duyulan işlem gücü de artmaktadır.

Gerçek zamanlı olarak çalışan sistemin kapasitesi dakikada 300 kovan olarak hedeflendiğinden, iki kovanın geçişi arasındaki süre en fazla 200 ms olmalıdır. İki kovanın geçişi arasındaki süre içerisinde kovanın bir mekanik sistemden geçişi, her bir kovan için 10 adet görüntünün alınması, alınan görüntülerin bilgisayara aktarılması, kovanın sistem üzerinden ayrılmadan görüntülerinin işlenerek hatalı olup olmadığına karar verilmesi ve hatalı ise ayrıştırılması işlemleri yapılmalıdır. Her bir kovandan alınan görüntü sayısı ve her bir görüntünün ayrı ayrı işlenme zorunluluğu göz önüne alındığında ihtiyaç duyulan işlem gücü yüksek olmaktadır. Bu nedenle bu tez çalışması kapsamında kovan hatalarını tespit etmek amacıyla algoritmalar geliştirilmekte ve işlem süresini kısaltmak amacıyla paralel işleme teknikleri kullanılmaktadır.

Görüntü işleme konusu son yılların en önemli araştırma alanlarından biridir. Görüntü işleme, insan görme sisteminin yaptığı işlemlerin bilgisayar ortamında gerçekleştirilmeye çalışılmasıdır. Endüstrinin birçok alanında süreç ve ürün denetimi, sınıflandırma ve kontrolde kullanılmaktadır [5]. Görüntü işleme çalışmalarında bir problemin çözümü için genel olarak bazı işlem basamakları kullanılmaktadır. Bunlar önişleme, özellik çıkarma ve sınıflandırma işlemleridir. Bu işlemler sıra ile gerçekleştirilerek görüntülerden bir sınıflandırma bilgisi elde edilir. Bu tez çalışmasında kovanların ağız, kapsül ve yan yüzeylerinden çekilen fotoğrafları işlenerek kovanların hatalı olup olmadığına karar verilmektedir. Farklı açılardan çekilen fotoğraflarda farklı özellikler analiz edilmekte, bunun için de farklı algoritmalar kullanılmaktadır. Ancak farklı işlemler gerçekleşmesine rağmen işlem basamakları her görüntü için uygulanmaktadır.

Sayısal görüntü işleme konusunda geliştirilen algoritmalar çok fazla işlem yüküne sahip olduğundan çok hızlı sistemler gerektirmektedir. Her ne kadar klasik bilgisayar sistemleri hız bakımından sürekli artış gösterseler de işlemleri seri olarak yürüttükleri için işlenen görüntünün boyutları arttıkça gerekli işlem zamanı da artmaktadır. Özellikle gerçek zamanlı görüntü işleme algoritmalarını işlemede hız bakımından ihtiyacı karşılayamamaktadır.
Paralel programlama bilgisayarda bir problemin çözümünün daha kısa sürede bitirilmesini sağlayan bir yaklaşımdır. Paralel programlama ile bir uygulamaya ait program parçaları birden fazla işlemcide aynı anda çalıştırılarak toplam çalışma zamanı düşürülmektedir.

Görüntü işleme uygulamaları da yüksek işlem gücü gerektirdiğinden paralel programlamanın kullanılması ile işlem zamanı düşürülebilir. Özellikle gerçek zamanlı uygulamalarda elde edilen görüntülerin işlenebilmesi amacıyla paralel programlama algoritmalarının kullanılması zorunlu hale gelmektedir.

Paralel sistemlerde ortak bellekli veya dağıtık bellekli sistemler kullanılmaktadır. Problemin durumu, donanım özellikleri, kısıtlamaları ve diğer özellikleri göz önünde bulundurularak paralelleştirilecek iş için hangi mimarinin kullanılacağına karar verilir. Her problemin kendine özgü farklı bir en iyi çözümü bulunabilir. Dağıtık bellekli sistemlerde paralelleştirme veri iletiminin belli bir süre almasında dolayı veri alışverişi en aza indirecek şekilde planlanır. Ortak bellekli sistemlerde işlemciler aynı bellek alanını kullanırlar ve dağıtık bellekli sistemlere göre veriye ulaşım ve iletimi daha hızlıdır. Bu tez çalışmasında kovan hatalarını tespit etmek için kullanılan gerçek zamanlı sistemin paralelleştirimesinde ortak bellekli sistemler kullanılmaktadır.

Ortak bellekli ve çok çekirdekli merkezi işlem birimlerinin (CPU) programlamasında kullanılan paralel programlama modelleri arasında OpenMP, TBB, Cilk bulunmaktadır. Çok çekirdekli işlemcilerin paralelleştirilmesinde kullanılan bu modeller 2012 yılında CernOpenLab'ın yaptığı bir araştırmada [6] karşılaştırılmıştır. Elde edilen verilere göre en iyi sonuçların OpenMP ile sağlandığı görülmektedir.

2000'li yıllardan sonra CPU'ların fiziksel sınırlarına gelinmesinden dolayı frekans artışı gerçekleştirilememiş onun yerine çok çekirdekli CPU'lar piyasaya sürülmüştür. Ancak çok çekirdekli CPU'ların da hızlanmaya etkisi belli bir düzeyde olmuş ve bu gücün kullanılabilmesi için yazılımların çok çekirdekli işlemcilerde çalışabilecek şekilde düzenlenmesi gerekliliği ortaya çıkmıştır. Bununla aynı zamanlarda üretilen grafik işlem birimlerinin (GPU) işlem gücü yüksek oranlarda artarak devam etmiş ve CPU işlem gücünün çok üzerine çıkmıştır. GPU'ların işlem gücünün artmasıyla genel amaçlı programlamada kullanımı artarak devam etmiştir. Son 10 yılda ekran kartlarının programlanmasında kullanılan platformlar ve diller çok gelişmiştir. Bu dillerden en önde gelenler OpenCL ve CUDA olmuştur. 2012 yılında Temizel ve diğerlerinin [7] yaptıkları çalışmada bu iki platform karşılaştırılmış ve CUDA'nın daha iyi sonuç verdiği görülmüştür.

Bu tez çalışmasında hem CPU çekirdeklerini kullanan hem de GPU'nun işlem gücünü kullanan hibrit bir sistem önerilmektedir. CPU çekirdeklerinin paralel çalıştırılmasında OpenMP, GPU çekirdeklerinin paralel çalıştırılmasında ise CUDA platformu kullanılmaktadır. Kovan hatalarının bulunmasında ağız, kapsül ve yan yüzey hataları OpenMP kullanılarak farklı işlemci çekirdeklerinde işlenecek şekilde sistem tasarımı gerçekleştirilmektedir. GPU programlamada en yaygın olarak kullanılan ekran kartı programlama ortamı olarak ta OpenCL'den daha iyi sonuçlar üreten CUDA platformu seçilmiştir.

Chakroun ve diğerleri [8] 2013'te yaptıkları çalışmada hem CPU çekirdeklerini hem de GPU'yu kullanarak bir optimizasyon algoritmasını hızlandırmışlardır. GPU ve daha fazla CPU çekirdeği kullandıklarında hızlanmanın daha yüksek olduğunu göstermişlerdir. Sanchez ve diğerleri [9] 2013'te yayınlanan çalışmalarında işlem zamanını kısaltmak için CPU ve GPU'nun işlemci gücünü birlikte kullanmışlardır. Çalışmada CPU'da OpenMP, GPU'da ise CUDA platformu kullanılmıştır. Görüntüdeki ani gürültüleri temizlemek amacıyla yüksek çözünürlüklü görüntülerin performans analizini CPU ve GPU'ya paylaştırarak gerçekleştirmişlerdir. Çalışmalarda ölçülen süreler göstermektedir ki GPU daha fazla görev üstlenmekle birlikte her iki aygıt birlikte çalışmalıdır. Elde edilen sonuçlara göre hedeflenen iş için GPU programlama ve çok çekirdekli işlemcilerin kullanılması daha kısa işlem zamanıyla çözüm üretilebileceğini desteklemektedir.

Yapılan çalışmalar hem CPU hem de GPU'nun birlikte kullanıldığı durumlarda performansın arttığını göstermektedir. Bu tez çalışmasında ağız ve kapsül hataları CPU çekirdeklerinde işlenirken yan yüzey hataları GPU'da işlenmektedir.

Bu çalışma yedi bölümden oluşmaktadır. Çalışmanın ikinci bölümünde görüntü işlemede hata bulma ve paralelleştirme hakkında şimdiye kadar yapılan çalışmalar incelenmektedir. Üçüncü bölümde sayısal görüntü işleme ve geliştirilen uygulamalarda kullanılan temel görüntü işleme teknikleri hakkında genel bilgiler verilmektedir. Dördüncü bölümde paralel programlama ve kullanılan paralel programlama teknikleri hakkında bilgi verilmektedir. Beşinci bölümde ise GPU programlama ve CUDA hakkında genel bilgiler ve bazı uygulamalar yer almaktadır. Çalışmanın altıncı bölümünde sistem tasarımı yapılmaktadır. Gerçekleştirilen sistem üzerinde kovan hatalarını bulmak için geliştirilen algoritmaların işlem basamakları ve performans ölçümleri verilmektedir. Elde edilen hızlanmalar

gösterilmekte ve değerlendirmeler yapılmaktadır. Son bölümde ise sonuçlar ve öneriler yer almaktadır.

2. LİTERATÜR TARAMASI

Bu tez çalışmasında kovan üzerindeki hataları bulmak için görüntü işleme teknikleri kullanılması önerilmektedir. Sistemin gerçek zamanlı çalışması için paralel işleme teknikleri ile hem CPU hem de GPU'nun birlikte çalışacağı bir platform önerilmektedir. Bu bölümde yapılan çalışma ile aynı alandaki çalışmalar incelenerek, benzerlikleri ve farklılıkları ortaya koyulmuştur. Literatür taraması iki ana başlık altında yapılmıştır. Bunlardan ilki hata bulma için görüntü işleme tekniklerinin kullanıldığı çalışmalar, ikincisi ise CUDA platformunda yapılan çalışmalar ile hem CPU hem GPU'nun birlikte kullanıldığı çalışmalardır.

2.1. Görüntü İşleme Teknikleri ile Hata Bulma Çalışmaları

Kovan muayene ile ilgili olarak literatürde herhangi bir çalışmaya rastlanmamıştır. Ancak çeşitli yüzeylerde hataların bulunmasıyla ilgili çalışmalar bulunmaktadır. Bunlar içerisinde de paralel algoritmaların kullanıldığı herhangi bir çalışmaya rastlanmamıştır. Yüzeylerdeki hataların tespit edilmesi ile ilgili çalışmalar içerisinde konuya en yakın olanlar ve metal yüzeylerde hata bulma ile ilgili olanlar seçilip incelenmiştir.

Metal yüzeylerde hataların bulunmasıyla ilgili Xue-wu ve diğerleri [10] 2011'de gerçekleştirdikleri çalışmada sınıflandırma için destek vektör makinelerini (SVM– Support Vector Machine) kullanarak yansıtıcı metal yüzeylerde hataların bulunmasını sağlayan bir bilgisayarla görme kontrol sistemi geliştirmişlerdir. Metal yüzeylerin belli bir boyuttaki siyah-beyaz görüntülerinde bulunan hataları ön işleme, öznitelik çıkarımı ve sınıflandırma olarak üç işlem basamağında işlemişler ve hatalı olanları yedi farklı sınıfa ayırarak bulmuşlardır. %75 ile %90 arasındaki başarı oranları ile hatalıları tespit etmişlerdir.

Metalik yüzeylerdeki hataları bulan başka bir çalışmada ise Zheng ve diğerleri [11] genetik algoritma kullanarak otomatik bir kontrol sistemi geliştirmişlerdir. Metal yüzeylerdeki parlama problemi için özel bir aydınlatma kullanmışlar ve hataları delik ve çatlak olarak iki sınıfa ayırmışlardır. %90 oranında hatalı yüzeyleri tespit etmişlerdir. Kendi çalışmaları için özel üretilen mekanik sistemde de özel aydınlatma sistemi kullanmışlardır.

Literatürde karşılaşıldığı kadarıyla silindirik yüzeyler üzerinde hata bulan çalışmalar ise şöyledir: Bu konuda rastlanan ilk çalışmada Rosati ve diğerleri [12] 2009'da yansıyan eğimli yüzeylerdeki hataları gerçek zamanlı olarak tespit etmişlerdir. Çekilen fotoğraflarda yansımanın olmaması veya en aza indirilmesi için kullanılacak ışık kaynağının konum ve açısını optimize etmişlerdir. 2011 yılında silindirik yüzeylerde hata bulma konusunda iki çalışmaya rastlanmaktadır. Bunlardan ilki Wang ve diğerlerinin [13] yuvarlak ve dönen nesnelerin yüzey hatalarını buldukları çalışmadır. Nesnenin görüntüsü üzerinde önişleme, bölütleme, tanıma ve hatayı sınıflandırma işlem basamaklarını sıra ile gerçekleştirerek hata tespiti yapmışlardır. Diğeri ise Agarwal ve diğerlerinin [14] silindirik yüzeylerdeki hataları sınıflandırmak için destek vektör makinelerini kullanan bir yaklaşım geliştirdikleri çalışmadır. Geleneksel çok sınıflı destek vektör makinelerinden farklı bir çoklu sınıf kullanarak silindirik yüzeylerdeki hataları sınıflandıran bir sistem geliştirmişlerdir.

Karşılaşılan diğer çalışmalar ise farklı yüzeylerde hata bulma çalışmalarıdır. Laofor ve Peansupap [15] 2012 yılında seramik yüzeylerdeki hataların bulunması ve kalitenin arttırılması amacıyla görüntü işleme tekniklerini kullanan bir kalite kontrol sistemi geliştirmişlerdir. Yaptıkları çalışmada yapılan iş dört bölüme ayrılmaktadır. Bunlardan ilki görüntünün alınmasıdır. Bir sonraki bölüm ise resme perspektif ayarlarının yapılması, gürültülerin kaldırılması, kenar çıkarma gibi işlemleri içeren önişlemedir. Üçüncü bölümde ise veri analizi ile bir sonuç elde edilmekte ve sınıflandırma işlemi gerçekleştirilmektedir. En son aşamada ise elde edilen sonuç uygulanmaktadır. Bu tez çalışmasında hata bulma algoritmasında da benzer adımlar bulunmaktadır. Elde edilen sonuçların başarısı değerlendirilmektedir.

Tsai, Wu ve Li [16] 2012'de Fourier dönüşümü kullanarak güneş panellerindeki üretim hatalarını tespit etmişlerdir. Hataların bulunmasında görüntü kalitesinin ve büyüklüğünün oldukça etkili olduğunu ve bununla birlikte çeşitli parametrelerin değiştirilmek zorunda kalındığını söylemişlerdir. Bu çalışma görüntü işleme teknikleriyle hata bulunmasının farklı bir alanda uygulamasıdır.

Ulusal çalışmalar arasında rastlanan konuya en yakın iki çalışmadan ilki 2011'de Samtaş ve Gülesin'in [17] farklı alanlarda sayısal görüntü işleme uygulamalarını araştırmalarıdır. İkincisi ise yine aynı yılda Çomak ve diğerlerinin [18] görüntü işleme tekniklerinin beton teknolojisinde kullanılması ile ilgili yaptıkları çalışmadır. Bu konuda yapılan çalışmaları ve

yöntemlerini incelemişlerdir.

Yapılan literatür taramasında yüzey üzerindeki hataları bulmak için görüntü işleme tekniklerinin kullanıldığı çalışmaların bulunduğu görülmektedir. Çalışmalar farklı yüzeyler için farklı yaklaşımlar ile çözüm üretildiğini göstermektedir. Metal yüzeyler üzerinde hata bulma çalışmalarında aydınlatma konusuna değinilmiştir. İyi sonuçlar elde edebilmenin ilk aşaması kaliteli ve işlenebilir görüntü elde edilmesidir. İşlenebilir görüntü alabilme görüntü işlemenin en önemli ve başlangıç adımıdır. Daha iyi görüntü alabilmek için aydınlatmanın önemli olduğu incelenen çalışmaların birkaçında belirtilmektedir. Özellikle metal veya silindirik yüzeylerde yapılan çalışmalarda özel aydınlatmaların kullanıldığı ve bu konuda çalışıldığı belirtilmektedir.

Bu tez çalışmasında üzerindeki hataların tespit edileceği mermi kovanı hem silindirik hem de metal yüzeylidir. Literatür çalışmalarından elde edilen sonuçlara göre alınan görüntülerde metal yüzeyde yansıma olmaması, kovan hatalarının bulunması için önerilen algoritmada sonucun doğruluğunu önemli ölçüde etkilemektedir. Alınan görüntülerde yansımanın en aza indirilmesi için özel aydınlatmaların kullanılması gerektiği tespit edilmiştir.

2.2. Görüntü İşlemede Paralelleştirme

Bu tez çalışmasında kovan hatalarını bulmak için kullanılan görüntü işleme algoritmalarının sistem üzerinde gerçek zamanlı olarak çalıştırılması için hızlandırılması gerekmektedir. Görüntü işleme tekniklerini hızlandırmak için kullanılan yöntemlerden biri GPU programlamadır. GPU programlamada en çok kullanılan platform ise CUDA'dır. CUDA platformu ilk olarak Şubat 2007'de yayınlanmıştır. CUDA platformu ilk olarak matrislerin çarpımı [19, 20] veya bilinen algoritmaların uygulanması [21-23] gibi genel amaçlı problemlerin çözümünde kullanılmıştır. Yapılan uygulamalarda elde edilen sonuçlar GPU teknolojilerinin ve CUDA platformunun hızla gelişmesini ve yaygınlaşmasını sağlamıştır.

Görüntü işlemede GPU kullanımı ile ilgili literatürde rastlanan ilk çalışma Diez ve diğerlerinin [24] 2008 yılında yaptıkları çalışmadır. Çalışmada farklı yöntemlerle görüntü geriçatma işlemi gerçekleştirilmiştir. Bu yöntemlerde GPU üzerinde paralelleştirilmiş ve işlem zamanları ve kazançları farklı çözünürlükteki resimler için gerçekleştirilmiştir.

Popüler görüntü geriçatma algoritmalarından ağırlıklı geri beslemeli algoritmalardan sürekli iteratif geriçatma (SIRT) ve sürekli cebirsel geriçatma (SART) teknikleri GPU üzerinde uygulanmıştır. CPU ile karşılaştırıldığında 60 ile 80 kat arasında kazanç elde edilmiştir.

Stone ve diğerlerinin [25] 2008 yılında yaptıkları çalışma görüntü işlemede CUDA kullanılmasının ilk örneklerindendir. Çalışmada manyetik rezonans (MR) görüntüleri üzerinde geriçatma algoritmaları GPU üzerinde çalışacak hale getirilerek CUDA 1.0 ve CUDA 1.1 karşılaştırması yapılmıştır. Ayrıca CUDA çalışırken verilerin hafızadaki hareketi şematik olarak gösterilmiştir. Hafızada sabit bellek ve yazmaçlara verilerin aktarım farkının hıza etkisi gösterilmektedir. Çalışmada elde edilen sonuçlara göre global belleği çok kullanan uygulamalar daha yavaş çalışırken yazmaçları ve sabit belleği kullanan uygulamalar daha hızlı çalışmaktadır. 2 ile 9 kat arasında hızlanma sağlanmıştır.

Che ve diğerleri [26] 2008 yılında yaptıkları çalışmada OpenMP ve CUDA konusunda kapsamlı bir araştırma yayınlamışlardır. Belirli bir uygulama olmamakla birlikte o dönemde çok yeni bir konu olmasından dolayı genel sonuçlar ortaya koymuşlardır. Bellek kullanımı ve yığın çatışmasının (bank conflict) oluşumu anlatılmakta ve kazancı artırmak için yığın çatışmasının önüne geçilmesi gerektiği belirtilmektedir. Elde edilen sonuçlara göre 70 kata kadar hızlanma sağlanmıştır. Başka bir çalışmada ise Qui ve diğerleri [27] görüntü öznitelik çıkarımı tekli komut çoklu veri (SIMD) ile iş parçacıklarının paralelleştirilmesini kullanarak gerçekleştirmişlerdir. Çalışmada öznitelik çıkarımı gerçekleştirilmiş ve hızlandırmak amacıyla paralelleştirilmiştir. Görüntüde öznitelik çıkarımı farklı çözünürlüklü veriler üzerinde gerçekleştirilmiştir. Elde edilen sonuçlara göre 8 kata kadar hızlanma sağlanmıştır.

Bergen ve diğerleri [28] 2015 yılında görüntü bölütleme konusunda çalışmışlardır. 4 boyutlu (3 boyut uzaysal, bir temporal) MR görüntülerinin veri setlerinin sayısı ve boyutu büyüdükçe etkili bölütleme algoritmalarına ihtiyaç duyulmaktadır. Faz kontrast MR görüntülerinden kalp akış bilgisinin ayrılması işlemi bu çalışmada gerçekleştirilmiştir. Faz kontrast bölütleme algoritması basit ortalama tabanlı hesaplama ve en küçük kareler yöntemiyle gerçekleştirilmiştir. Bu bölütleme işlemi çok çekirdekli CPU'da 10 saniye (s) kadar sürmektedir. GPU tabanlı algoritma ile ise 0,5 s'de gerçekleştirilmiştir. Görüntü bülütlemede GPU kullanımına örnek başka bir çalışma D'ambra ve Pilippone'un [29] 2016 yılında yaptıkları çalışmadır. 2 boyutlu (2B) görüntüler üzerinde PSBLAS tabanlı bir bölütleme işlemi gerçekleştirmişler ve hızlandırmak için GPU'yu kullanmışlardır. Seriye

göre 14 kata kadar hızlanma elde etmişlerdir.

Swargha ve Rodrigues [30] çalışmalarında kenar yakalama algoritmalarından bazılarını paralelleştirmişlerdir. Kenar yakalama resim üzerinde görüntünün parlaklığını değiştirerek hedeflenen noktaların tanımlanmasını hedef alan bir terminolojidir. Burada amaç resim üzerinde derinlik gibi devam etmeyen kısımların keskinleştirme ile yakalanmasıdır. Bu çalışmada OpenCL kullanılarak eşzamanlı programlama ile paralellik sağlanmıştır. Bu sayede algoritma daha hızlı çalışmaktadır. Sobel, Laplace ve Canny algoritmalarında sırası ile 25, 39 ve 50 kez hızlanma sağlanmıştır.

Görüntü işleme alanının bir dalı olarak görüntü çakıştırma konusunda da GPU uygulamalarına rastlamaktayız. Zhang ve diğerleri [31] 2014 yılında yaptıkları çalışmada görüntü çakıştırma işlemi için kullanılan kabadan inceye (coarse-to-fine) yöntemini GPU ile hızlandırmışlardır. Yüksek çözünürlüklü görüntülerde algoritma çalıştırılmış ve farklı yöntemlerle olan sonuçlar verilmiştir. İşlemlerin çoğunda GPU uygulamasında CPU'ya göre hızlanma sağlanmıştır.

Li ve diğerleri [32] 2015 yılında yayınladıkları çalışmalarında yine görüntü çakıştırma işlemini GPU ile paralelleştirmişlerdir. Çalışmada beş farklı korelasyon oranı tabanlı metot geliştirmişlerdir. Geliştirdikleri algoritmaları Fermi, Kepler ve Maxwell mimarilerinde çalıştırmışlardır. CPU uygulamasına göre dört GPU ile 55 kata kadar hızlanma elde etmişlerdir. 145 s olan görüntü çakıştırma süresini 2,6 s'ye kadar indirmişlerdir.

Başka bir çalışmada ise Shams ve diğerleri [33] medikal görüntülerde görüntü çakıştırma işlemini gerçekleştirirken histogram hesaplamada ortak bilgi temeli yöntemini kullanmaktadırlar. Kullandıkları yöntemi grafik işlemciyi kullanarak paralel hale getirmek için bir algoritma geliştirmişlerdir.

Saha ve diğerleri [34] görüntü iyileştirme algoritmalarından bazılarını CUDA kullanarak paralelleştirmişlerdir. Alçak geçiren filtre, yüksek geçiren filtre, Sobel kenar yakalama algoritmasını GPU ile paralelleştirmişler ve 3000X3000 çözünürlükteki görüntüler üzerinde elde edilen hızlanma sonuçlarını vermişlerdir. Sobel kenar yakalama algoritmasında 11 kat, alçak ve yüksek geçiren filtrelerde 12 kat hızlanma elde etmişlerdir.

Quin ve diğerleri [35] yaptıkları çalışmada video görüntülerinde arka zemin çıkarmayı GPU ile hızlandırmışlardır. Arka zemin çıkarma yöntemi ile hareket eden nesneleri tespit etmektedirler. Bunun için Gabor filtresini kullanmışlardır. Gabor filtresi kullanılarak ve kullanılmadan video görüntülerinde arka zemin çıkarma yöntemi ile hareket algılaması gerçekleştirmişlerdir. Gabor filtresi uygulanmadan 10 kat hızlanma sağlanırken, Gabor filtresi ile 14,5 kat hızlanma elde edilmiştir.

Werff ve Bakker [36] 2014 yılında yaptıkları çalışmada görüntü sınıflandırma için en çok kullanılan mesafe ölçme algoritmaları kullanmışlardır. Çalışmada Öklit uzaklığı ve iki vektörün benzerliği olmak üzere iki farklı algoritma kullanılmaktadır. Verilen sonuçlarda GPU'ya veri transferinin ne kadar zaman aldığı ayrıca verilmektedir. Çalışma zamanı veri transferleri ve kernel süresi olarak ayrıca belirtilmektedir. İki vektörün benzerliği (spectralangle) metodu ile Öklit uzaklığına göre daha iyi hızlanma sağlanmıştır.

Ganiau ve Onchiş [37] tarafından 2014 yılında gerçekleştirilen çalışmada yüz tanıma noktalarının işaretlemesinde Gabor kareleri GPU kullanılarak uygulanmıştır. Matris çarpımı, Hızlı Fourier Dönüşümü (FFT), kenar yakalama, yüz tanıma algoritmaları GPU'da çalıştırılmıştır. CPU ve GPU işlem zamanları tablo halinde verilmektedir. Matris çarpımında 48 kat, FFT'da 10 kat, kenar yakalama algoritmasında 9 kat, yüz tanıma algoritmasında 5 kat hızlanma sağlanmıştır.

Akgün ve diğerleri [38] 2015 yılında yayınlanan çalışmasında MR görüntülerinde dinamik fonksiyonel bağlantı analizini paralelleştirmişlerdir. Paralelleştirme ayrı ayrı CPU ve GPU kullanılarak gerçekleştirilmiştir. CPU çekirdeklerini kullanmak amacıyla OPENMP kullanılmıştır. GPU uygulamasında iki farklı yöntem kullanılmıştır. Bunlar iş parçacığı tabanlı ve blok tabanlı olarak isimlendirilmiştir. Blok tabanlı yöntemde paylaşımlı bellek kullanılmıştır. OPENMP kullanılarak sekiz çekirdekli işlemci ile gerçekleştirilen algoritmada 7,7 kata kadar hızlanma sağlanmıştır. GPU ile gerçekleştirilen algoritmada ise iş parçacığı tabanlıda 18,5 kata kadar, blok tabanlıda ise 157 kata kadar hızlanma sağlanmıştır.

Chouchene ve diğerleri [39] 2015 yılında yaptıkları çalışmada yüz yakalama için kullanılan Viola ve Jones algoritmasını GPU ile paralelleştirmişlerdir. Paylaşımlı bellek kullanarak ve kullanmadan iki farklı uygulama yapmışlardır. 16 kata kadar hızlanma elde etmişlerdir.

Lastra ve diğerleri [40] 2015 yılında yayınlanan çalışmalarında biyometrik tanımlama sistemlerinde sıklıkla kullanılan parmak izi tanıma konusunda çalışmışlardır. Parmak izi tanıma işlemlerinde veri tabanının büyüklüğüne bağlı olarak işlem zamanı yükselmektedir. Parmak izi tanıma sistemlerinde tek bir parmak izinin tanınması daha kısa zaman alırken milyonlarca parmak izinin taranması çok fazla zaman almaktadır. Bu çalışmada parmak izi tanıma sisteminde GPU kullanılarak işlem zamanı kısaltılmıştır. Yapılan çalışmada tek GPU ile 15 kata kadar, dört GPU ile ise 54 kata kadar hızlanma sağlanmıştır. Elde edilen sonuçlara göre paralel olarak saniyede 300.000–1.500.000 adet parmak izi taranabilmiştir.

Zhu ve diğerleri [41] 2012 yılında yaptıkları çalışmada vücuda enjekte edilen sıvıların dağılımını görüntü işleme kullanarak işlemişlerdir. Bu çalışmalarında işlem zamanını düşürmek amacıyla genel amaçlı GPU programlamayı kullanmışlardır. İşlem zamanını özellikle yüksek çözünürlüklü görüntüler için düşürmüşlerdir. 5,56 kata kadar hızlanma sağlamışlardır. Çalışma sonuçlarına göre veri miktarı arttıkça GPU programlamada hızlanma artmaktadır.

Paralel programlamada birden fazla işlemci kullanılarak işlem zamanını azaltma hedeflenmektedir. GPU'lar sahip oldukları çok sayıda çekirdek ile yüksek işlem gücüne sahiptirler. Ancak bu gücün kullanılabilmesi için GPU üzerinde çalışan uygun yazılımlara ihtiyaç vardır. Klasik algoritmalar ve kodlar CPU'da tek çekirdek üzerinde çalışmaktadır. Bu algoritmaların GPU üzerinde çalıştırılabilmesi için çekirdekleri mümkün olduğunca verimli kullanabilen kodlar haline getirilmesi gerekmektedir. Bu durum, algoritmaya bağlı olmakla birlikte genel olarak zor ve zahmetli bir iştir. Çünkü algoritma tasarımı yapılırken iş yükünün çekirdeklere paylaştırılması, bellekler arasındaki veri alışverişi, işlemcilerin bellekler ile iletişimi programcı tarafından planlanmalıdır. İşlemcilerin eş zamanlı çalışması sırasında aynı bellek bölgesine erişim, bellekteki verilerin farklı işlemciler tarafından değiştirilmesi, senkronizasyon gibi konular programcının göz önünde bulundurması gereken noktalardır. Bu durum göz önüne alındığında bir algoritmanın paralel kodu seriye göre daha karmaşıktır.

İncelenen çalışmalarda GPU konusunda ilk yapılan çalışmaların temel algoritmalar olduğu görülmektedir. Bu algoritmalarda elde edilen hızlanmalar GPU konusunda yapılan çalışmaları hızlandırmış ve farklı görüntü işleme algoritmalarının GPU uygulamaları gerçekleştirilmiştir. İncelenen görüntü işlemede GPU uygulamalarında 145 kata kadar

hızlanma elde edildiği görülmektedir. Daha yakın dönemlerde yapılan çalışmalarda ise bir problemin çözümüne yönelik çalışmaların bulunduğu görülmektedir. Bu çalışmalarda elde edilen hızlanmalar temel algoritmalardaki hızlanmalar kadar olmamakla birlikte incelenen çalışmalar içerisinde 16 kata kadar hızlanma elde edildiği görülmüştür.

İncelenen çalışmalarda çeşitli algoritmaların paralelleştirilip farklı platformlarda çalıştırıldığı görülmektedir. Algoritmaların seri çalışma süreleri ve paralel çalışma süreleri karşılaştırılarak hızlanmalar gösterilmektedir. GPU uygulamaları tek çekirdekli ve çok çekirdekli işlemcilerle karşılalaştırılmaktadır. GPU programlama, çok çekirdekli işlemcilerin paralel programlanması ya da bunların birlikte kullanıldığı çalışmalar bulunmaktadır.

Literatür taramasından elde edilen çıkarımlar neticesinde, bu tez çalışması kapsamında yapılacak işin bir bölümünü GPU'ya bir bölümünü de CPU'ya yaptırarak işlem zamanını kısaltmak amacıyla hibrit bir algoritma gerçekleştirilmesinin gerekliliği ortaya çıkmıştır. Ekran kartının işlemci gücünü kullanmak amacıyla GPU programlama kullanılacaktır. Ayrıca çok çekirdekli bilgisayarların paralel programlaması için eş zamanlı çoklu iş parçacığı kullanılacaktır. Bu iki yapının birlikte kullanıldığı hibrit bir yapının oluşturulması bu tez çalışmasında önerilmektedir.

3. GÖRÜNTÜ İŞLEME

Görüntü işleme son yılların en önemli araştırma konularından biridir. Görüntü işleme teknikleri tıp, coğrafi bilimler, uzay bilimleri, savunma sanayi, güvenlik sistemleri vb. gibi pek çok alanda kullanılmaktadır.

Görüntü işleme, sinyal girişinin bir görüntü (fotoğraf veya bir video karesi), çıkışın ise yine bir görüntü veya görüntüye ait karakteristikler olduğu bir sinyal işleme sürecidir. Sayısal görüntü işleme, sayısal görüntülerin bilgisayarla işlenmesidir. Sayısal görüntü işleme teknikleri insan görmesi işlevlerini bilgisayar ortamında modellemeye çalışmaktadır. İnsan görmesi karmaşık bir işlemdir ve bu konuda pek çok çalışma yapılmasına rağmen ancak bir kısmı modellenebilmiştir. Aynı şeyi amaçlamalarına karşın görüntü işleme teknikleri insan görmesinden tamamen farklıdır.

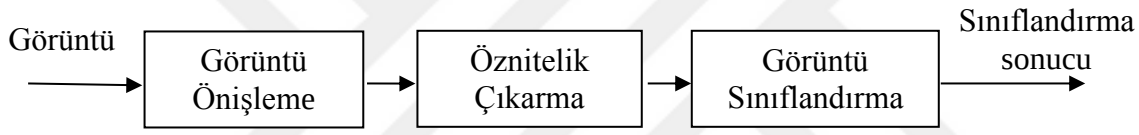
Sayısal görüntü, piksel adı verilen ve her birinin özel bir konumu ve değeri olan sonlu sayıdaki elemanlardan meydana gelmektedir. Görüntü iki veya üç boyutlu olarak piksellerin bir araya gelmesiyle oluşur. Sayısal görüntü işlemede geliştirilen algoritmalar genellikle her bir pikselin ayrı ayrı işlenmesini gerektirir. Görüntü çözünürlüklerinin gün geçtikçe artmasıyla bu işlemlerin yapılabilmesi için gerekli işlem gücü de artmaktadır. İşlem gücüne olan ihtiyacın artması paralel işlem yapabilen çok hızlı, yoğun ve bağımsız görüntü işleme yeteneğine sahip sistemlerin geliştirilmesi her geçen gün daha fazla önem kazanmaktadır. Bu ihtiyacı karşılamak için görüntü işleme uygulamalarını gerçek zamanlı olarak çok hızlı bir şekilde gerçekleyebilen farklı sistemler geliştirilmektedir [42].

Görüntü işleme alanında çalışılan konular aşağıdaki başlıklar altında gruplandırılabilir. Bunlar;

- Görüntü geriçatma
- Görüntü iyileştirme
- Görüntü bölütleme
- Görüntü çakıştırma
- Görüntü sınıflandırma'dır.

Her bir başlık altında pek çok algoritma kullanılmaktadır. Bu algoritmalardan bazılarının bilinen algoritmalar olmasıyla birlikte her alanda yeni çalışmalar yapılmakta ve farklı yöntemler önerilmektedir. Kullanılan algoritmaların başarısı uygulamaya göre değişiklik göstermektedir. Bu algoritmalardan bir veya bir kaçı belli bir amaca ya da bir problemin çözümüne yönelik kullanılabilir.

Görüntü işleme çalışmalarında öncelikle kameradan görüntülerin alınması işlemi gerçekleştirilir. Alınan görüntüler önişleme yapılarak işlenmeye uygun hale getirilir. Ardından uygulamaya göre değişiklik gösteren ve temel işlemlerin yapıldığı görüntünün tüm özelliklerinin belirlendiği işlem adımı olan öznitelik çıkarma gerçekleştirilir [43]. Son adım olarak da elde edilen bilgiler değerlendirilerek sınıflandırma yapılır. Şekil 3.1'de işlem basamakları gösterilmektedir;



Şekil 3.1. Görüntü işleme işlem basamakları

Görüntü işlemenin ilk aşaması önişleme aşamasıdır. Ön işleme ise görüntü üzerindeki gürültülerin temizlenmesi ve geometrik düzeltme işlemlerinin yapılması ile görüntünün işlenmeye hazır hale getirilmesi işlemidir.

Görüntü üzerindeki nesnelerin tespit edilmesi için yapılan çalışmalar öznitelik çıkarma adımına girer. Bu adım probleme göre farklı alt adımların uygulandığı ve verilerin analizinin yapıldığı işlem basamağıdır. Şekil tanıma, kenar tanıma, örüntü çıkarma gibi pek çok işlem ya da işlemler bu adımda gerçekleştirilir. Bu adım yapılan uygulamaya göre farklı algoritmaların kullanıldığı işlem kısmıdır. Öznitelik çıkarma adımında işlenen görüntülerden istenen veriler elde edilir.

Sınıflandırma aşaması problemin çözümüne yönelik bir değerlendirmedir. İstenen sonuçların elde edilmesi için gerekli algoritmalar çalıştırılır. Sınıflandırma yöntemleri kontrollü ve kontrolsüz olmak üzere iki gruba ayrılmaktadır. Eğitim verisi kullanılmadan yapılan sınıflandırma işlemine kontrolsüz, eğitim verisi kullanılarak yapılan sınıflandırma işlemine kontrollü sınıflandırma denir. K en yakın komşu (kNN), destek vektör makineleri

(SVM) en çok kullanılan yöntemlerdendir. Sınıflandırma işlemi sonrasında oluşan sınıfların doğruluklarını belirlemek amacı ile yapılan işleme doğruluk analizi denir.

Görüntü işleme alanında çalışılan konular şu şekildedir:

3.1. Görüntü Geriçatma (Image Reconstruction)

Görüntü geriçatma nesnelerin verilere dönüştürülmesi işlemlerinin tümüdür. Nesnelerin iki boyutlu (2B) ve üç boyutlu (3B) yüzey görüntüleri kameralar kullanılarak elde edilebilir. Öte yandan nesnelerin. kesitlerinin görüntüsünü elde etmek için özel cihazlar ve sensörler kullanılmaktadır. Bu cihazlarla toplanan verilerin matematiksel işlemlere tabii tutularak bilgisayar ortamında görüntüye dönüştürülmesi işlemine görüntü geriçatma denir. Görüntünün oluşturulması için gerçek bir nesneden cihazlar ile farklı açılardan elde edilen sinyallerin her biri dönüştürülüp tek bir görüntü üzerinde birleştirilerek gerçek nesneye en yakın görüntü elde edilmeye çalışılır.

3.2. Görüntü İyileştirme (Image Enhancement)

Görüntü iyileştirme alanında uygulanan tekniklerin hedefi, görüntünün niteliğini istenilen ve beklenen şekilde yükseltmektir. Görüntü iyileştirme, gerçek görüntüden farklı olarak alınan görüntülerde var olan gürültü, bulanıklık, renk, ışık vb. gibi problemlerin ortadan kaldırılması ve görüntünün mümkün olduğunca gerçeğine yakın hale getirilmesi işlemlerinin tümüdür. Görüntü işleme adımlarının başarısını artırmak için de görüntü iyileştirme algoritmaları kullanılabilir. Kontrast ve parlaklık düzenlemeleri, kenar zenginleştirme, histogram eşitleme gibi uygulamalar, kişinin görsel algılamasını etkilemeyi hedefleyen ve bu konuda başarı sağlayabilen en temel görüntü iyileştirme teknikleridir.

3.3. Görüntü Bölütleme (Image Segmentation)

Görüntü bölütleme görüntüyü her biri içerisinde farklı özelliklerin tutulduğu anlamlı bölgelere ayırma işlemidir. Uygulamaya bağlı olarak kullanılan yöntemler ve yöntemlerin performansı değişiklik gösterebilir. Tüm görüntülere uygulanabilecek genel bir bölütleme yöntemi yoktur [44]. Yani görüntü bölütleme için tasarlanan yöntemler ve bu yöntemlerin başarımları uygulamaya göre değişiklik göstermektedir. Bu nedenle de üzerinde çok

çalışılan konulardan biridir. Görüntü bölütleme, görüntü işlemenin en zor ve en önemli işlemlerinden biri olmakla birlikte pek çok farklı yaklaşımı da içermektedir [45, 46]. Bu yaklaşımlar; histogram tabanlı, kümeleme tabanlı, ayırma ve birleştirme, bölge büyütme (region growing) ve morfolojik olarak gruplandırılabilir.

3.4. Görüntü Çakıştırma (Image Registration)

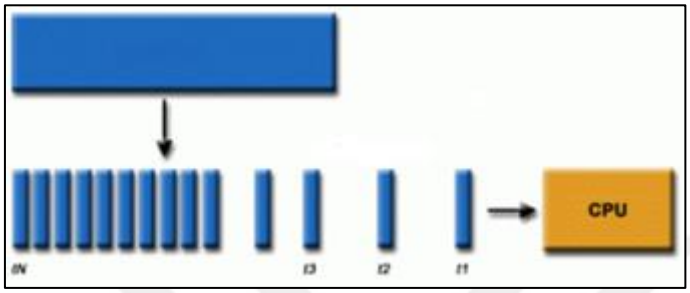
Görüntü çakıştırma farklı kaynaklardan, farklı açı veya farklı zamanlarda elde edilen görüntüler içerisindeki ortak alt görüntülerin her noktası geometrik olarak üst üste gelecek şekilde hizalanması olarak tanımlanır. Görüntü çakıştırmada öteleme, döndürme ve boyutlandırma işlemleri gerçekleştirilir. Bu konuda kullanılan pek çok algoritma bulunmaktadır.

3.5. Görüntü Sınıflandırma (Image Classification)

Sınıflandırma birçok bilim dalında bir karar verme işlemidir. Görüntü sınıflandırması görüntünün çeşitli özelliklerini analiz eder ve onları kategorilere ayırır. Görüntü sınıflandırma teknikleri kullanılırken iki farklı yaklaşım vardır [47]. Bunlardan birincisi piksel tabanlı görüntü sınıflandırma yaklaşımıdır. Bu yaklaşım eğitimli ve eğitimsiz görüntü sınıflandırma olarak ikiye ayrılmaktadır. Bu sınıflandırma yaklaşımında görüntü üzerindeki piksel değerlerinden çeşitli algoritmalar kullanılarak sınıflandırma gerçekleştirilir. Diğeri ise nesne tabanlı görüntü analizi kullanarak yapılan sınıflandırmadır [48]. Bu yaklaşımla eldeki görüntülerin sınıflandırılması için diğer görüntü işleme teknikleri kullanılmaktadır. Komşu piksellerin gruplandırılması ve nesnelerin ölçeklerine göre ayarlanması işlem basamakları içerisinde yer alır. Sınıflandırma için genellikle diğer görüntü işleme tekniklerinden bir veya birkaçı uygulanır. Elde edilen sonuçlar değerlendirilerek sınıflandırma gerçekleştirilir.

4. PARALEL PROGRAMLAMA

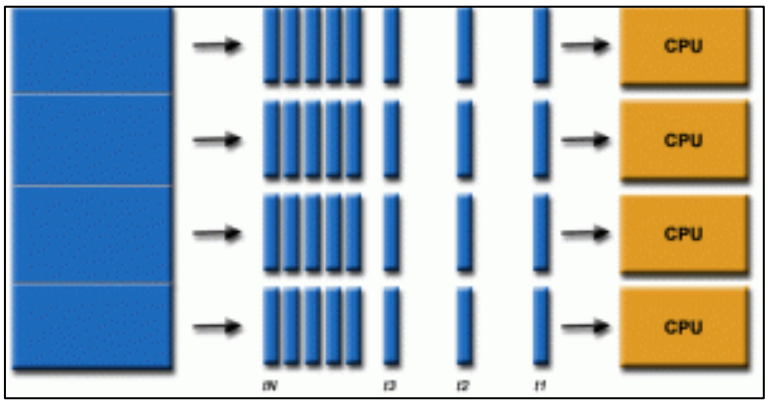
Geleneksel yöntemlerde programlar seri olarak yazılırlar. Program bir bilgisayarda bir işlemci tarafından yürütülür. Problem farklı komut serilerine bölünür. Her komut bir diğerinden sonra işleme alınır. Birim zamanda sadece bir komut işlenir (Bkz. Şekil 4.1).



Şekil 4.1. Seri kod işlemenin şematik gösterimi

Paralel hesaplama, daha hızlı sonuç almak için bir uygulamaya ait program parçalarının birden fazla hesaplama kaynağında aynı anda çalıştırılmasıdır. Bir işin paralelleştirilmesinde programın toplam çalışma zamanını azaltmak amaçlanır.

Problemin çözümünde birden çok işlemci kullanılır. Problem eşzamanlı çözülmek için farklı parçalara bölünür. Her parça farklı komut serilerine bölünür. Komutlar aynı anda farklı işlemciler üzerinde işlenir (Bkz. Şekil 4.2). Bu işlemciler tek bir makine üzerinde olabileceği gibi bir ağ ile bağlı bilgisayarlar üzerinde de olabilmektedir. Böylece her bir işlemciye düşen işlem sayısı azaltılarak işlem süresini azaltmak mümkün olmaktadır.



Şekil 4.2. Paralel kod işlemenin şematik gösterimi

Paralellik ile ilgili ilk fikirler, ilk olarak 1958'de IBM firması araştırmalarında nümerik hesaplamaların yapılabileceği fikrini ortaya atan John Cocke ve Daniel Slotnick ile ortaya çıkmıştır. Paralellik fikrinin gelişimi aşağıdaki gibidir:

- 1960'ta Novosibirsk Matematik Enstitüsünden (IMN) E. V. Yevreinov'un birbirine bağlantılı şekilde programlanabilen paralel mimarileri tasarlamasıyla paralel kavramı yeni bir boyut kazanmıştır.
- 1964'de ise Daniel Slotnick tarafından Lawrence Livermore National Laboratuarlarında kullanılmak üzere büyük paralel makineleri geliştirilmiştir.
- 1967'de Amdahl ve Daniel Slotnick tarafından paralel işlemlerin olasılığıyla ilgili bir makale yayınlanmıştır. Bu paralellik ile ilgili olan tartışma konusu daha sonra "Amdahl Yasası" olarak adlandırılmıştır.
- 1983'te Goodyear Aerospace, Ulusal Havacılık ve Uzay Dairesi (NASA) Goddard için Büyük Ölçekli Paralel İşlemci (MPP)'yi geliştirmiştir.
- 1985'te David Gelernter tarafından Linda paralel programlama sisteminin temellerini oluşturulmuştur.
- 1986'da Paralel Sanal Makine (PVM) projesi dağıtık bilgisayarlarda gerekli olan yazılımların kullanılabilmesi için geliştirilmiştir.
- 1993'te IBM firması Azaltılmış Komut Kümeli Bilgisayarlar (RISC) RS/6000 işlemcisine dayalı ilk SP1 Power Parallel sistemi piyasaya sürmüştür.
- 1993'te Mesaj Geçiş Arayüzü (MPI) standardı dağıtık bilgisayarlarda kullanılmak üzere PVM'e alternatif olarak yayınlanmıştır.
- 1994'te NASA'da Beowulf projesi adı altında 16 bilgisayardan oluşan ilk küme bilgisayar sistemi oluşturulmuştur.
- 2007'de NVIDIA firması GPU üzerinde çalışan CUDA platformunu geliştirmiştir.
- 2008 yılında Apple tarafından Khronos Grup'a OpenCL platformu önerilmiştir [49].

Michael J. Flynn paralel bilgisayar mimarilerini komut ve veri akışlarına göre dört grupta sınıflandırmıştır: Tekli komut tekli veri (SISD), tekli komut çoklu veri (SIMD), çoklu komut tekli veri (MISD), çoklu komut çoklu veri (MIMD). SISD sistemler tek komut ve tek veri mantığına göre çalışmaktadırlar. Kullandığımız tek işlemcili ve bellekli sistemler bu gruba girmektedir. SIMD sistemler bir komutu birden çok veri üzerinde işleyen sistemlerdir. Bu tür sistemler günümüzde pek çok uygulama alanı bulmaktadır. Farklı veriler üzerinde aynı komut setinin çalıştırılması ile uygulanırlar. MISD sistemler aynı veri üzerinde farklı

komutların işlemesiyle uygulanan sistemlerdir. MIMD sistemlerde hem birden fazla veri hem de çoklu komut seti kullanılmaktadır. Birbirinden bağımsız işlemlerin paralelleştirilmesinde kullanılmaktadır.

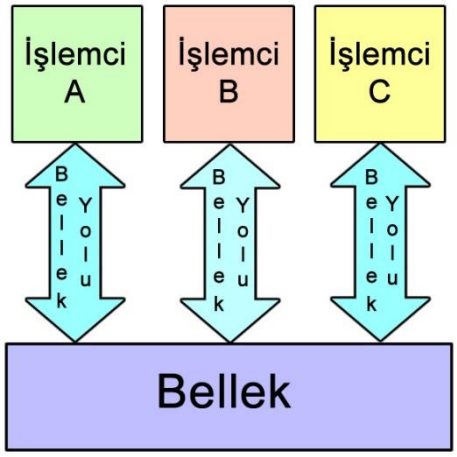
Bir paralel programlama modeli, paralel algoritmaları açıklayan bir yazılım teknolojileri kümesidir. Bu model; uygulamalar, diller, derleyiciler, kütüphaneler, iletişim sistemleri ve paralel giriş/çıkış alanlarını kapsar.

Programcı problemin özelliğine göre kullanacağı modeli seçip uygular. Birden fazla model aynı problem üzerinde uygulanabilir. Bu modeller ortak bellekli ve dağıtık bellekli olmak üzere iki gruba ayrılırlar. Problemin durumu, donanım özellikleri, kısıtlamaları ve diğer özellikleri göz önünde bulundurularak paralelleştirilecek iş için hangi mimarinin kullanılacağına karar verilir. Her problemin kendine özgü farklı bir en iyi çözümü bulunabilir.

4.1. Ortak Bellekli Sistemler

Bu tür sistemlerde birden çok işlemci aynı fiziksel belleği paylaşır ve genel olarak bütün işlemciler belleğe global adres uzayından erişirler. Bir işlemcinin bellek alanında yaptığı değişiklik bütün işlemciler tarafından görülür. Burada bellekten kasıt bilgisayar birincil hafızası yani rasgele erişimli bellek (RAM)'tir.

Bu tür sistemlerde ortak belleğin kullanılması haberleşme ve senkronizasyon açısından programcılara kolaylık sağlamaktadır. Ancak bir takım kısıtlamaları da bulunmaktadır. Örneğin bu tip sistemlerin yüksek performanslı belleklere ihtiyaçları vardır. İşlemci sayıları sınırlıdır. Şekil 4.3'te görüldüğü gibi aynı bellek kullanan belli sayıda işlemci ile işlem gücü arttırılmaktadır.



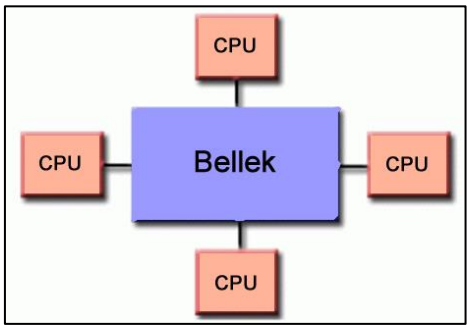
Şekil 4.3. Ortak bellekli bilgisayar mimarisi şematik gösterimi

Birden çok çekirdeğe sahip işlemciler ve grafik işlemcilerin paralel programlaması bu ortak bellekli sistemlerin paralelleştirilmesine örnektir. İş parçacığı seviyesinde paralelleştirme de yine bu gruba girmektedir. İş parçacıkları programcıların programı eş zamanlı parçalara ayırarak ortak belleği kullanarak bu parçaları eşzamanlı olarak çalıştırmalarıyla oluşturulan program parçacıklarıdır.

Ortak bellek kullanan bilgisayarlar belleğe ulaşım zamanlarına göre iki ana sınıfa ayrılırlar: Tekdüze bellek erişimi (UMA) ve tekdüze olmayan bellek erişimi (NUMA).

4.1.1. Tekdüze bellek erişimi (UMA-Uniform Memory Access)

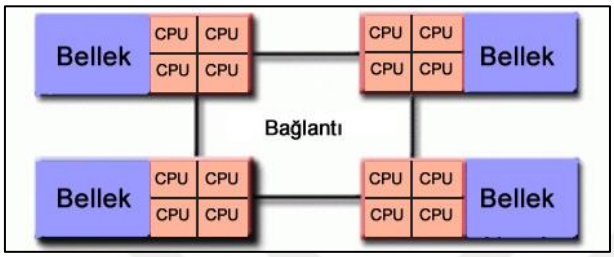
Tekdüze bellek erişimi olan sistemlerde belleğe erişim tüm işlemciler tarafından aynı şekilde ve erişim hızları eşittir (Bkz. Şekil 4.4). Birden fazla çekirdek içeren yani çok çekirdekli işlemciler aynı bellek alanını kullandıklarından bu tür arasına girer. Simetrik çoklu işlemcili bilgisayarlarda kullanılır.



Şekil 4.4. Tekdüze bellek erişimi mimarisi

4.1.2. Tekdüze olmayan bellek erişimi (NUMA-Non-Uniform Memory Access)

Fiziksel olarak birbirlerine bağlı iki ya da daha fazla simetrik çoklu işlemcili bilgisayardan oluşur. Bir simetrik çoklu işlemci direk olarak diğer simetrik çoklu işlemcilerin belleklerine ulaşabilir. Ancak bütün işlemciler bütün belleklere aynı erişim süresine sahip değildir (Bkz. Şekil 4.5).



Şekil 4.5. Tekdüze olmayan bellek erişimi mimarisi

Bellek erişimi için global adres uzayının kullanılması programlama açısından kullanıcı dostudur. İşlemciler arası veri paylaşımı hızlıdır. Fakat ortak belleği kullanan işlemci sayısı artırılırsa işlemci bellek arasındaki trafik artacağından darboğaz oluşturur. Bununla birlikte belleğin senkronizasyonu ve belleğe doğru şekilde erişilmesi programcının sorumluluğunda olduğundan bu tür sistemler için kod yazmak daha fazla zaman ve dikkat gerektirir. Kodlar üzerinde hata olasılığı artar.

Çok çekirdekli işlemcilerde kullanılan ortak bellek programlama modelleri şunlardır;

- Pthreads
- OpenMP
- TBB
- Cilk

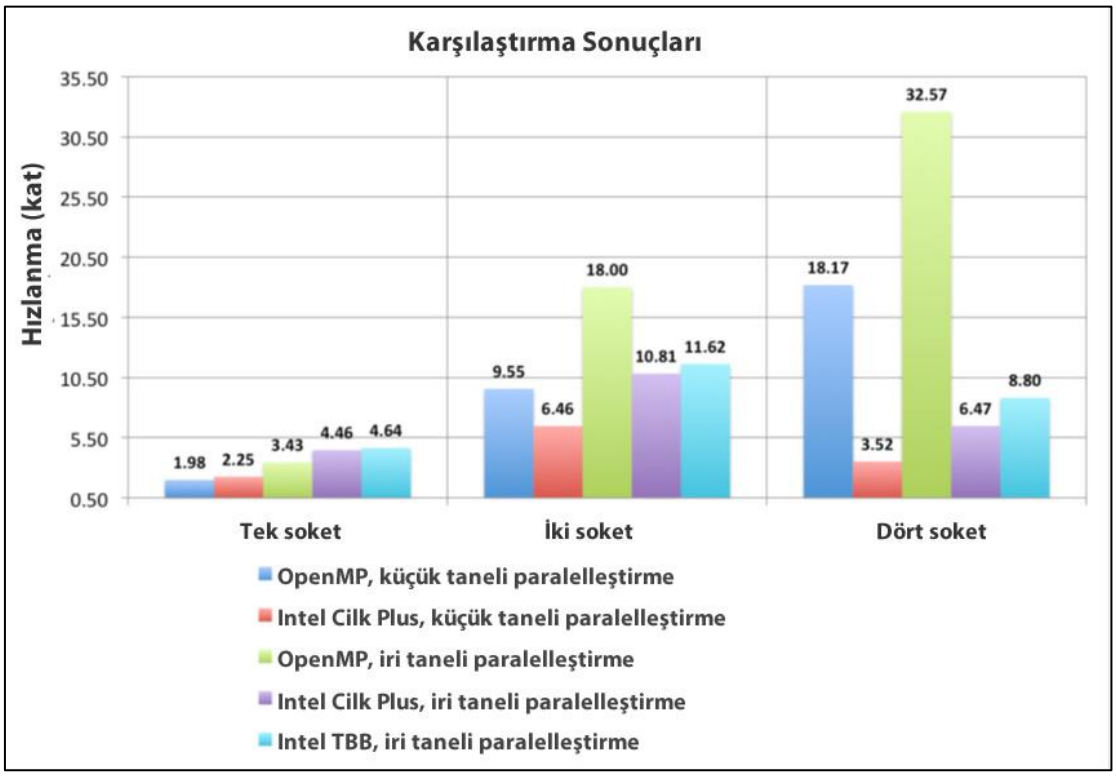
Pthreads Unix ve Linux işletim sistemlerinde çok çekirdekli işlemcilerin programlanmasında C dili ile kodlanabilmesi için genellikle UNIX türevi işletim sistemlerinde kullanılan programlama kütüphanesidir. Kütüphane POSIX olarak isimlendirilmektedir ve bu kütüphanenin baş harfi olan P harfi ile thread kelimesinin birleşmesinden türemiştir. OpenMP paylaşımlı bellekler üzerinde çoklu işlemeyi destekleyen bir arayüzdür. Pek çok işletim sistemi üzerinde çalışabilmektedir. TBB

(Threading Building Blocks) Intel'in C++ programlama dilinde kullanılabilen paralel programlama kütüphanesidir. Cilk 1994 yılında MIT laboratuvarları tarafından C programlama dili üzerinde çalışan çok çekirdekli işlemciler üzerinde çalışan genel amaçlı programlama modelidir.

4.1.3. Ortak bellek modellerinin karşılaştırılması

Çok çekirdekli işlemcilerin paralelleştirilmesinde kullanılacak sisteme göre bir programlama modeli tercih edilmelidir. Ortak bellek programlama modellerinin karşılaştırması CernOpenlab'ta uygulanarak 2012'de yayınlanmıştır. Ortak bellek programlama modelleri OpenMP, TBB ve Cilk karşılaştırması Şekil 4.6'da verilmektedir.

Küçük taneli (fine grained) paralelleştirmede genelde her döngüde paralelleştirme vardır. Çok sayıda döngü parallelleştirilir ve kodun çok iyi bilinmesine gerek yoktur. Program içerisinde pek çok senkronizasyon noktası vardır. İri taneli (coarse-grained) paralelleştirmede geniş döngülerle paralelleştirme yapılır. Daha az senkronizasyon noktası vardır. Kodun iyi anlaşılması gerekir.



Şekil 4.6. OpenMP, Cilk ve TBB karşılaştırması [5]

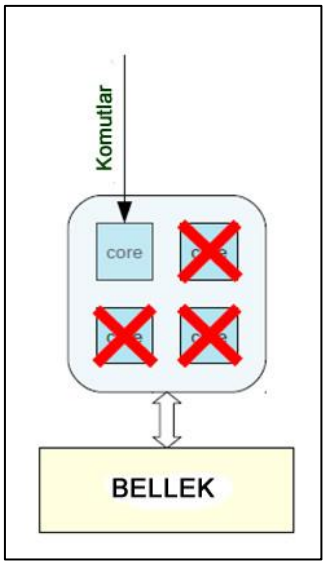
Şekil 4.6'ya göre OpenMP ile elde edilen hızlanmaların TBB ve Cilk'e göre daha yüksek olduğu görülmektedir. Aynı zamanda OpenMP platform bağımsız olduğuından her türlü donanım üzerinde çalışabilmektedir. Şekil 4.6'da elde edilen sonuçlar ve donanımdan bağımsız olması göz önünde bulundurularak bu tez çalışmasında işlemci çekirdeklerinin paralelleştirmesinde OpenMP kullanılmasına karar verilmiştir.

4.1.4. OpenMP

OpenMP Solaris, IBM AIX, HP-UX, GNU/Linux, Mac OS ve Windows işletim sistemleri üzerinde çalışan, Fortran, C++, C programlama dillerinde kullanılabilen paylaşımlı bellek üzerinde çoklu işlemeyi destekleyen bir arayüzdür. OpenMP kar amacı olmayan OpenMP Architecture Review Board (OpenMP ARB) isimli uluslararası bir birlik tarafından yönetilir. OpenMP ilk arayüzünü 1997 yılında Fortran için yayınlamıştır. Bir sonraki yıl ise C/C++ standardı yayınlanmıştır. OpenMP, 2.0 2002'de, 3.0 2008'de, 4.0 2013'te, 5.0 versiyonu ise 2018'de kullanıma sunulmuştur.

OpenMP, ortak bellek sistemleri için tasarlanmıştır ve genellikle OpenMP ile daha az çaba harcayarak paralelliği uygulamak için mevcut seri programlar kullanılır. Bu yüzden seri kodlar çok fazladır ve tam bir paralellik sağlanmaz. Diğer paralelleştirme platformlarıyla birlikte kullanılabilir.

Geleneksel bir seri program işlemci çekirdeklerinin bir tanesi üzerinde çalışmaktadır (Bkz. Şekil 4.7). Diğer çekirdeklerin işlemci gücü kullanılamamaktadır. Bu durum donanımın verimsiz olarak kullanılmasına neden olmaktadır.



Şekil 4.7. Seri programın çok çekirdekli mimaride çalışması

Çoklu çekirdekli mimarilerde tüm çekirdeklerin kullanılabilmesi amacıyla çok iş parçacıklı (multi-threading) programlama kullanılır. Bir işin gerçekleştirilmesi amacıyla yaratılan iş parçacıkları (thread) diğer çekirdeklerde çalışır ve aynı bellek alanını kullanırlar.

OpenMP'nin temel elemanları iş parçacığı oluşturma, iş yükü dağıtımı, veri ortam yönetimi, iş parçacığı senkronizasyonu, kullanıcı seviyeli çalışma zamanı rutinleri ve ortam değişkenleri için kullanılan komutlardır. C/C++'ta OpenMP kullanımı için *#pragma* etiketi kullanılır. OpenMP komutlarının çalıştırılması için C/C++'da *omp.h* kütüphanesinin eklenmesi gerekmektedir.

OpenMP kullanımına örnek olması bakımından işlemci sayısı kadar iş parçacığı yaratan ve her bir iş parçacığında kodu çalıştıran ve çalıştırılan iş parçacığı numarasını ekrana yazdıran programın C++ kodu verilmiştir;

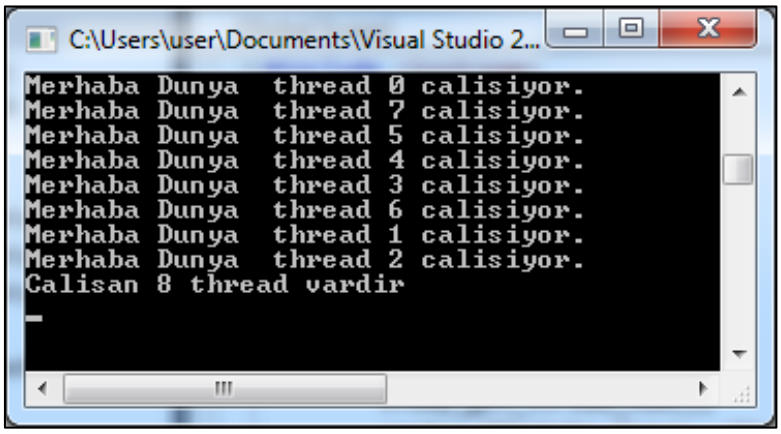
```
#include <iostream>
usingnamespace std;
#include<omp.h>

int main(int argc, char *argv[])
  {
        int th_id, nthreads;
```

```
        #pragma omp parallel private(th_id) shared(nthreads)
          {
                th_id = omp_get_thread_num();
               #pragma omp critical
                {
                   cout<<"Merhaba Dunya thread "<<th_id<<" calisiyor.\n";
                }
               #pragma omp barrier

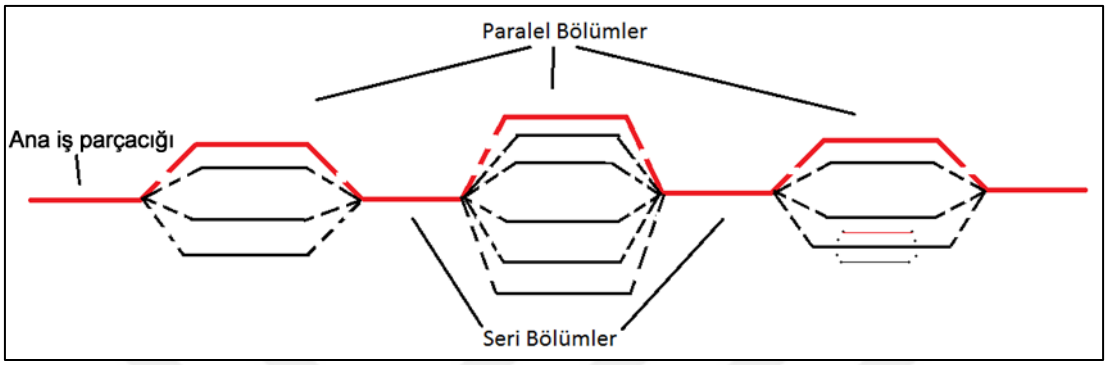
               #pragma omp master
                {
                    nthreads = omp_get_num_threads();
                    cout<<"Calisan "<<nthreads<<" thread vardir"<<'\n';
               }
          }
         getchar();
}
```

Yukarıda verilen program örneğinin çalıştırılması sonucu Şekil 4.8'deki gibi bir ekran görüntüsü elde edilir. Burada aynı kod tüm çekirdekler üzerinde çalıştırılmaktadır. Şekil 4.8'de de görüldüğü gibi çekirdekler üzerinde aynı anda çalıştırılan kodlarda iş parçacıkları sırasının karışık olduğu görülmektedir. Her çalıştırıldığında iş parçacıkları sırası farklı olacaktır.



Şekil 4.8.OpenMP örnek program çıktısı

OpenMP'de yürütme ana iş parçacığı (master thread) üzerinde gerçekleştirilir. Yavru iş parçacıları işlemlerin okunması sırasında ve kendilerine ihtiyaç duyulduğu yerde çalışmaya başlarlar (Bkz. Şekil 4.9). İş parçacıkları, program paralel alana girdiğinde ortaya çıkarlar. Paralel bölgeden çıkıldığında ise iş parçacıkları bırakılırlar.



Şekil 4.9. OpenMP iş parçacığı oluşumu

İş parçacıklarının işlemleri bitirme zamanı ya da aynı zamanda bitirecekleri garanti edilemediği için senkronizasyona ihtiyaç duyulan durumlarda *barrier* komutu kullanılmalıdır. İş parçacığı bilgilerinin elde edilmesi için kullanılan bazı komutlar şunlardır:

- omp_get_thread_num() komutu çalışan iş parçacığı numarasını verir.
- omp_get_num_procs() komutu çekirdek sayısını döndürür.
- omp_get_num_threads() komutu o anda çalışan iş parçacığı sayısını döndürür.
- omp_get_max_threads() komutu en fazla çalışabilecek iş parçacığı sayısını döndürür.

İş parçacıklarının dallanmasını kontrol için kullanılan komutlar ise *parallel*, *parallel for* ve *section* komutlarıdır.

parallel komutu

Parallel komutu içerisine yazılan kodlar tüm çekirdeklerde paralel olarak çalıştırılır. Tüm çekirdeklerde aynı anda çalıştırılır ancak her seferinde komutu bitirme zamanı farklı olabilmektedir.

Kullanımı aşağıdaki gibidir;

```
#pragma omp parallel
{
        // Bu bölüme yazılan kodlar paralel olarak çalıştırılır.
        Printf("Merhaba!\n");
}
```

Verilen kodda ekrana çekirdek sayısı kadar "Merhaba!" yazacaktır. Hangi işlemcinin önce, hangisinin sonra çalışacağı ise rasgeledir ve her çalıştırmada farklı olacaktır.

Parallel for komutu

Paralel for içerisine yazılan kod parçacığında işlem çekirdeklere paylaştırılır. Her komut bir kez işlenir. Komutlar birden fazla çekirdekte çalıştığından işlem zamanı da çekirdek sayısıyla orantılı olarak düşer. Ancak hiçbir zaman doğrusal bir orantı bulunmaz. Mutlaka iş parçacığı oluşturulması sırasında bir iş yükü oluşur.

Kullanımı iki farklı şekilde olabilir;

```
#pragma omp parallel for[clauses]
{//Buraya yazılan kodlar işlemciler arasında paylaşılır.
        For (int i=0; i<1000; i++) {
                a[i] = b[i] + c[i];
        }
}
```

ya da

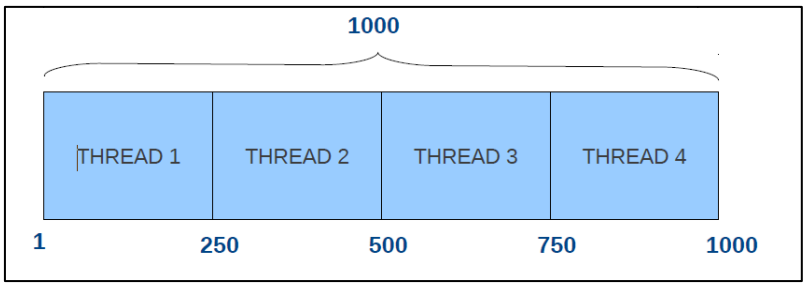
```
#pragma omp parallel [clauses]
{ //Buraya yazılan kodlar tüm işlemcilerde işlenir.
        #pragma omp for [clauses]
        { //Buraya yazılan kodlar işlemciler arasında paylaşılır.
                For (int i=0; i<1000; i++) {
                        a[i] = b[i] + c[i];
                }
        }
}
```

[clauses] kısmına kullanılabilecek parametreler gelmektedir. İlk kullanım şeklinde bazı parametreler kullanılamamaktadır. İkinci kullanımda ise tüm parametreler kullanılabilmektedir. Uygulamanın durumuna göre komut kullanımı belirlenmektedir.

Paralel for komutunun parametrelerinden biri *schedule*'dur. Bu komut ile iş parçacıklarının işlemciler arasında nasıl paylaştırılacağı belirlenir. *Schedule* 4 farklı parametre türü ile kullanılabilir. Bunlar *dynamic, static, guided* ve *runtime* parametre türleridir. Kullanımı *schedule(tür[, chunksize])* şeklindedir. Bu parametrelerin çalışma şekilleri aşağıdaki gibidir:

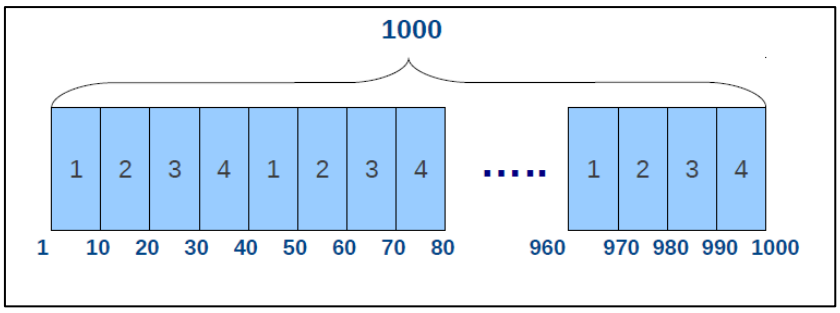
- ***static*** parametresi verilen veri bloğu büyüklüğündeki (chunksize) parçaları işlemciler üzerinde sıra ile dağıtır. Hangi parçaların hangi işlemcilerde çalışacağı baştan belirlenir.
- ***dynamic*** parametre türü kullanıldığında ise işlem veri bloğu büyüklüğü kadar parçalara ayrılır. İşlemcilerde çalışmaya başlanır. İşlemci işini bitirdiğinde sıradaki parçayı işler. Bu şekilde tüm işlem bitene kadar devam eder.
- ***guided*** parametresinde döngüler 1'e kadar sürekli azalan veri bloklarına (chunk) bölünür.
- ***runtime*** parametresinde uygulamanın çalıştırılması sırasında OMP_SCHEDULE ile hangi türün kullanılacağı program içerisinde belirlenebilir.

Örneğin;
1000 adet işlemin olduğu bir döngüde *schedule (static, 250)* seçildiğinde Şekil 4.10'daki gibi Thread 1'de 1'den 250'ye kadar olan bölüm, Thread 2'de, 251'den 500'e kadar olan bölüm, Thread 3'te 501'den 750'ye kadar olan bölüm, Thread 4'te ise 751'den 1000'e kadar olan bölüm işlenir.



Şekil 4.10. *Schedule (static, 250)* için iş parçacıklarının çalışması

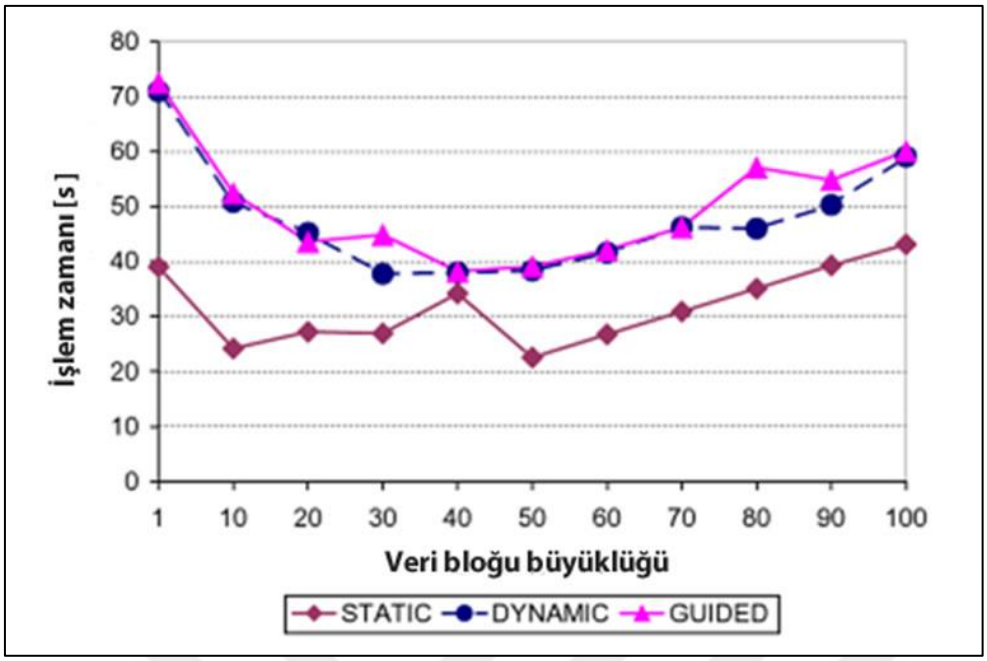
Aynı işlem için veri bloğu büyüklüğü 10 iken ise iş parçacıklarının hangi işlemcilerde çalışacağı Şekil 4.11'de verilmektedir.



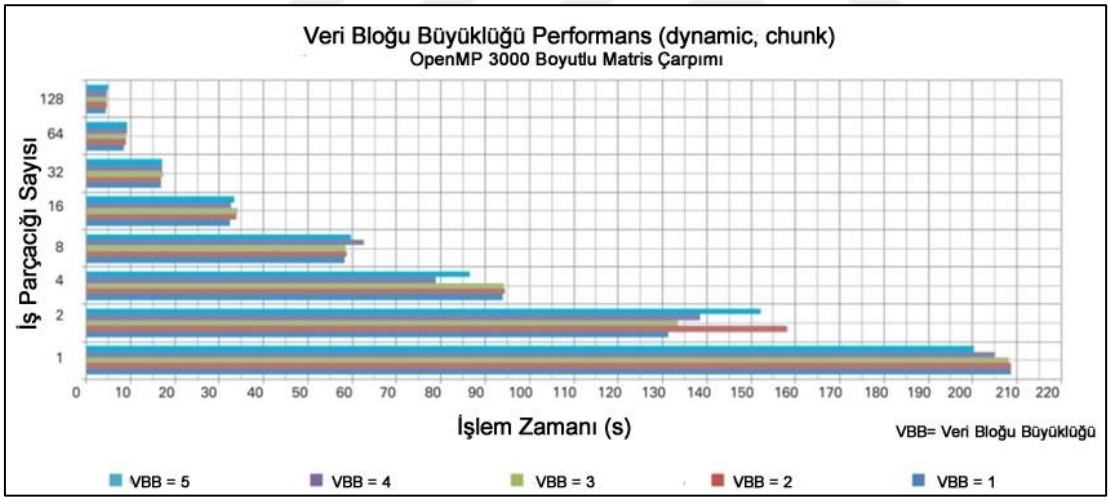
Şekil 4.11. *Schedule (static, 10)* için iş parçacıklarının çalışması

Dynamic olarak iş parçacıklarının işlenmesinde ise yine veri bloğu büyüklüğünde ayrılan gruplar işlemci numaralarına sabit olarak atanmazlar. Sıra ile boşalan işlemciye atanırlar.

Veri bloğu büyüklüğünün değişmesi ile işlem zamanı farklılık gösterir. Çalıştırılan uygulamaya göre veri bloğu büyüklüğünün etkisi farklı olabilir. Ya da işlem zamanı açısından en uygun veri bloğu büyüklüğü değeri uygulamaya göre farklılık gösterebilir. Bu konuda yapılan bir farklı iki çalışmanın sonucu Şekil 4.12 ve Şekil 4.13'te verilmektedir. Bu çalışmalara göre işlem zamanı ile veri bloğu büyüklüğü arasında doğrusal bir ilişki bulunmamaktadır. Buna göre en kısa işlem zamanı için en uygun veri bloğu büyüklüğü değeri uygulamaya göre değişmektedir.



Şekil 4.12. İşlem zamanı ve veri bloğu büyüklüğü arasındaki ilişki [50]

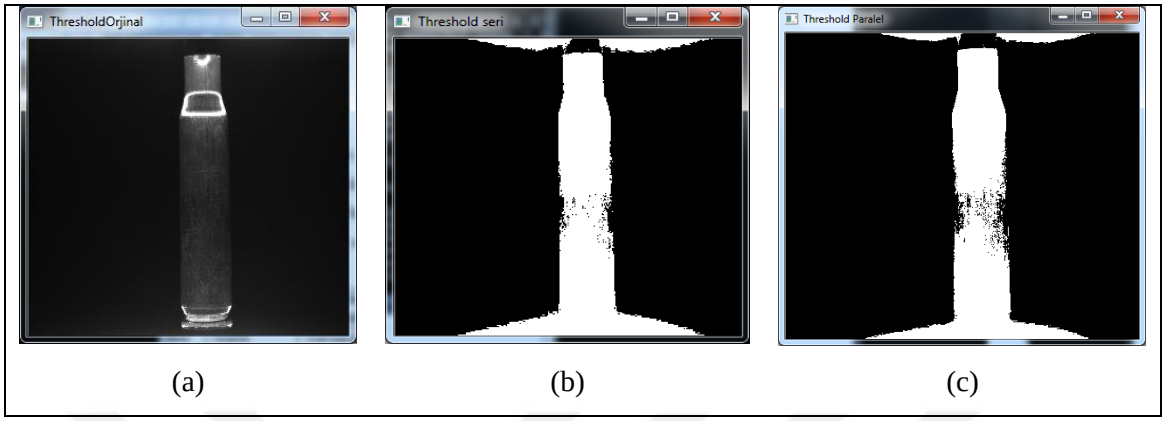


Şekil 4.13. İş parçacığı sayısı ve işlem zamanı arasındaki ilişki [51]

"Parallel for " için örnek bir uygulama

Veri bloğu büyüklüğünün hızlanmaya etkisini göstermek için yapılan örnek uygulamada gerçek kovan görüntüleri üzerinde eşikleme işlemi OpenMP kullanılarak paralelleştirilmiştir. Kovan görüntüleri üzerinde eşikleme işleme seri ve paralel olarak algoritmaları yazılmıştır. Yazılan kodlar gerçek kovan görüntüleri üzerinde çalıştırılmıştır.

Örnek kovan görüntüsü, seri ve paralel olarak gerçekleştirilen işlem sonucunda elde edilen görüntüler Şeki 4.14'teki gibidir.



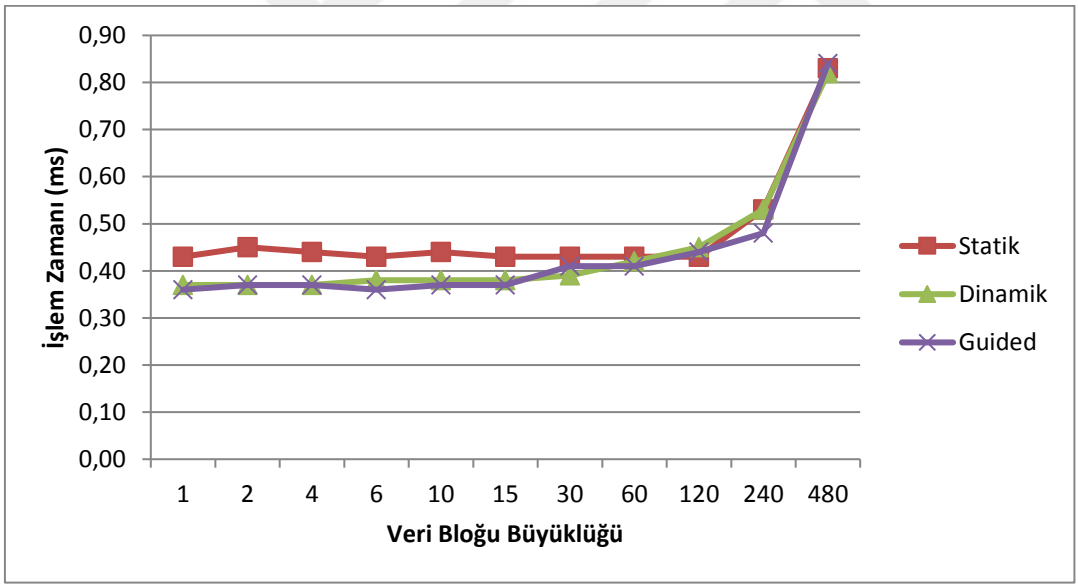
Şekil 4.14. Kovan görüntüleri (a) orjinal (b) seri eşiklenmiş (c) paralel eşiklenmiş

Kovan görüntüleri üzerinde OpenMP ile eşikleme işleminin seri ve paralel işlem süreleri karşılaştırılmıştır. Uygulama Intel i3630QM işlemci üzerinde çalıştırılmıştır. İşlemci fiziksel olarak 4 çekirdektir ancak *hyperthreading* özelliği ile 8 çekirdek gibi çalışabilmektedir. *Dynamic, static* ve *guided* türü zaman planlaması ile farkı veri bloğu büyüklükleri ile elde edilen sonuçlar Çizelge 4.1'de verilmektedir. Uygulama kameralardan alınan kovan görüntülerinin çözünürlüğü olan 1280X960 boyutlarındaki resimler üzerinde çalıştırılmıştır. Seri işlem süresi ile paralel işlem süresi farklı zaman türü planlaması ile farklı veri bloğu büyüklükleri için elde edilen süreler ve hızlanmalar verilmiştir.

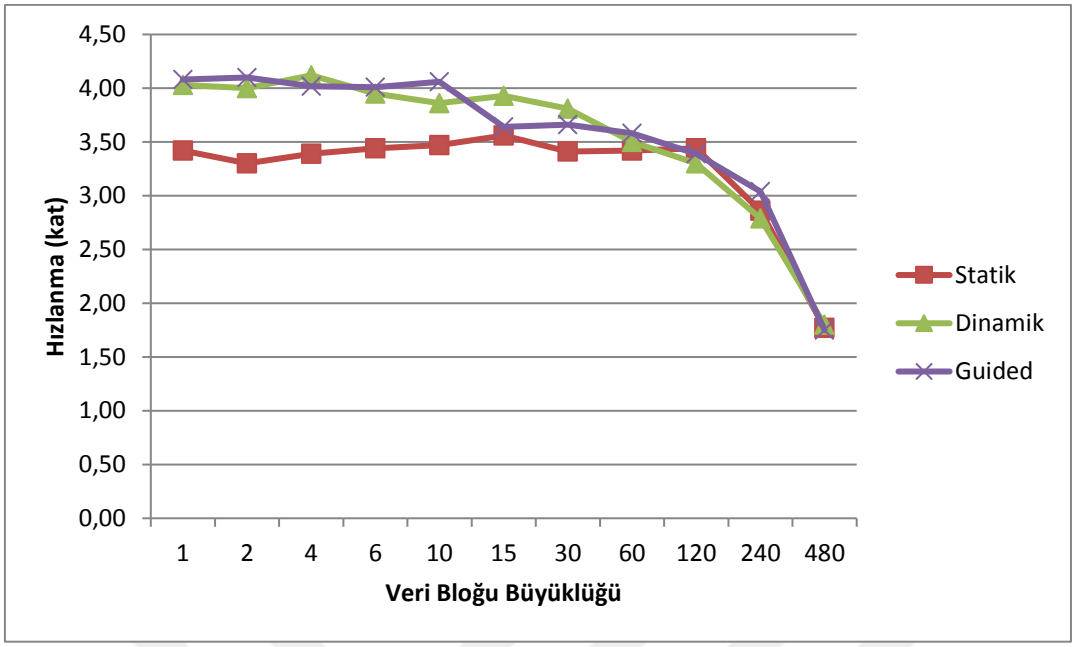
Çizelge 4.1. OpenMP ile elde edilen hızlanma

Planlama türü (schedule)	Veri bloğu büyüklüğü (chunksize)	Seri Süre (ms)	OpenMP ile süre (ms)	Hızlanma
Static	Varsayılan	1,49	0,43	3,43
Static	1	1,50	0,43	3,42
Static	2	1,49	0,45	3,30
Static	4	1,49	0,44	3,39
Static	6	1,49	0,43	3,44
Static	10	1,54	0,44	3,47
Static	15	1,53	0,43	3,56
Static	30	1,49	0,43	3,41
Static	60	1,49	0,43	3,42
Static	120	1,50	0,43	3,44
Static	240	1,50	0,53	2,86
Static	480	1,48	0,83	1,77
Dynamic	Varsayılan	1,49	0,36	4,11
Dynamic	1	1,50	0,37	4,03
Dynamic	2	1,49	0,37	4,00
Dynamic	4	1,52	0,37	4,12
Dynamic	6	1,51	0,38	3,95
Dynamic	10	1,49	0,38	3,86
Dynamic	15	1,49	0,38	3,93
Dynamic	30	1,49	0,39	3,81
Dynamic	60	1,49	0,42	3,50
Dynamic	120	1,49	0,45	3,30
Dynamic	240	1,48	0,53	2,79
Dynamic	480	1,49	0,82	1,80
Guided	Varsayılan	1,48	0,36	4,07
Guided	1	1,49	0,36	4,08
Guided	2	1,52	0,37	4,10
Guided	4	1,50	0,37	4,02
Guided	6	1,49	0,36	4,01
Guided	10	1,52	0,37	4,06
Guided	15	1,49	0,37	3,94
Guided	30	1,50	0,41	3,66
Guided	60	1,49	0,41	3,58
Guided	120	1,50	0,44	3,39
Guided	240	1,48	0,48	3,04
Guided	480	1,48	0,84	1,75

Çizelge 4.1'e göre elde edilen paralel işlem süresi ve veri bloğu büyüklüğü grafik olarak Şekil 4.15'te verilmektedir. Grafiğe göre belli bir veri bloğu büyüklüğüne kadar işlem zamanı sabit kalmakta, veri bloğu belli bir büyüklüğü geçtiğinde işlem süresi artmaktadır. Ayrıca Şekil 4.16'da seri işlem süresine göre elde edilen hızlanma ile veri bloğu büyüklüğü arasındaki ilişki grafik olarak verilmiştir. *Dynamic* ve *guided* zaman planlaması türü birbirine benzerlik göstermektedir. Veri bloğu büyüklüğü arttıkça hızlanmada azalma görülmektedir. Veri bloğu büyüklüğü 1-10 arasında birbirine yakın hızlanmalar elde edilmişken veri bloğu büyüklüğü arttıkça hızlanmanın azaldığı görülmektedir. Veri bloğu büyüklüğü 120'yi geçtiğinde hızlanmada önemli bir azalma olduğu ve veri bloğu büyüklüğü arttıkça hızlanmada azalmanın arttığı gözlemlenmektedir. Bu nedenle tüm zaman planlaması türlerinde probleme göre uygun en iyi hızlanmanın olduğu veri bloğu büyüklüğünün bulunarak uygulamada kullanılması gerekmektedir. *Static* ile yaklaşık 3,5 kat hızlanma elde edilmişken *dynamic* ve *guided* ile 4 kat hızlanma elde edilmiştir.



Şekil 4.15. İşlem zamanı ile veri bloğu büyüklüğü arasındaki ilişki



Şekil 4.16. Hızlanma ile veri bloğu büyüklüğü arasındaki ilişki

section komutu

Section komutu ile farklı işlemlerin farklı çekirdekler tarafından işlenmesi sağlanır. Bir *section* içerisindeki tüm işlemler bir çekirdek tarafından yapılır. Diğer *section* içerisindeki ise farklı bir çekirdek tarafından işlenir. *Section* komutu ile en iyi hızlanmayı sağlamak için her bir bölüm içine işlemci yükü birbirine mümkün olduğunca yakın olmalıdır. Her bir bölüm birbirinden bağımsız olarak işlenir.

Kullanımı iki farklı şekilde olabilir;

```
#pragma omp parallel sections
{
        { Work1();}
        #pragma omp section
                { Work2();
                Work3();}
        #pragma omp section
        { Work4();}
}
```

ya da

```
#pragma omp parallel
{
//Work0(); // Bu fonksiyon tüm iş parçacıkları tarafından çalıştırılır.

   #pragma omp sections// İşler bölümlere ayrılır
   {
        // Buradaki her bir bölüm bir işlemci tarafından işlenir.
                { Work1();}
        #pragma omp section
                { Work2();
                Work3();}
        #pragma omp section
                { Work4();}
    }
    Work5(); // Bu fonksiyon tüm iş parçacıkları tarafından çalıştırılır.
}
```

Kovan görüntüleri üzerinde çalıştırılan paralel eşikleme uygulaması *Section* komutu ile ortalama süre 0,35 ms olarak ölçülmüş ve hızlanma 4,20 kat olarak gerçekleşmiştir. *Parallel for* komutunda 0,37 ms işlem süresi ile en iyi hızlanma 4,12 kat olarak gerçekleşmiştir. Section komutu kullanılarak daha iyi hızlanma elde edildiği görülmüştür.

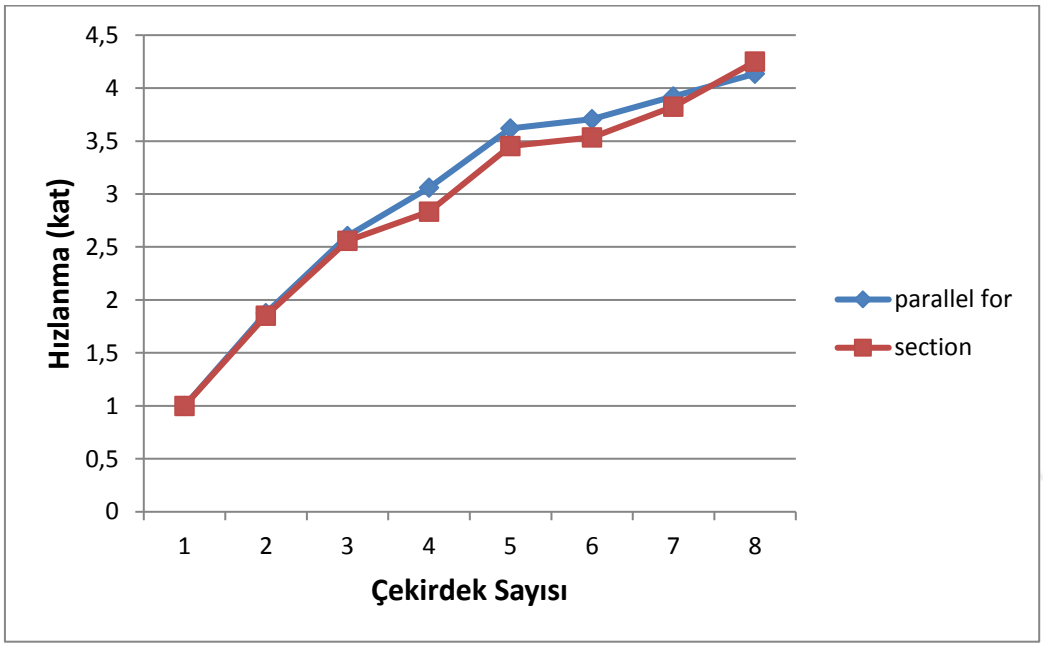
Çekirdek sayısı ve hızlanma

Çok çekirdekli işlemcilerde ideal hızlanma çekirdek sayısı kadardır. Olabilecek en yüksek hızlanma ideal hızlanmadır. Ancak pratikte bu orana ulaşmak mümkün değildir.

Eşikleme örneği için farklı çekirdek sayılarına göre elde edilen hızlanmaların sonuçları Çizelge 4.2'de verilmektedir. Elde edilen sonuçlara göre çekirdek sayısı arttıkça hem *parallel for* hem de *section* komutu için hızlanmanın arttığı görülmektedir. Elde edilen hızlanmaların çekirdek sayısına göre grafiği Şekil 4.17'de verilmektedir.

Çizelge 4.2. OpenMP ile çekirdek sayısına göre elde edilen hızlanma

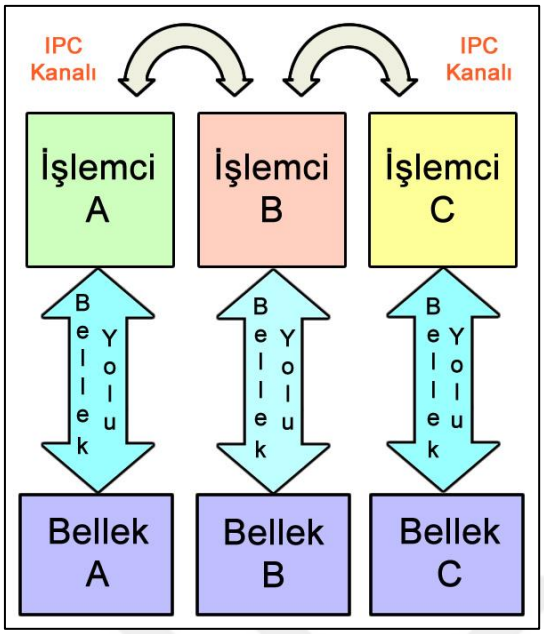
Çekirdek sayısı	Seri süre (ms)	Paralel süre (ms)		Hızlanma	
		parallel for	section	parallel for	section
2	1,5	0,8	0,81	1,88	1,85
3	1,51	0,58	0,59	2,60	2,56
4	1,53	0,5	0,54	3,06	2,83
5	1,52	0,42	0,44	3,62	3,45
6	1,52	0,41	0,43	3,71	3,53
7	1,53	0,39	0,4	3,92	3,83
8	1,53	0,37	0,36	4,14	4,25



Şekil 4.17. OpenMP'de çekirdek sayısı ile hızlanma arasındaki ilişki

4.2. Dağıtık Bellekli Sistemler

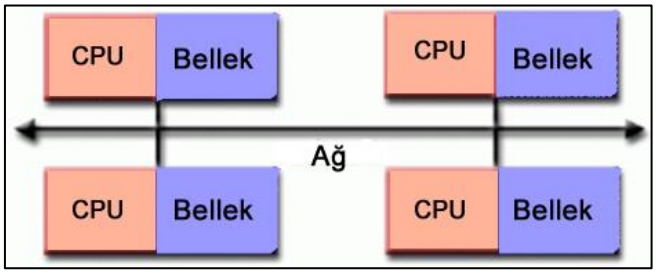
Yüksek performanslı sistemler işlemci saat frekansının arttırılması veya donanımın paralel kullanılmasıyla elde edilebilir. İşlemci saat frekansının arttırılması ekonomik nedenler veya fiziksel sınırlar nedeniyle mümkün olamamaktadır. Bu nedenle donanımım paralel kullanılmasıyla elde edilen sistemler kullanılmaktadır [51]. Dağıtık bellekli sistemlerde her işlemciye ait farklı bir bellek bulunmaktadır. Aralarındaki iletişim veri yollarıyla sağlanmaktadır (Bkz. Şekil 4.18). Bu tür sistemlere en güzel örnek küme bilgisayar sistemleridir. Süreçler arası iletişim (IPC) kanalı iş parçacıkları arasında veri transferine izin vermektedir. Bu iletişimde farklı teknikler kullanılabilir.



Şekil 4.18. Dağıtık bellekli bilgisayar mimarisi şematik gösterimi

Küme bilgisayar sistemleri birbirinden bağımsız bilgisayarların iletişim ağlarıyla birbirlerine bağlanmasıyla oluşan sistemlerdir. Bu sistemler düşük maliyetlerle yüksek işlemci gücüne ulaşmaktadırlar. Tüm dünyada yaygın olarak kullanılmaktadır. Artık süper bilgisayarların neredeyse tümü küme bilgisayarlardan oluşmaktadır.

Dağıtık bellekli sistemlerde bellekler arası bir iletişim ağı vardır. İşlemcilerin kendi bellekleri vardır ve bu belleği kullanırlar (Bkz. Şekil 4.19).



Şekil 4.19. Dağıtık bellek yapısı

Bir işlemci diğer bellek adreslerini bilmez bu yüzden bellek adresleri için global adres uzayı kavramı yoktur. Her işlemci kendi belleğini organize eder ve yaptığı değişiklikler diğer işlemcilerin belleklerini etkilemez.

İşlemci sayısı artarsa orantılı olarak bellek de artar. Her işlemci kendi belleğine hemen erişebilir ağ iletişimi ile uğraşmaz. İşlemciler arası veri iletişiminin detaylarından programcı sorumludur. Bellek erişim süreleri farklılık gösterebilir.

Dağıtık bellekli sistemlerde en yaygın kullanılan modeller Paralel Sanal Makine (PVM) ve Mesaj Geçiş Arayüzü (MPI)'dür.

4.2.1. PVM

PVM (Parallel Virtual Machine), farklı platformlarda ağ bağlantılı bilgisayarlar arasında dağıtılmış işlem yapmayı sağlayan açık kaynak kodlu bir platformdur. Yazılım, çoğunlukla Unix olmak üzere Unix tabanlı veya Windows sisteminde çalıştırılabilir. PVM, 1989 yılında Oak Ridge Ulusal Laboratuvarı'nda temeli atılmış ve 1991'de Tennessee Üniversitesi'nde yeniden yazılmış bir platformdur.

PVM, heterojen dağıtılmış bilgi işlem için bir yazılım ortamıdır. Bir kullanıcının, dağıtılmış işlemcilerden oluşan bir paralel hesaplama sistemini oluşturmasına, erişmesine ve sonuçta ortaya çıkan sistemin tek bir sanal makine olarak ele alınmasını sağlar.

Bir kullanıcının sanal makinesindeki donanım, tek işlemci iş istasyonları, vektör makineleri, paralel süper bilgisayarlar veya bunların herhangi bir kombinasyonu olabilir. Tek tek elemanlar, kullanılan tüm makineler bir veya daha fazla ağ üzerinden bağlandığı sürece PVM tarafından çalıştırılabilir. Bir merkezi kontrol altında farklı kaynakları bir araya getirme becerisi, PVM kullanıcısının bir problemi alt görevlere bölmesine ve her birinin bu alt-görev için en uygun işlemci mimarisi üzerinde çalıştırılmasına izin verir.

PVM, paralel programlamanın mesaj gönderme modeline dayanmaktadır. Mesajlar, bağlantı ağları üzerindeki görevler arasında iletilir. Kullanıcının görevleri, diğer görevleri başlatabilir ve sonlandırabilir, veri gönderip alabilir ve bir mesaj aktarım rutini kütüphanesi kullanarak birbiriyle senkronize olabilir. Görevler dinamiktir; yani, bir programın çalışması sırasında başlatılabilir veya sonlandırılabilir.

4.2.2. MPI

MPI (Message Passing Interface) bir bilgisayar iletişim protokolüdür. Dağıtık bellekli bir sistemde paralel program koşan düğümlerin arasındaki iletişim için kullanılan standart bir protokoldür. MPI uygulamaları Fortran, C, C++ ve Ada programlarından çağrılan kütüphane yordamlarından oluşur. MPI'ın üstünlüğü taşınabilir ve hızlı olmasıdır.

İlk MPI çalışmalarına Kasım 1992'de başlanmıştır. MPI standardı yaklaşık elli kişilik bir grup tarafından birçok farklı organizasyonda tartışılarak geliştirilmiştir. 1993 yılında süper bilgisayarlar için geliştirilen ilk programların taslakları sunulmuş ve bir yıl sonra ilk MPI standardı olan MPI 1.0 bir çok farklı platformda denenerek tamamlanmıştır. Bu standart 2005 yılında sunulmuş olup Fortran 77 ve C'de çağrılacak alt rutin ve fonksiyon isimlerini ve sonuçlarını içermektedir. MPI 2.0 versiyonu 2016 yılında yayınlanmıştır. 3.0 ise 2017'de kullanıma sunulmuştur. Son sürümü olan 4.0 ise 2018'de yayınlanmıştır.

MPI'ın hedefi dağıtık programlar yazmak için geniş kullanıma sahip bir standart sağlamaktır. Arayüz; dağıtıklık için pratik, taşınabilir, etkin ve esnek bir standart kurmaya çalışır.

MPI kütüphanesi farklı gruplara ayrılmış birçok komutu içermektedir. Bu gruplar;

- Başlatma/Sonlandırma Fonksiyonları
- Noktadan Noktaya Haberleşme Fonksiyonları
- Toplu Haberleşme Fonksiyonları
- Haberleşme Topolojisi Fonksiyonları
- Kullanıcı Veri Tipi Tanımlama Fonksiyonları
- Araçlar (Zaman ölçme gibi)'dır.

Bir MPI programında C++ dosyasına mpi.h kütüphanesi eklenmelidir. MPI kütüphanesi eklendikten sonra mpi komutları çalıştırılabilir. Bir MPI programının genel yapısı aşağıdaki gibidir;

```
#include<mpi.h> // mpi kütüphesi tanıtma
.....
...
MPI_INIT     // MPI komutları bu komuttan sonra yazılır.
...
// MPI komutları buraya yazılır.
...
MPI_Finalize  /MPI komutları bu komutla sonlandırılır.
```

MPI arayüzü komutlarını kullanan C programlama dilinde yazılmış örnek uygulama aşağıdaki gibidir;

```
#include<mpi.h>
#include<stdio.h>
#include<string.h>

int main(int argc, char *argv[])
{
   char idstr[32];
   char tamp[128];
   int islemcino;
   int benimid;
   int i;
   MPI_Status durum;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&islemcino);
   MPI_Comm_rank(MPI_COMM_WORLD,&benimid);

   if(benimid == 0)
   {
        printf(" %d tane islemcim var\n", islemcino);
        for(i=1;i<islemcino;i++)
        {
           sprintf(tamp, "Merhaba %d! ", i);
           MPI_Send(tamp, 128, MPI_CHAR, i, 0, MPI_COMM_WORLD);
        }
        for(i=1;i<islemcino;i++)
```

```
        {
           MPI_Recv(tamp, 128, MPI_CHAR, i, 0, MPI_COMM_WORLD, &durum);
           printf("%s\n", tamp);
        }
   }
   else
   {
        MPI_Recv(tamp, 128, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &durum);
        sprintf(idstr, "Islemci %d ", benimid);
        strcat(tamp, idstr);
        strcat(tamp, "gorevehazir\n");
        MPI_Send(tamp, 128, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
   }

   MPI_Finalize();
   return 0;
}
```

Temel MPI fonksiyonlarından en çok kullanılanlar aşağıdaki gibidir;

MPI_Init: MPI çevresini hazırlar. Her MPI programında mutlaka bulunur. Herhangi bir MPI fonksiyonundan önce çağrılmalıdır. MPI programı içinde sadece bir kere çağrılır.
Örnek kullanımı:
MPI_Init(int *argc, char **argv[])

MPI_Finalize : MPI çalışma çevresini sonlandırır. Bu fonksiyondan sonra çağrılan MPI fonksiyonları geçersizdir.
Örnek kullanımı:
MPI_Finalize(void)

MPI_Comm_size: Haberleşme dünyası ile ilgili grubun kaç sürece sahip olduğunu belirler. Genellikle MPI_COMM_WORLD haberleşme ortamı uygulamanın kullanacağı süreç sayısını belirtir.
Örnek kullanım:
int MPI_Comm_size(MPI_COMM_WORLD, int *size)

MPI_Comm_rank: Çağrıldığı sürecin haberleşme ortamındaki sıra numarasını belirtir. MPI_COMM_WORLD haberleşme ortamında sıra numarası 0 ile (Süreç sayısı - 1) arasında tekil tamsayıdır.
Örnek kullanım:
int MPI_Comm_rank(MPI_COMM_WORLD, int *rank)

MPI_Send: İstenilen veriyi istenilen noktaya bloklu şekilde gönderir. Basit bloklu gönderme işlemidir. Fonksiyon gönderici sürecinin uygulama tamponu tekrar kullanmaya uygun olduğunda sonlanmaktadır.

MPI_Recv: Belirli bir veriyi istenilen kaynaktan bloklu bir şekilde alır. Fonksiyon alıcı süreci bir mesaj alana ve uygulama tamponunda istenilen veri olana kadar sonlanmayacaktır.

5. GPU PROGRAMLAMA VE CUDA

GPU (Graphical Processing Unit) yoğun matematiksel hesaplamaları yapmak üzere tasarlanmış ekran kartlarının işlemcisi için kullanılan bir isimdir. GPU'lar, CPU'lardan daha verimli ve performanslı bir biçimde ekrana görüntü verirler. Grafikleri işleme ve göstermekte son derece verimlidir ve paralelleştirmeye uygun yapıları sayesinde karmaşık algoritmalar için CPU'dan daha etkin şekilde çalışırlar. GPU ekran kartının hemen üstünde olabileceği gibi ana karta entegre halde de olabilir. GPU, ev kullanıcıları için genellikle oyun amaçlı kullanılırken profesyonel kullanıcılar için farklı uygulamalar yapılabilmektedir.

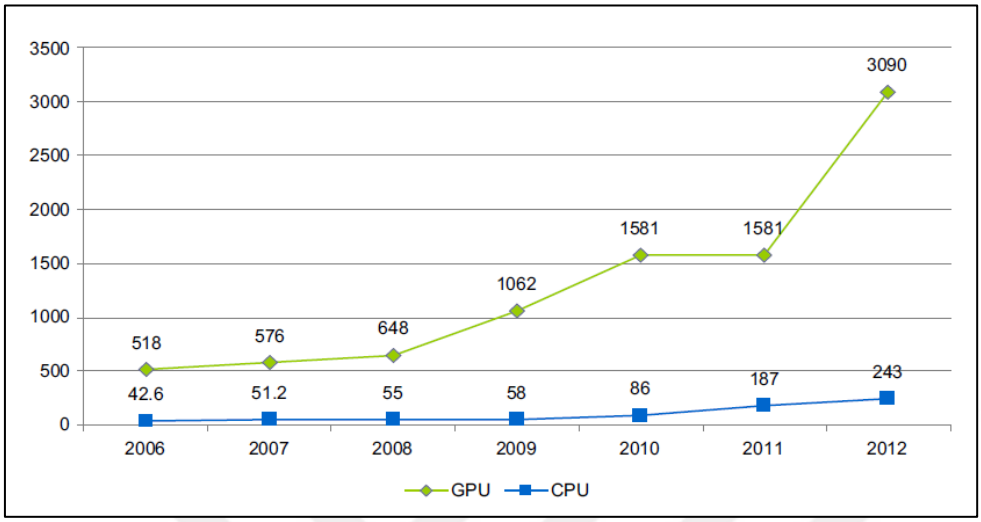
CPU'lar seri hesaplamalarda, giriş çıkış işlemlerinde iyi bir performansa sahiptirler. GPU'lar ise büyük ölçekli paralel hesaplamalar için uygun bir mimariye sahiptirler.

5.1. Genel Amaçlı GPU Programlama Tarihi

Ekran kartları GPU işleme veya GPGPU (genel amaçlı GPU programlama) ismiyle genel amaçlı programlamada yaklaşık 10 yıldır kullanılmaktadır. İlk GPU programı daha çok bir hesap makinesine benziyordu: İlk GPU programları işlem sürelerini kısaltmak için basit grafik işlemlere uygulanmaktaydı. Bu programları geliştirmek, hata ayıklamak ve optimize etmek zordu ve derleyici hatalarına sıklıkla rastlanıyordu. Buna rağmen GPU kullanılan programlarda hızlanma CPU ile karşılaştırıldığında oldukça yüksek olmaktaydı. Bu durum da bu konuda yapılan çalışmaların artmasına neden olmuştur [53].

CPU ile karşılaştırıldığında GPU hesaplamada başarının anahtarı yüksek performans olmaktadır. Günümüzde teorik olarak verilen en yüksek bant genişliği ve kayan noktalı hesaplama (gigaflops) performansına bakıldığında GPU'lar yedi kez artarken CPU'lar iki kez artmaktadır (Bkz. Şekil 5.1). Bu performans farkı iki işlemci arasındaki fiziksel çekirdek sınırlamaları ve mimari farkından kaynaklanmaktadır. CPU seri olarak çalışır ve seri operasyonların işlem hızını optimize etmeye çalışır. CPU'da geleneksel olarak performans artışı frekansın yükseltilmesiyle elde edilir. Frekans iki katına çıkarsa hız da iki katına çıkar ve uzun yıllar performans artışı bu şekilde sağlanmıştır. 2000'li yıllardan sonra fiziksel sınırlardan dolayı frekans artışı istenildiği kadar gerçekleştirilememiştir. Frekans artışı ısınma nedeniyle 4,0 GHz civarında durdurulmak zorunda kalmıştır. Seri hesaplamada şu

andaki teknolojilerle en yüksek performansa ulaşılmış ve onun yerine performans artışı çoklu çekirdek ve vektör komutları ile sağlanmaya başlamıştır [53].



Şekil 5.1. CPU, GPU bant genişliğinin yıllara göre karşılaştırması [54]

Bununla aynı zamanlarda GPU'ların büyük ölçüde paralellik sayesinde performansı üstel (exponentially) olarak artmaya devam etmiştir. Ekran üzerindeki renkli bir pikselin işlenmesi tüm diğer piksellerden bağımsızdır ve doğal bir paralellik içermektedir. Bu da paralel uygulamalarda performans artışını beraberinde getirmektedir. Bu nedenle GPU'lar büyük ölçekli paralel uygulamalarda performansı arttırmak için çok kullanışlıdırlar. Kodun paralel bölümünün seri kısmına göre artması performansın da artmasını sağlar.

Amdahl Kuralına göre bu tercih edilen bir durumdur. Amdahl Kuralı uygulamanın teorik olarak en fazla ne kadar hızlanabileceğini tanımlayan ve paralel programlama için önemli kanunlardan biridir. Amdahl bir programın belli bir yüzdesinin paralelleşemediğini varsayar. Geri kalanında ise mükemmel paralellik olduğunu ve diğer tüm gecikmeleri yok sayarak kodun idealde ne kadar hızlanacağını hesaplar.

5.2. GPU Üreticileri

PC pazarında üç büyük GPU üreticisi vardır. Bunların en büyüğü ise Intel'dir. Fakat Intel sadece düşük performans pazarında baskındır. Yüksek performans pazarını ise AMD ve NVIDIA paylaşıyorlar. Akademik ve endüstriyel uygulamalarda ise NVIDIA, AMD'ye göre daha yaygın olarak kullanılmaktadır.

5.3. GPU Programlama

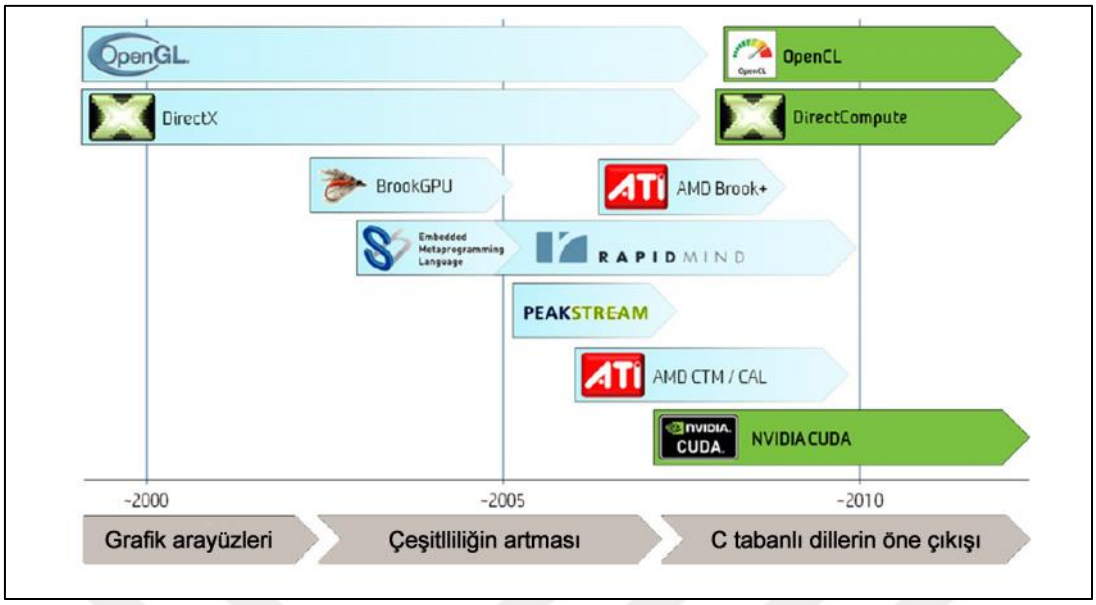
Genel amaçlı GPU programlamada üç platform kullanılmaktadır. Bunlar;

- CUDA
- OpenCL
- DirectCompute'dir

CUDA (Compute Unified Device Architecture) NVIDIA firması tarafından 2006 yılında sunulan GPU'larda genel amaçlı programlama yapılmasını sağlayan bir platformdur. Linux, Windows ve Mac OSX üzerinde çalışabilen hem düşük hem de yüksek seviyeli bir yazılım geliştirme arayüzü sunmaktadır.

OpenCL (Open Computing Language), Apple tarafından 2008 yılında kar amacı gütmeyen teknoloji şirketleri birliği KhronosGroup'a önerilen, kabul gördükten sonra pek çok şirketin katkılarıyla hazırlanan heterojen hesaplama platformudur. OpenCL destekli grafik işlemcileri, genel amaçlı işlemciler ve alan programlanabilir kapı dizileri (FPGA) gibi farklı platformlarda hesaplama yapılmasına olanak sağlar. OpenCL platformu, AMD, Intel, NVIDIA ve ARM gibi farklı üreticiler tarafından desteklenmektedir.

Direct Compute Windows Vista ve Windows'un sonraki sürümleri tarafından desteklenen, ekran kartını genel amaçlı programlamayı sağlayan bir arayüzdür. Microsoft DirectX 10 ve DirectX 11 tarafından desteklenmektedir.

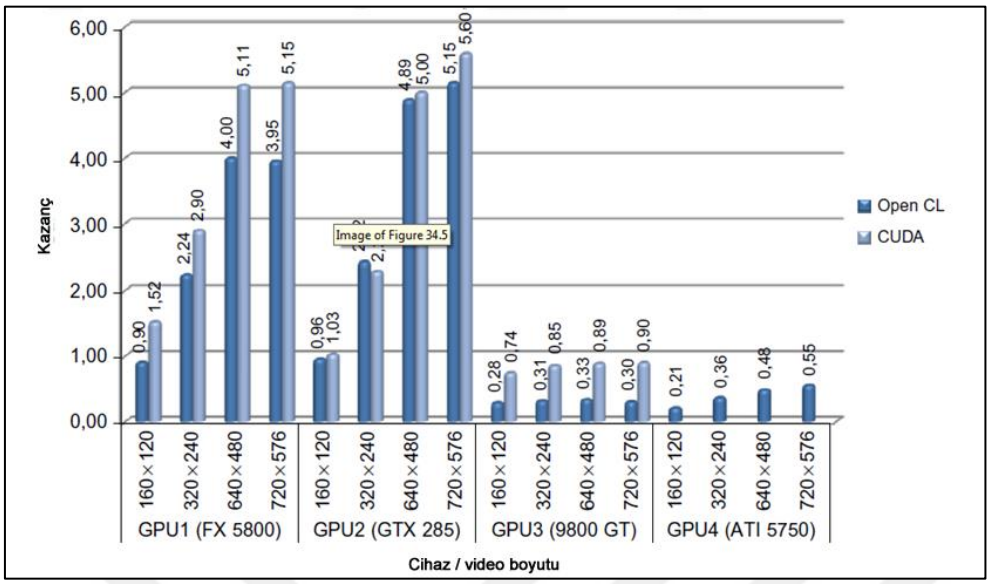


Şekil 5.2. GPU programlama gelişimi [53]

Genel amaçlı GPU programlama için kullanılan dillerin yıllara göre gelişimi Şekil 5.2'de verilmektedir. 2000 yıllarında OpenGL ve DirectX grafik arayüzleri kullanılmaktayken ilerleyen yıllardan pek çok yazılımın ortaya çıktığını görmekteyiz. Ancak 2010 yıllarından C tabanlı üç dilin öne çıktığı görülmektedir. CUDA genel amaçlı programlamada kullanılan üç dilden en gelişmiş araçlara sahip olanıdır [53]. Donanıma bağımlı olmasına rağmen en yaygın olarak kullanılan GPU programlama dili CUDA'dır.

CUDA NVIDIA tarafından yayınlanmış ve sadece NVIDIA tarafından üretilen ekran kartlarında çalışan bir platformdur. Bu nedenle donanım bağımlıdır. OpenCL ise Intel, AMD ve NVIDIA başta olmak üzere pek çok ekran kartı üreticisi tarafından desteklenmektedir. Ancak performans açısından CUDA, OpenCL ile karşılaştırıldığında pek çok uygulamada daha hızlı çalışmaktadır [54]. Paralel işlemcilerin hedefinin de yüksek hız olduğu düşünüldüğünde CUDA platformu pek çok kullanıcı tarafından tercih edilmektedir.

Örneğin, Temizel ve diğerleri [7] yaptıkları çalışmada CUDA ve OpenCL platformunu karşılaştırmışlar ve Şekil 5.3'te verilen sonuçları elde etmişlerdir. Yaptıkları çalışmada tüm ekran kartlarında CUDA platformunda elde edilen hızlanmalar OpenCL'den yüksek olmuştur. GPU ile elde edilen 12 adet testin 11 tanesinde CUDA daha iyi sonuçlar vermişken sadece bir tanesinde OpenCL'in daha performanslı olduğu görülmektedir. Bu sebeple bu tez çalışmasında CUDA platformu kullanılmıştır.



Şekil 5.3. OpenCL ve CUDA karşılaştırması [7]

5.4. GPU ve CPU Mimarisi Arasındaki Farklar

Bilgisayarda "*işlem*" denildiğinde akla ilk gelen bileşen CPU'dur. Fakat iş paralel programlamaya geldiğinde GPU'lar CUDA ile kendisini daha fazla öne çıkarmaktadır. Bir CPU çekirdeği, tek bir yönerge akışını bir kerede mümkün olduğu kadar çabuk yürütmek için tasarlanmıştır. GPU'lar ise birçok paralel yönerge akışını mümkün olduğu kadar çabuk yürütmek için tasarlanmışlardır.

5.4.1. Transistör kullanımı

CPU'larda transistörler yönergelerin yeniden düzenlenmesini gerçekleştiren tamponlar, rezerve istasyonları, tahmin mekanizmaları ve büyük tampon bellek gibi bileşenlerden oluşur. Bu bileşenler, tek bir iş parçacığının yürütülmesini hızlandırmak için tasarlanmıştır. GPU'da ise transistörler işlemci dizileri, çoklu işlem donanımı, paylaşılan bellek ve birden fazla bellek denetleyicilere atanmıştır. Bu özellikler belirli bir iş parçacığının işletilmesinin hızlandırılmasına değil, yonganın üzerinde bulunan on binlerce iş parçacığının aynı anda yürütülmesi için tasarlanmıştır ve böylece iş parçacıkları arasında iletişimi kolaylaştırarak yüksek bellek bant genişliği sağlar [55].

5.4.2. Ön belleğin (Cache) rolü

CPU, ön belleğini, genel bellek erişim gecikmesini azaltarak performans arttırmak amacıyla kullanır. GPU ise ön belleği (ya da yazılımla yönetilen paylaşılan belleği) bant genişliğini arttırmak için kullanır.

5.4.3. Gecikme (Latency) yönetimi

CPU'da bellek gecikmesi, geniş arabellekler ve tahmin mekanizmaları ile yönetilir. Bunlar tasarım üzerinde büyük yer kaplarlar ve genellikle güç tüketimi konusunda savrukturlar. GPU ise gecikmeyi binlerce iş parçacığını bir anda destekleyerek yönetir. Eğer herhangi bir iş parçacığı bellekten yük bekliyorsa, GPU herhangi bir gecikmeye sebep olmadan derhal başka bir işe geçiş yapar.

5.4.4. Çoklu iş parçacığı (Multi-threading)

CPU'lar çekirdek başına bir veya iki iş parçacığı yürütme desteğine sahiptir. CUDA destekli GPU'lar ise çoklu işlemci başına 1024'e kadar iş parçacığı destekler. CPU'nun uygulamalar arasında bir kez geçiş yapması yüzlerce saat döngüsüne mal olurken GPU'nun geçiş yaparken herhangi bir kaybı yoktur. Zaman kaybetmeden birçok geçişi ardı ardına yapabilir.

5.4.5. SIMD ve SIMT

CPU'lar vektörel işlemler için SIMD birimleri kullanırlar. GPU'lar ise ölçekli yürütme için tek yönergeyle çok iş parçacığı (SIMT) kullanırlar. SIMT, programcının verileri vektörler halinde düzenlemesini gerektirmez, iş parçacıkları için rastgele dağılıma olanak tanır.

5.4.6. Bellek denetleyici

Örneğin, Intel CPU'lar üzerinde dahili bellek denetleyici bulunmaz. CUDA destekli GPU'lar, 8 adete kadar dahili bellek denetleyici kullanırlar. Sonuç olarak GPU'lar, CPU'lara göre 10 kata kadar daha fazla bant genişliğine sahiptirler.

5.5. İşlem (Process) ve İş Parçacığı (Thread) Farkı

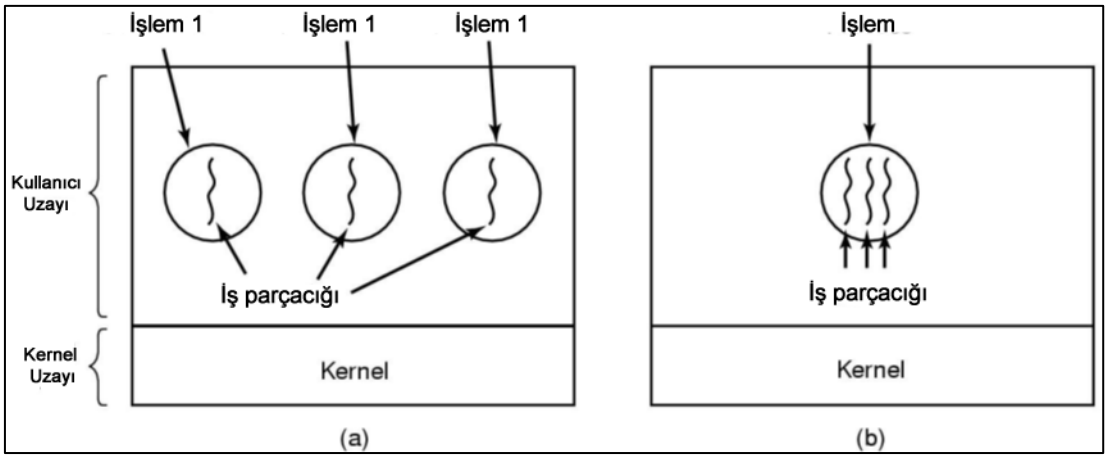
Disk üzerinde pasif halde duran yapıya program adı verilmektedir. Bu yapının belleğe yüklenerek çalıştırılmasıyla ortaya çıkan yapıya ise işlem ya da süreç adı verilmektedir. İşletim sistemi işlemin yaratılması, ortadan kaldırılması, durdurulması ve kaldığı yerden devam edebilmesi gibi işlemlerden sorumludur.

Günümüzde tüm modern işletim sistemleri aynı anda birden fazla iş yapabilmektedir. Yani aynı anda birden fazla işlem çalıştırabilmektedir. Bu durumda işletim sistemi iki işlem arasındaki senkronizasyonu sağlamak ve gerektiği takdirde iletişim sorunlarını çözmek için gerekli mekanizmaları sağlamak zorundadır.

İşlemlerin kendilerine ait kaynakları vardır. Örneğin her işlem kendine ait hafıza alanına sahiptir. Bunun yanında kendine ait program sayacı, yazmaçları (registers), yığın göstericisi (stack pointer), işlem durumu, işlem numarası vardır. İşlem genellikle uygulamanın kendisi olarak görülür ancak bir uygulama içinde birbirine bağımlı ya da birlikte çalışan işlemler meydana gelebilir.

İş parçacığı ise bir işin eş zamanlı olarak işleyen her bir bölümüdür. İş parçacığı ile işlem arasındaki fark kaynakların paylaşımıdır. İş parçacıkları işlemler içinde paralellik elde etmek için kullanılırlar. Bir işlem içerisinde en az bir olmak üzere pek çok iş parçacığı barındırılabilir (Bkz. Şekil 5.4). Aynı işlem içindeki tüm iş parçacıkları aynı hafıza alanını ve aynı durumu paylaşır, birbirleriyle direk olarak haberleşebilirler.

Çoklu iş parçacıkları paralel olarak pek çok bilgisayar sisteminde uygulanabilir. Tek işlemci kullanıldığı durumlarda çok iş parçacıklı uygulama zaman dilimleme ile gerçekleştirilir; tek işlemci farklı iş parçacıkları arasında çok hızlı geçiş yapar ve bu durumda işlemler gerçekte olmasa bile eş zamanlı koşuluyormuş izlenimi verir. Çok işlemcili sistemlerde farklı iş parçacıkları farklı işlemciler üzerinde eş zamanlı olarak çalışabilir.



Şekil 5.4. İşlem (a) ve iş parçacığı (b) çalışması

Yazılan programın çoklu iş parçacıklı çalışabilmesi için yazılan programın iş parçacığı mekanizmalarını destekliyor olması ya da destekleyen kütüphaneleri kullanabiliyor olması gereklidir.

Kernel ise çeşitli işletim sistemlerinde, yazılım ile donanım arasındaki bağlantıyı kuran, çeşitli komutları ve ayarları donanıma iletir. Bir CUDA *kernel*'i iş parçacığı dizileri tarafından çalıştırılır. Tüm iş parçacıklarında aynı kod çalıştırılır. Her bir iş parçacığının kendine ait hafıza adresleri hesaplayabilmek ve kontrol kararları alabilmek için bir kimlik numarası (ID) vardır.

5.6. CUDA (Compute Unified Device Architecture)

CUDA, NVIDIA tarafından ortaya konulan, GPU gücünü kullanarak işlem performansına yüksek oranda katkı yapan bir paralel programlama platformudur. C, C++, C#, Fortran, Java, Python gibi programlama dilleri ile yazılmış algoritmaların GPU yani "grafik işlem birimi" üzerinde çalışmasını sağlayan bir sistem olarak da tanımlanabilir. En yaygın olarak kullanılan diller C ve C++'dır. Windows, Linux ve Mac OSX işletim sistemlerini desteklemektedir.

Paralel programlama, birden fazla işlemcinin bir arada çalışmasını gerektirir. CPU'lar aslen seri işlemcilerdir ve birden fazla CPU'nun bir arada kullanılabilmesi için karmaşık yazılımlar gerekmektedir. CUDA ise bünyesinde zaten paralellik yarattığı için bu zorlukları ortadan kaldırır.

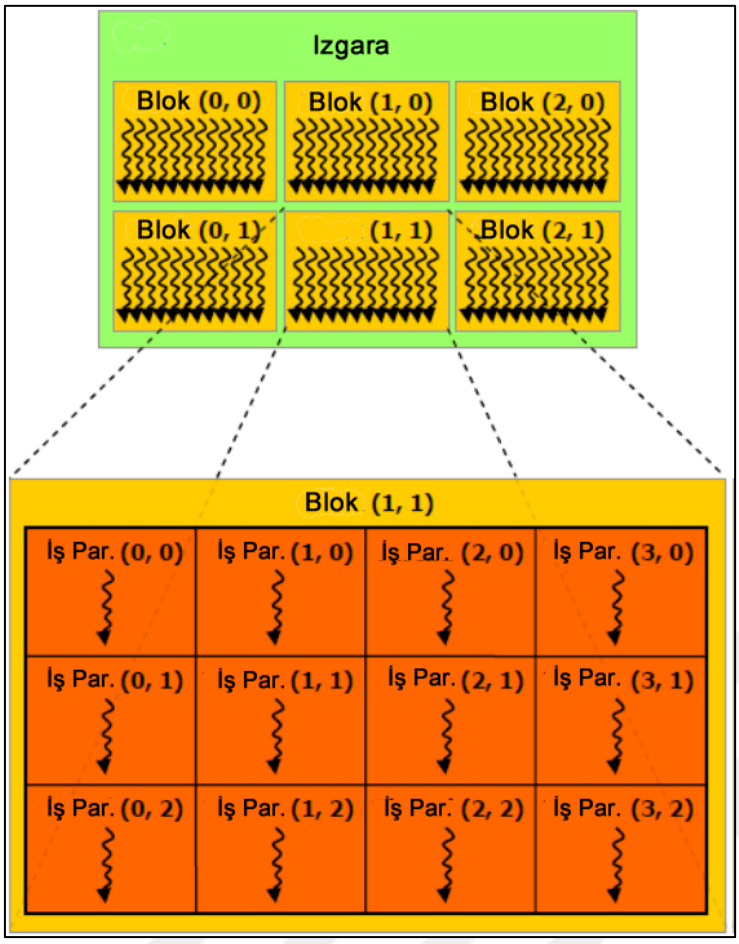
CUDA etkinleştirilmiş GPU ile görüntü ve video işlem, hesaplamaya dayalı biyoloji ve kimya, akışkan dinamiği, bilgisayarlı tomografi, sismik analiz, ışın izleme ve çok daha fazlası dahil olmak üzere geniş bir aralıkta kullanım alanı bulmaktadır.

5.6.1. CUDA aygıtları

CUDA programlamada kullanılan bazı terimler bulunmaktadır. Bunlardan en temeli yönetici (host) ve cihaz (device) kavramlarıdır. Bir CUDA programında *yönetici* CPU, *cihaz* ise GPU için kullanılmaktadır. GPU'ların kendi dinamik RAM'leri (DRAM) vardır. Bunlar cihaz belleği (device memory) olarak adlandırılır. *Kernel* GPU üzerinde çalışan fonksiyonlardır. Uygulamanın paralel kısımları pek çok iş parçacığının aynı anda çalıştığı *kernel*'lerde işlenmektedir.

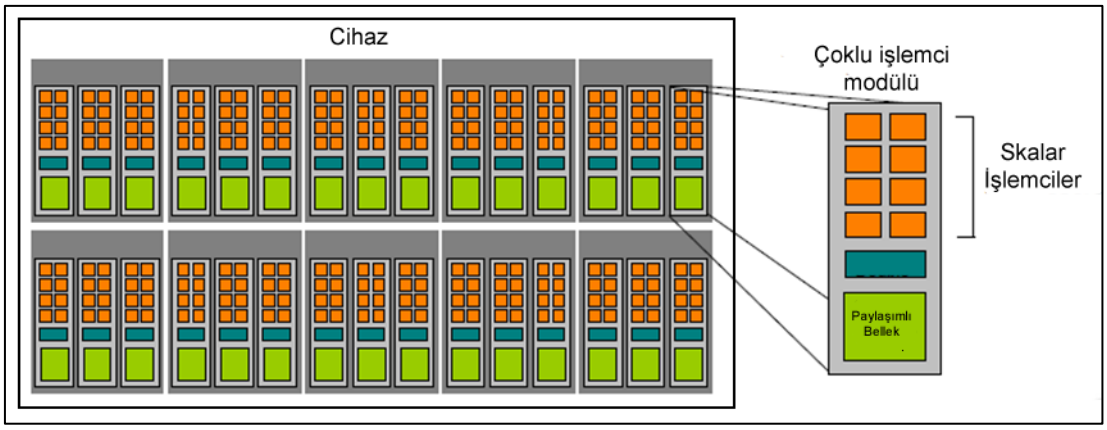
5.6.2. CUDA mimarisi

Bir *kernel* bir ızgara (grid) içerisindeki tüm iş parçacıklarında aynı anda SIMD olarak farklı veriler üzerinde aynı işlem yapılacak şekilde çalıştırılır. Bir GPU içerisinde ızgara adı verilen yapıları barındırır. Bir ızgara içerisinde bloklar bulunur. Bu bloklar içerisindeki işlemcilerde iş parçacıkları çalıştırılır. İş parçacıklarının çalışmasının şematik gösterimi Şekil 5.5'te verilmektedir.



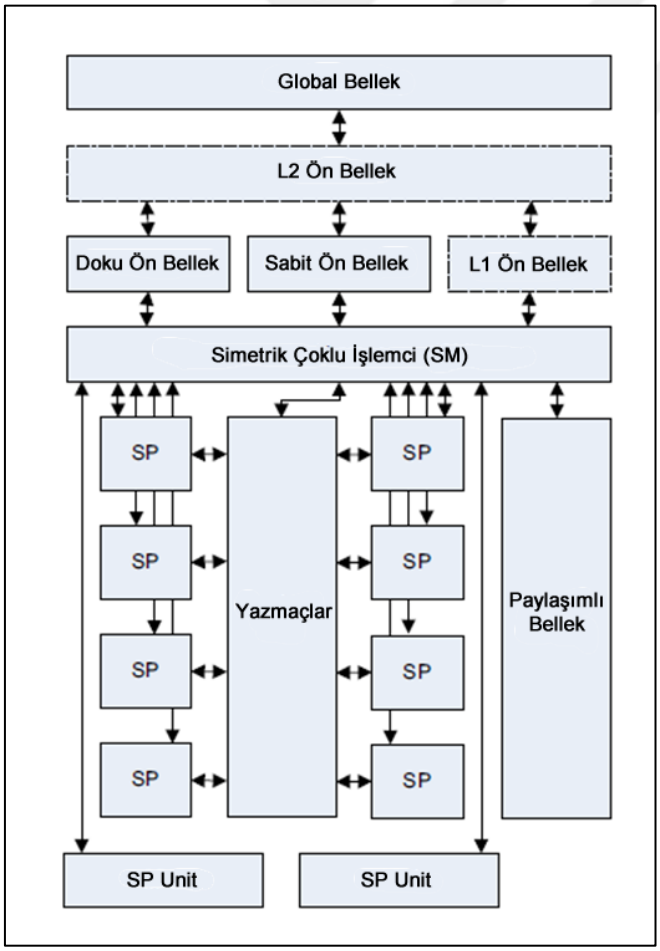
Şekil 5.5. İş parçacıklarının ızgara ve bloklarda çalıştırılması.

Blokların içerisinde Şekil 5.6'da gösterildiği gibi çok sayıda işlemci içeren ve çoklu işlemci modülü (SM-Streaming Multiprocessor) adı verilen yapılar bulunmaktadır. Her bir SM içerisinde skaler işlemciler (SP - Scalar Processor) ve bellekler bulunmaktadır. Bu SM'ler GPU'ların en küçük yapı taşlarıdırlar. SM'ler içerisinde fiziksel olarak paralel işleyen iş parçacığı gruplarına *warp* adı verilmektedir. Her bir SM içerisindeki işlemcilerin tümü aynı komutu eşzamanlı olarak işler.



Şekil 5.6. GPU mimarisi

Her bir SM içerisinde skalar işlemciler, yazmaçlar, paylaşımlı bellek (shared memory), doku bellek (texture memory), sabit bellek (constant memory), önbellekler ve çift duyarlıklı kayan nokta ünitesi (double precision unit) bulunmaktadır. SM mimarisi Şekil 5.7'de verilmektedir.



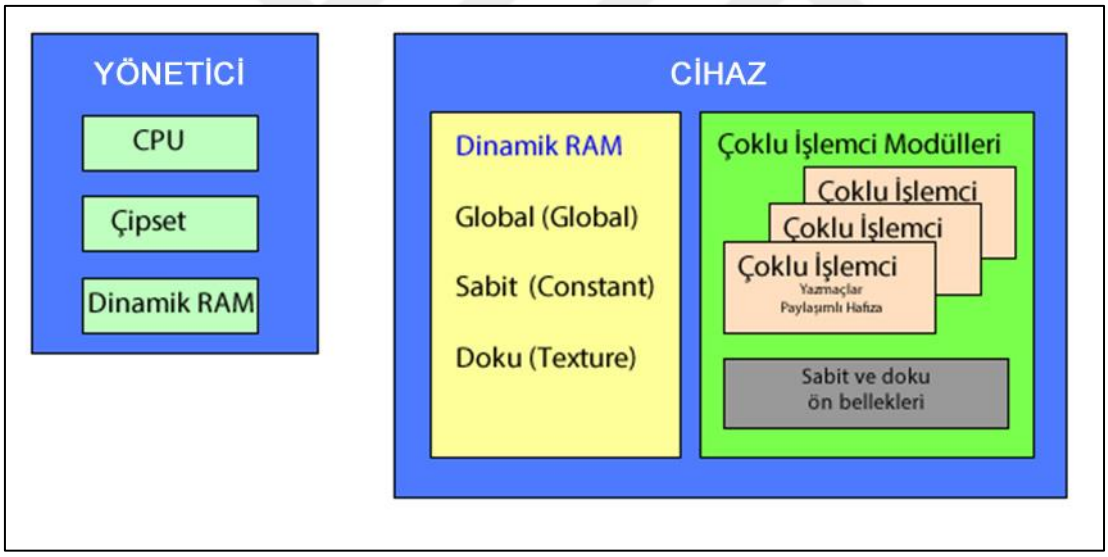
Şekil 5.7. SM'nin yapısı [53]

5.6.3. CUDA hafıza modeli

CUDA platformunda ekran kartının belleği içerisinde farklı bellek kavramları bulunmaktadır. Bunlar;

- Global Bellek
- Lokal Bellek
- Paylaşımlı Bellek
- Sabit Bellek
- Doku Bellek'tir.

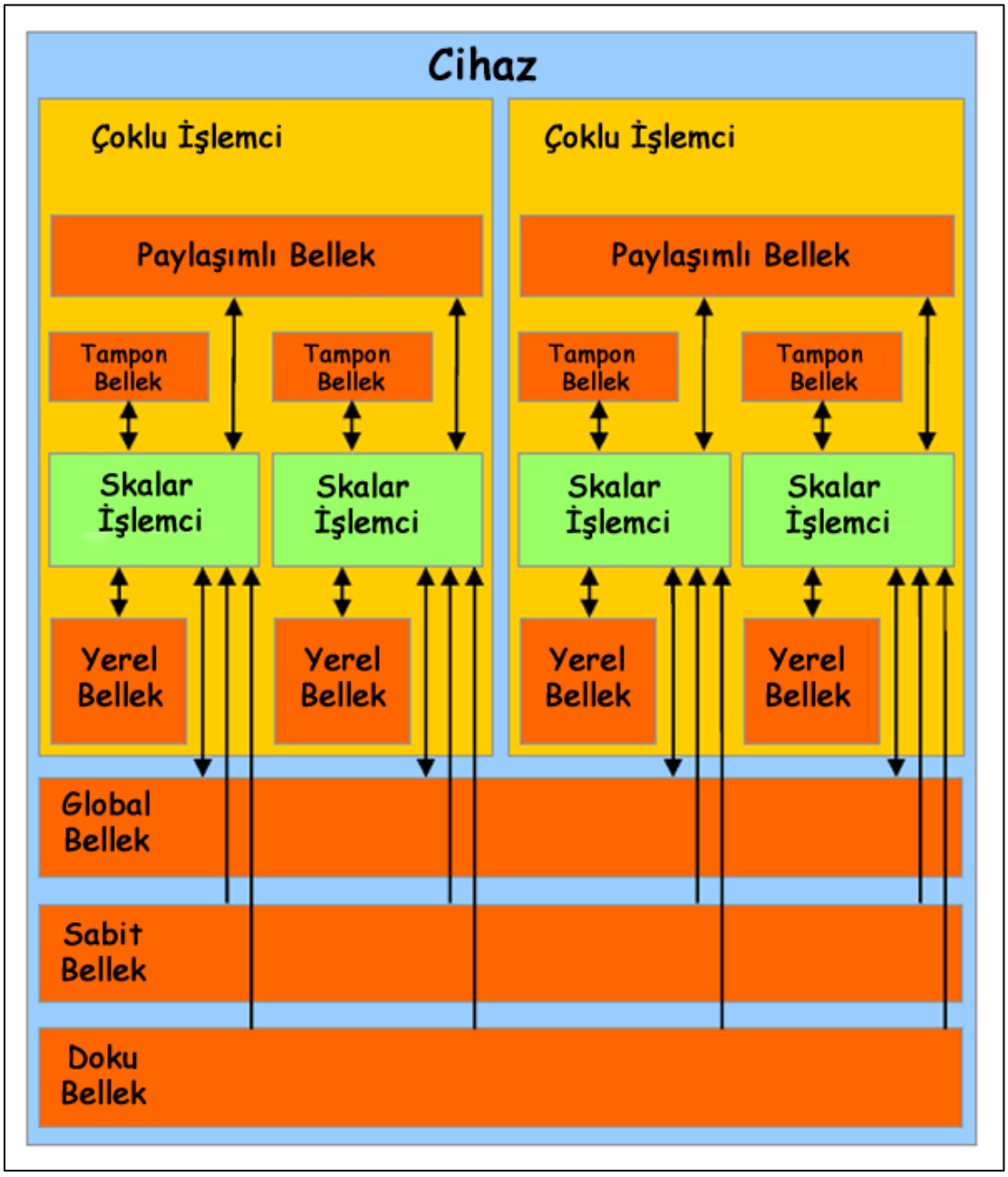
Bu belleklerin durumları Şekil 5.8'de gösterilmektedir. Global belleğe ana bellekten yani RAM'den veri aktarılabilmektedir. Bu aktarma CUDA programında *cudaMemcpy* komutu ile yapılmaktadır.



Şekil 5.8. CUDA Hafıza Modeli

NVIDIA GPU içerisinde belleklerin yerleşimleri ve aldıkları isimler Şekil 5.9'da gösterilmektedir. Her bir çoklu işlemci yani SM'in paylaşımlı bellek adını alan kendi belleği vardır. SM içerisindeki her bir işlemcinin kendine ait yazmaçları bulunmaktadır. Ayrıca sabit ve doku belleklerin önbellekleri de SM içerisinde bulunmaktadır. Bunlardan veri sadece okunabilmektedir. Global bellek adını alan bellek ise ekran kartının belleğidir. Aslında sabit ve doku bellekler de global bellek içerinde yer almaktadır. Bellekler hızları açısından karşılaştırıldıklarından en hızlı yazmaçlardır. Paylaşımlı bellek de çok hızlı

çalışmaktadır. SM içerisindeki işlemcilerin tümü paylaşımlı belleğe aynı hızla ulaşabilmektedir. Global, sabit ve doku belleklerin ise ekran kartının belleğidir ve verilerin işlemciye ulaşım süresi fazladır. Veriler DRAM'den GPU'ya aktarılırken global belleğe gelirler.



Şekil 5.9. GPU'da bellekler ve işlemcilerin yapısı

5.6.4. Hafiza optimizasyonu

CUDA platformunda çalıştırılan yazılımların daha iyi sonuç vermesi için donanımla ilgili bazı özelliklere dikkat edilmesi gerekmektedir. Bu durumlardan en önemlileri bellek bütünleştirme (memory coalescing), yığın çatışması (bank conflict) ve iş parçacığı ayrılma (thread divergence) dır.

CUDA platformunda çalıştırılan yazılımların daha iyi sonuç vermesi için donanımla ilgili yukarıdaki durumların dışında da önem verilmesi gereken unsurlar paylaşımlı hafızanın kullanılması ve atomik operasyonlardır. CPU-GPU arasındaki veri iletişimini kullanırken bant genişliği de önemli bir unsur olarak karşımıza çıkmaktadır.

Bant Genişliğinin Hesaplanması

Hafiza optimizasyonu için dikkat edilmesi gerekenlerden biri CPU'dan GPU'ya verilerin hangi hızla aktarıldığının bilinmesidir. Kullandığımız ekran kartının özelliğine göre veri iletim hızı değişmektedir. Gerçekleştirilen uygulamalarda bant genişliğini en iyi şekilde kullanılması uygulamanın performansını önemli ölçüde değiştirmektedir. Kullanılan donanım iyi tanınması ve ona uygun uygulama geliştirilmesi optimizasyon için çok önemlidir.

Kullanılan ekran kartına göre bant genişliğinin hesaplanmasının nasıl yapıldığını göstermek için örnek olarak kullanılan Geforce GT 635M ekran kartının özellikleri aşağıda verilmektedir;

GPU Aygıt Özellikleri:

384	CUDA Çekirdeği
967	Grafik Saat Frekansı (MHz)
967	Artırılmış Saat Frekansı (MHz)

Bellek Özellikleri:

1001 MHz	Bellek Frekansı
2048 MB	Bellek Miktarı
DDR3	Bellek Tipi

64-bit Bellek Arabirimi Genişliği

16.0 Bellek Bant Genişliği (Gigabayt (GB)/s)

Buna göre verilen özelliklere göre bant genişliği şu şekilde hesaplanır;

$$Bant\ Genişliği = \frac{Saat\ Frekansı \times Bellek\ Arabirimi\ Genişliği \times 2}{8 \times 1024^3} GB/s \quad (5.1)$$

Eş. 5.1'e göre Geforce GT 635M ekran kartının bant genişliği şöyle hesaplanır;

Bant Genişliği = 1001 X 10^6 X 64 / (8 X 2 / 1024^3) = 14,9 GB/s

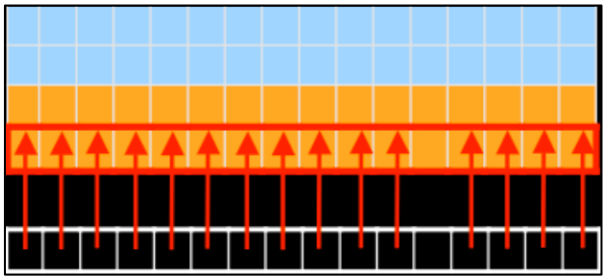
Geforce GT 635M ekran kartının özellikleri yukarıda verilmektedir. Ekran kartının saniyede yapabileceği veri transferi 14,9 GB olarak hesaplanmıştır. Yukarıda verilen bilgilerde bant genişliği 16,0 GB/s olarak verilmektedir. Arada fark olmasının nedeni bayt olarak bulunan verinin gigabayta çevrilirken 1024^3'e bölünerek hesaplanmasıdır. Ancak verilen özelliklerde 1000^3'üne bölünerek bulunmuştur.

Geforce GT 635M ekran kartında veri iletişimi gerçekleştirilirken maksimum bant genişliği saniyede 14,9 GB olacaktır. Uygulama yazılırken bu durum dikkate alınarak optimizasyon gerçekleştirilirse en iyi performans elde edilecektir. Burada hedef bant genişliğinin en verimli şekilde kullanılmasıdır. Uygulamanın durumuna göre bant genişliğinin mümkün olduğunca kullanarak performansı artırmak mümkün olabilmektedir.

Bellek bütünleştirme (Memory coalescing)

GPU'larda veriler GPU'nun genel belleğine depolanır ve önbellek kullanılmaz. Küresel belleğin önbelleğe alınmadığı gerçeği göz önüne alındığında, maksimum bellek bant genişliğini elde etmek için doğru erişim düzenini takip etmek daha önemlidir. Genel bellek bant genişliği, bir *halfwarp* halinde eşzamanlı bellek erişimi (tek bir okuma veya yazma komutunun yürütülmesi sırasında) 32, 64 veya 128 baytlık tek bir bellek işlemine birleştirildiğinde en verimli şekilde kullanılır. Bu nedenle, genel bellek erişim işlemleri, bitişik bellek alanlarında hizalanabilir. Bitişik olmayan bellek konumundaki işlemler de desteklenir, ancak bunlar en azından büyüklük sırasıyla daha yavaş ve bazı durumlarda daha

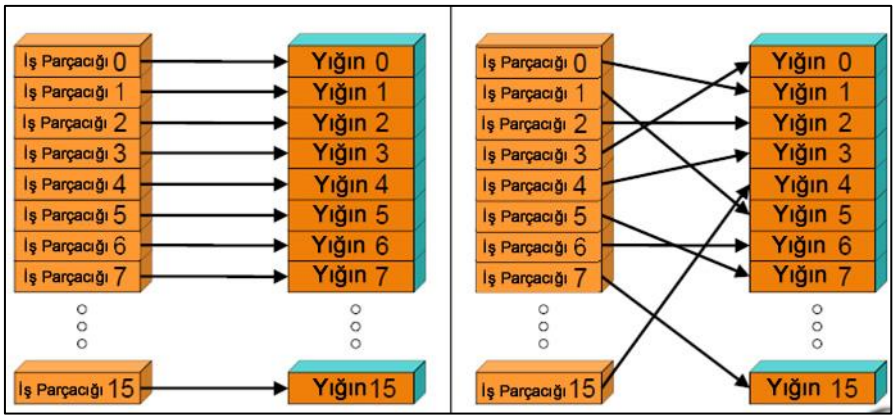
da fazla olur. Bu nedenle bir iş parçacığı bloğu içindeki iş parçacıkları aynı anda global bellekten peş peşe elemanlara erişmelidirler. Bu durumda tek bir bellek işleminde bellek erişimi gerçekleşmiş olur. Bu duruma "bellek bütünleştirme" (memory coalescing) denir (Bkz. Şekil 5.10).



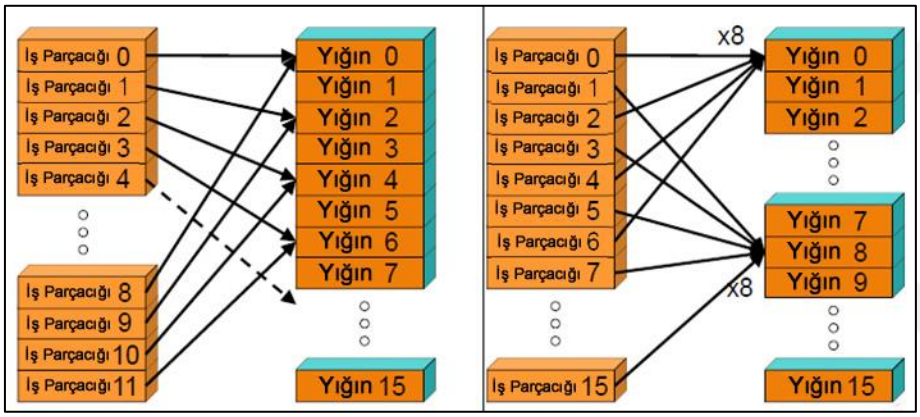
Şekil 5.10. Bütünleşmiş bellek

Yığın çatışması (Bank conflict)

Bir iş parçacığı bloğu içerisinde birden fazla iş parçacığı aynı bellek bloğuna erişmek isterse bu durum "yığın çatışmasına" neden olur. Bu durumda aynı anda aynı bellek bloğuna erişim olamayacağından bu işlem seri olarak gerçekleştirilir. Bu da işlem zamanını arttırır. İş parçacıkları bellek ile işbirliği içerisinde çalışmaktadır. Bellek "yığın (*bank)*" adı verilen parçalara ayrılmaktadır. Fermi mimarisinde her bir *yığın* 32 bit, Kepler mimarisinde ise her bir *yığın* 64 bittir. Bu parçalara aynı "*halfwarp"* içerisinden en kısa zamanda ulaşılabilmektedir. Her bir *halfwarp'*ta 16 iş parçacığı çalışmakta ve bu 16 iş parçacığı 16 *yığına* ulaşabilmektedir. Ancak bir iş parçacığı aynı anda bir *yığına* ulaşabilmektedir. Bir iş parçacığı aynı anda 2 *yığına* ulaşmak isterse bu durum iki katı zaman almaktadır ve bu durum "*yığın çatışması"* olarak adlandırılmaktadır.



Şekil 5.11. Yığın çatışması olmayan bellek adresleme



Şekil 5.12. Yığın çatışması gerçekleşen bellek adresleme

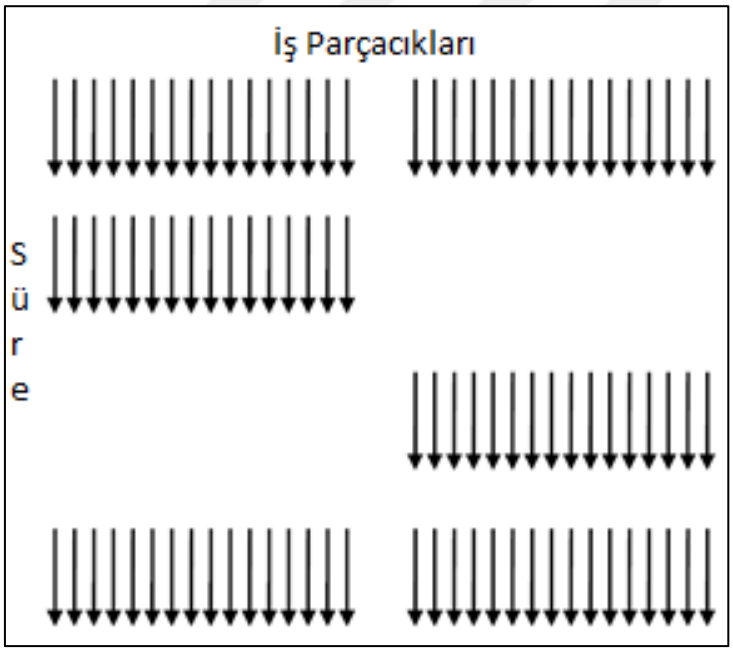
Şekil 5.11'de her bir iş parçacığı bir adet bellek bölgesine ulaşmakta ve her bir iş parçacığı aynı anda belleğe ulaştığından en kısa zamanda işlem gerçekleşmektedir. Şekil 5.12'de ise bir iş parçacığı 2 ve 4 adet bellek bölgesine aynı anda ulaşmak istemekte ve yığın çatışması oluşmaktadır. Bu durumun gerçekleşmesi 2 kat ve 4 kat zaman almakta yani iş parçacıkları birbirlerini beklemektedirler. Uygulamalarda yığın çatışması oluşturulmaması bellek optimizasyonu açısından önemlidir ve performansı etkilemektedir.

İş parçacığı ayrılma (Thread divergence)

GPU mimarisi çok iş parçacıklı yapıdadır. Çok çekirdekli büyük paralel mimarinin üst seviyesinde, iş parçacıkları bir ızgaraya organize olurlar. Izgara, iki boyutlu bloklar kümesi içerir. Bir blok tek bir SM üzerinde çalışır ve genel olarak diğer bloklarla senkronize edilemez. Blok içindeki iş parçacıkları, bir SM içindeki farklı çekirdeklere (SP) tahsis edilirler. Bir bloktaki tüm iş parçacıklarının kendi saklayıcıları vardır ve az yük getiren bir

bariyerle aynı işlevi gören iş parçacıkları senkronizasyon fonksiyonu vardır. İş parçacıkları *warp*'lara organize olurlar. *Warp*, 32 paralel iş parçacığından oluşan bir yapıdır. Bir warp bir anda bir komut çalıştırır, bu yüzden sadece *warp*'taki 32 iş parçacığının hepsi aynı çalıştırma yoluna sahipken tam verimliliğe ulaşılabilir. Bu durum iki önemli sonuç doğurur. Birincisi, bir *warp*'taki iş parçacıkları koşullu dallanma nedeniyle farklı çalıştırma yollarına sahipse, warp her dalı seri bir şekilde çalıştıracaktır. Bu duruma *"iş parçacığı ayrılma"* (divergence) denir (Bkz. Şekil 5.13) ve warp için çalışan komutların toplam zamanını arttırır. İkincisi, bir bloktaki iş parçacıklarının sayısı warp boyutunun bir katı olmazsa, kalan komut döngüleri boşa gidecektir [56].

```
if (condition) {
do smth
} else {
do smth
}
```



Şekil 5.13. Ayrılma ile *warp* içerisinde işlemler

Bu bölümde GPU üzerinde geliştirilen kodların en verimli şekilde çalıştırılması amacıyla dikkat edilmesi gereken bellek optimizasyon ilkelerinden bahsedilmiştir. Oluşturulan algoritmalar hızlanmanın istenen düzeyde olması amacıyla bu ilkeler göz önüne alınarak

yazılmalıdır. CPU algoritmalarının GPU uygulaması gerçekleştirilirken algoritmalar üzerinde düzenlemeler yapılması gerekmektedir. Başka bir deyişle bir CPU kodunu GPU için paralelleştirirken GPU'nun donanımı göz önünde bulundurularak yeni bir algoritma oluşturulmalıdır. Yeni algoritmanın GPU bellek optimizasyonu ilkelerine uygun alarak tasarlanması gerekmektedir. Bu durumda işlem zamanı azalmasını sağlamaktadır.

Atomik operasyonlar

Birden fazla iş parçacığının aynı anda çalıştırıldığı paralel uygulamalarda iş parçacıkları aynı anda aynı hafıza bölgelerini ya da aynı kaynakları kullanmak isteyebilirler Bu duruma "race condition" adı verilir. CUDA uygulamalarında "race condition" durumunu önlemek amacıyla atomik operasyonlar kullanılır.

Atomik operasyonlar başka iş parçacıklarının aynı bellek kaynaklarını okuma, yazma, değiştirme gibi işlemleri yapmalarını engellemek amacıyla kullanılırlar. Atomik operasyonlar CUDA'da hem paylaşımlı bellekte hem de global bellekte çalışabilmektedir. Atomik operasyonlar paylaşımlı bellekte genellikle aynı iş parçacığı bloğu içerisindeki farklı iş parçacıkları arasında aynı kaynakların kullanmasını önlemek amacıyla kullanılırlar. Global bellekte ise hangi iş parçacığı bloğu olduğuna bakmadan farklı iki iş parçacığının aynı kaynakları kullanmasını önlemek amacıyla kullanılmaktadırlar.

Atomik operasyonlar bazı durumlarda gerekli olabilmektedir. Ancak önemli olan bu durumun özellikle global belleğe ulaşım söz konusu ise en aza indirilmesi gerekmektedir. Eğer iki iş parçacığı atomik operasyon ile aynı hafıza adresine ulaşmak isterse bu operasyon seri olarak gerçekleştirilecektir. Bu nedenle performans açısından atomik operasyonların çok dikkatli kullanılması gerekmektedir.

Paylaşımlı bellek kullanımı

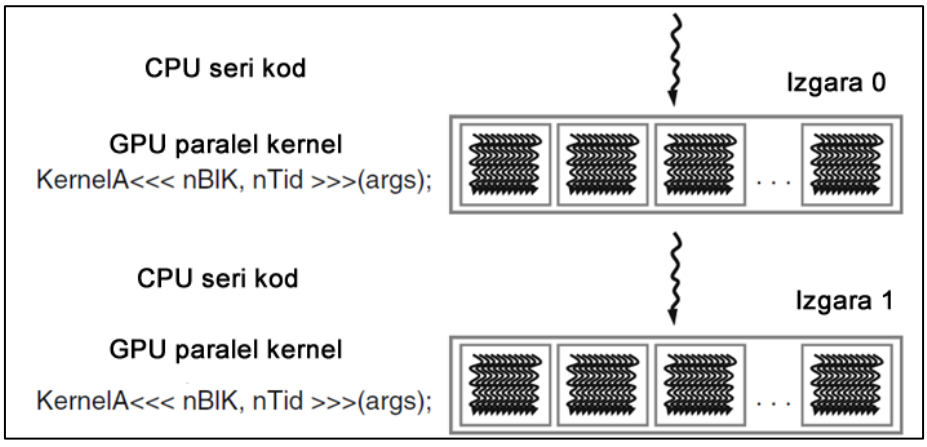
Paylaşımlı bellek ekran kartında yonga içerisinde bulunan bir bellek çeşididir. Bu nedenle buraya erişim çok hızlı olmaktadır. Çip dışında olan global belleğe erişim 400-600 saykıl zaman alırken paylaşımlı bellekten veri okuma 4 saykıl almaktadır. Yani global belleğin 100 katı kadar hızlı çalışmaktadır. Bu nedenle paylaşımlı bellek kullanımı hız açısında çok önemlidir.

5.6.5. Bir CUDA programının yapısı

Bir CUDA programı Şekil 5.14'te gösterildiği gibi hem CPU'da hem de GPU'da işlenecek olan kodları içerir. Bu kodların CPU'da işlenecek olan bölümlerinde herhangi bir veri paralelliği yoktur. GPU'da işlenecek olan kod bölümü veri paralelliği içeren kodlardan oluşur. Program hem CPU'da hem de GPU'da çalışan tek bir kaynak koddan oluşur. NVIDIA derleyicisi kodu derleme sırasında iki bölüme ayırır. *Yönetici* kodu CPU'da çalışır ve standart C derleyicisi tarafından derlenerek çalıştırılır. *Aygıt* kodları ise veri paralelliği içeren fonksiyonlardır. GPU'da çalışan bu fonksiyonlara "*kernel"* adı verilir. *Kernel*'ler CUDA SDK'sı tarafından çalıştırılırlar. *Kernel*'ler çok sayıda iş parçacığından oluşur. Bu iş parçacıkları CPU'da işlendiğinde sıralı olarak ve çok sayıda saykılda işlenecekken GPU'da aynı anda her biri farklı bir işlemcide farklı veriler kullanarak ve aynı kodla işlenirler.

GPU üzerinde çalışacak bir *kernel*'in işlem adımları şu şekildedir (Bkz. Şekil 5.14);

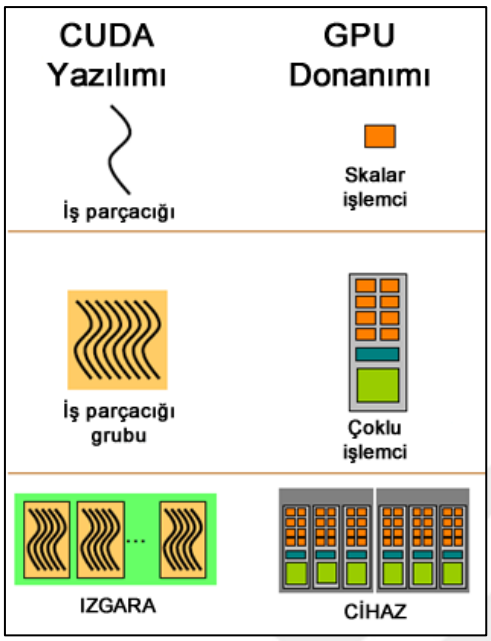
1. İşlenecek olan veriler CPU'dan GPU'ya aktarılır.
2. CPU, GPU'ya *kernel*'in işlenmesi için komut verir.
3. GPU *kernel*'i işler.
4. İşlenen veriler GPU'dan CPU'ya aktarılır.



Şekil 5.14. Bir CUDA programının işlemesi [57]

CUDA platformunda yazılım ve donanım kavramlarını eşleştirdiğimizde Şekil 5.15'te verilen durum ortaya çıkmaktadır. Buna göre her bir iş parçacığı bir işlemcide işlenir. Bir blok içerindeki iş parçacıkları bir SM'de işlenirler. Bu işlemler eşzamanlı olarak

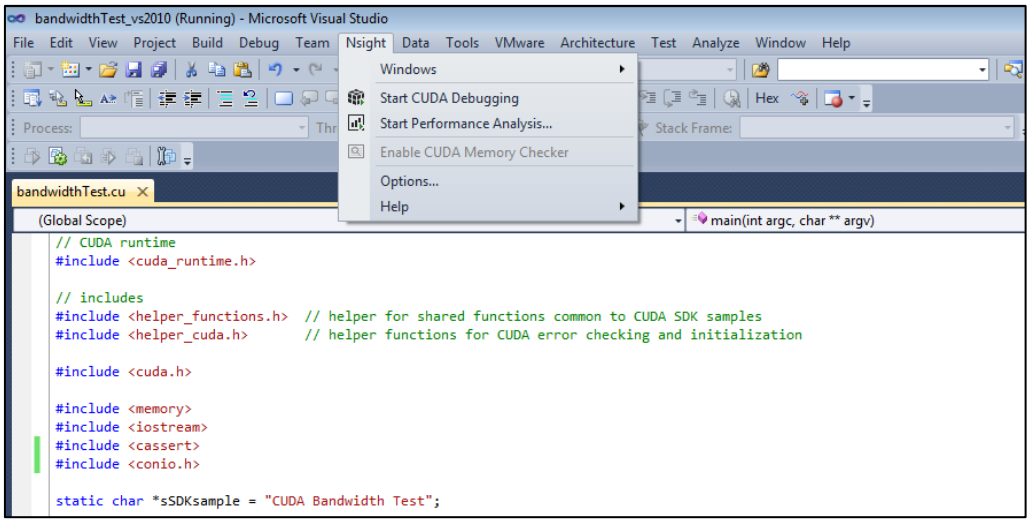
gerçekleştirilir. Blokların bir araya gelmesiyle ızgara adı verilen yapılar oluşur. Bir kernel bir ızgara içerisindeki bloklarda çalıştırılır. Bir blokta ancak bir kernel çalıştırılabilir.



Şekil 5.15. CUDA platformunda yazılım donanım kavramlarının eşleştirilmesi

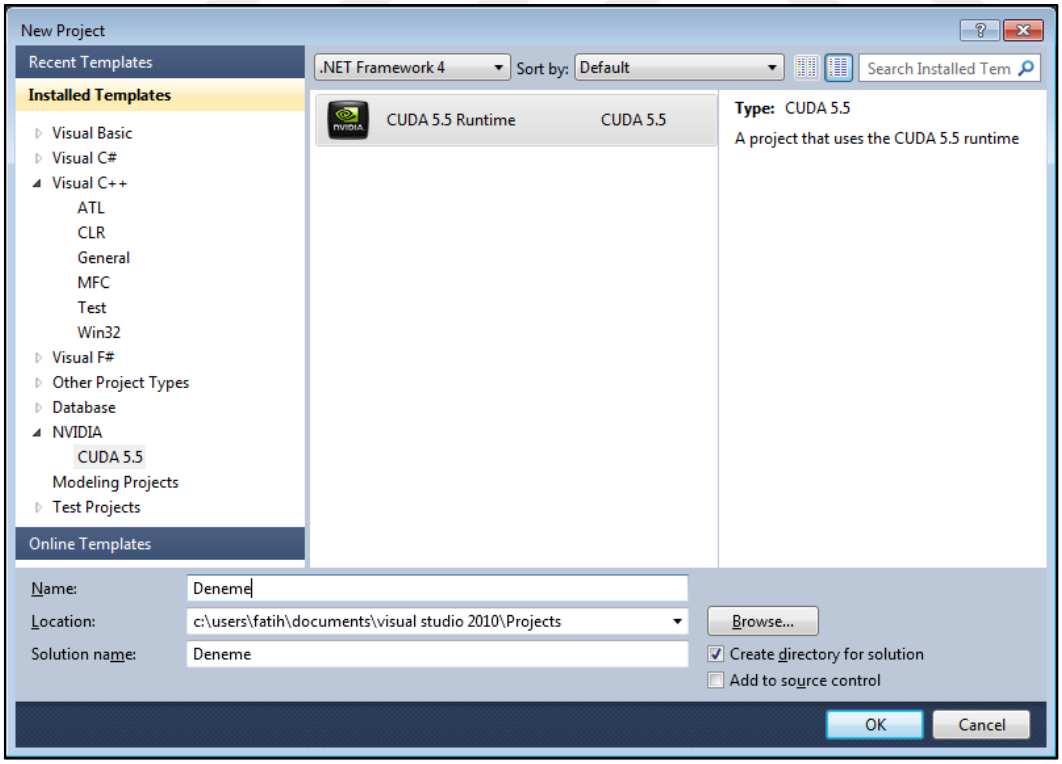
5.6.6. CUDA Araç Kiti ve donanım test araçları

CUDA Araç Kiti (Toolkit) NVIDIA'nın https://developer.nvidia.com/cuda-downloads sayfasından ücretsiz olarak indirilebilmektedir. Windows, Linux ve Mac OSX işletim sistemlerinin çeşitli sürümleri ve dizüstü ve masaüstü bilgisayarlar için farklı indirme seçenekleri vardır. CUDA Araç Kiti içerisinde CUDA araçları, CUDA kullanılan çeşitli örnekler, CUDA kullanımını anlatan dokümanlar ve Nsight eklentisi bulunmaktadır. Yüklendikten sonra Visual Studio programında C++ dilinde yazdığımız kodlarda CUDA platformu kullanılabilir. Ayrıca menüye Nsight seçeneği eklenir ve Nsight araçlarını kullanmaya imkân verir (Bkz. Şekil 5.16).



Şekil 5.16. Nsight menüsü ekran görüntüsü

CUDA Araç Kiti yüklendikten sonra Visual Studio'da CUDA yazılımı geliştirmek için yeni proje oluştururken NVIDIA kısmından CUDA Runtime seçeneğine tıklanması gerekir (Bkz. Şekil 5.17).



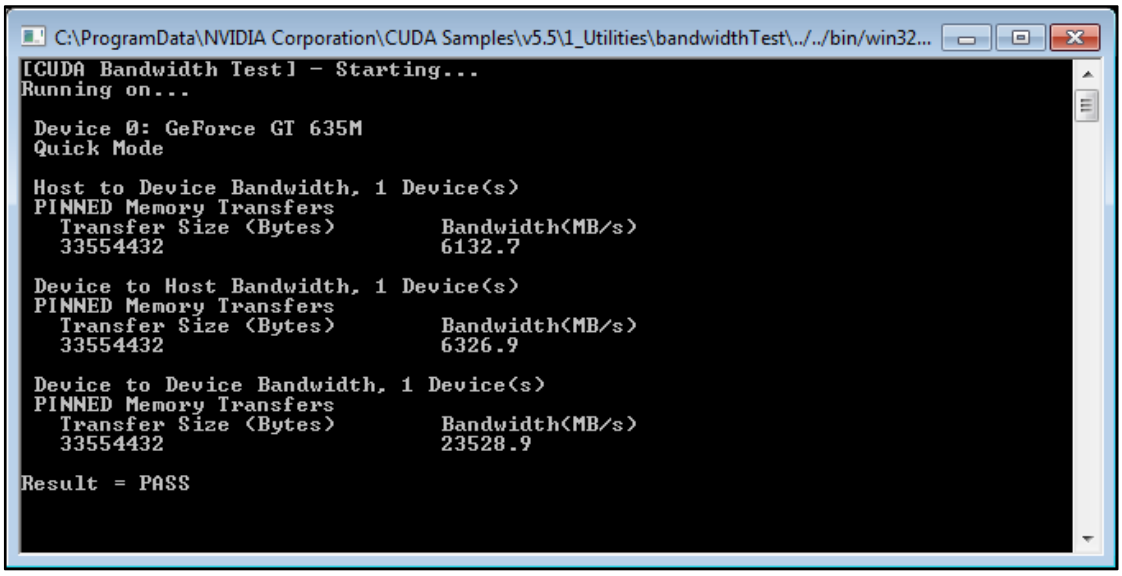
Şekil 5.17. Visual Studio'da CUDA projesi oluşturma

CUDA platformunda yazılım geliştirirken cuda.h ve cudaruntime.h kütüphanelerinin eklenmesi gerekmektedir. CUDA *kernel*lerini çalıştırıldığı dosya uzantısı *.cu* olmalıdır.

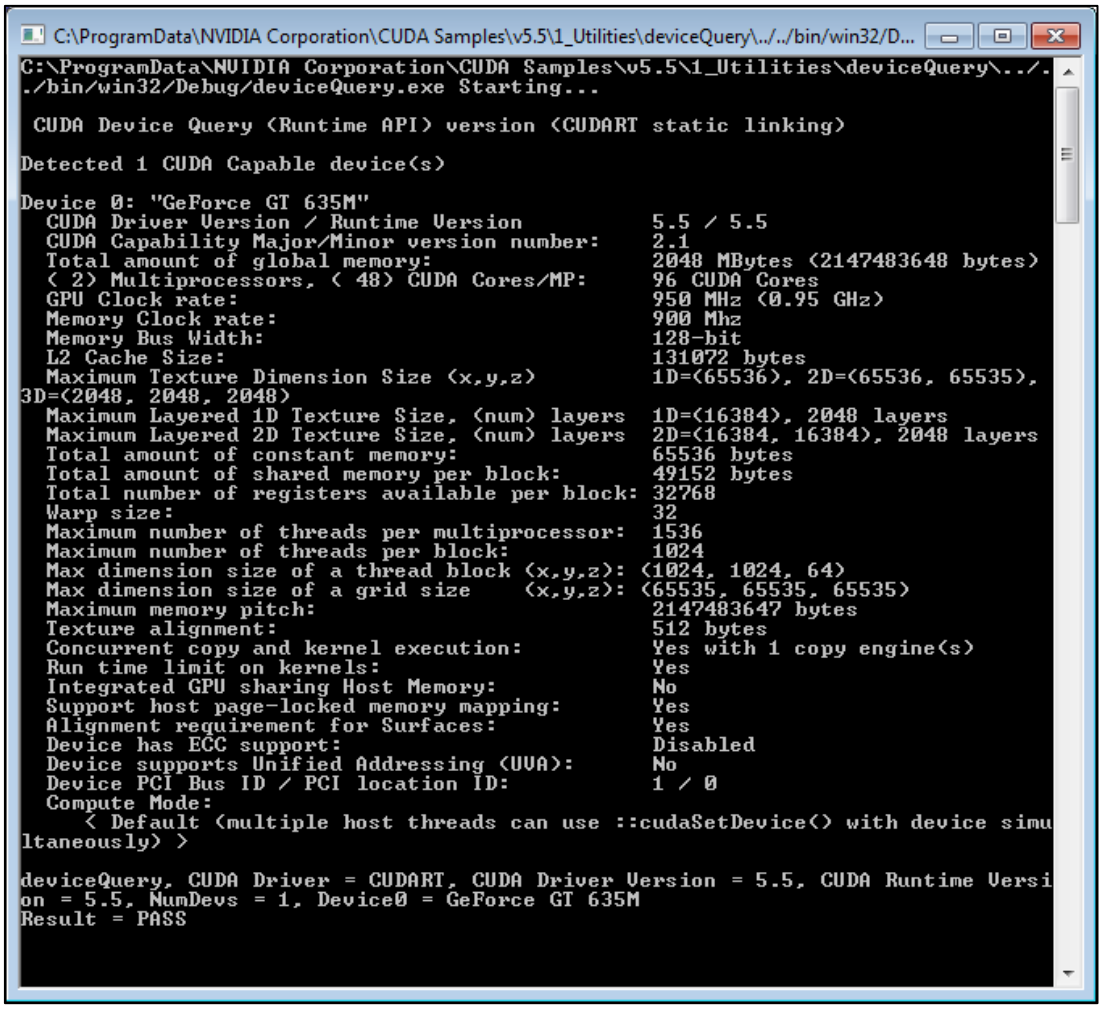
5.6.7. CUDA platformunda kod çalıştırma

Bilgisayarınızdaki CUDA destekli ekran kartının modelini, işlemciden ekran kartına, ekran kartından işlemciye ve ekran kartından ekran kartına verilerin bant genişliğini yani veri gönderme hızını veren yazılımın çıktısı Şekil 5.18'de verilmektedir. Bu bant genişliği en yüksek bant genişliği olup gerçekleştirilen yazılımların teknik olarak ulaşabileceği en yüksek bant genişliğidir. Ancak bu bant genişliğine ulaşmak neredeyse imkânsızdır. Yazılan programlarda kullanılan bant genişliğinin mümkün olduğunca arttırılmasına çalışılır. Bu durum yazılan kodun veri aktarımı açısından daha optimize çalışmasını sağlar.

Bilgisayardaki CUDA destekli ekran kartının modelini ve teknik özelliklerini belirleyen bant genişliği test yazılımının örnek çıktısı Şekil 5.19'da verilmektedir. Bu teknik özellikler göz önünde bulundurularak yazılım geliştirilmesi gerekmektedir. Donanım dikkate alınmadan geliştirilen yazılımlar ile hızın yeteri kadar artırılması mümkün olamamaktadır. Ekran kartının sahip olduğu donanımın mümkün olduğunca verimli kullanılması için bu değerlerin bilinmesi gerekir. Bu nedenle optimizasyon için donanım iyi bilinmeli ve yazılımda kullanılmalıdır.



Şekil 5.18. Ekran kartı modeli ve bant genişlikleri



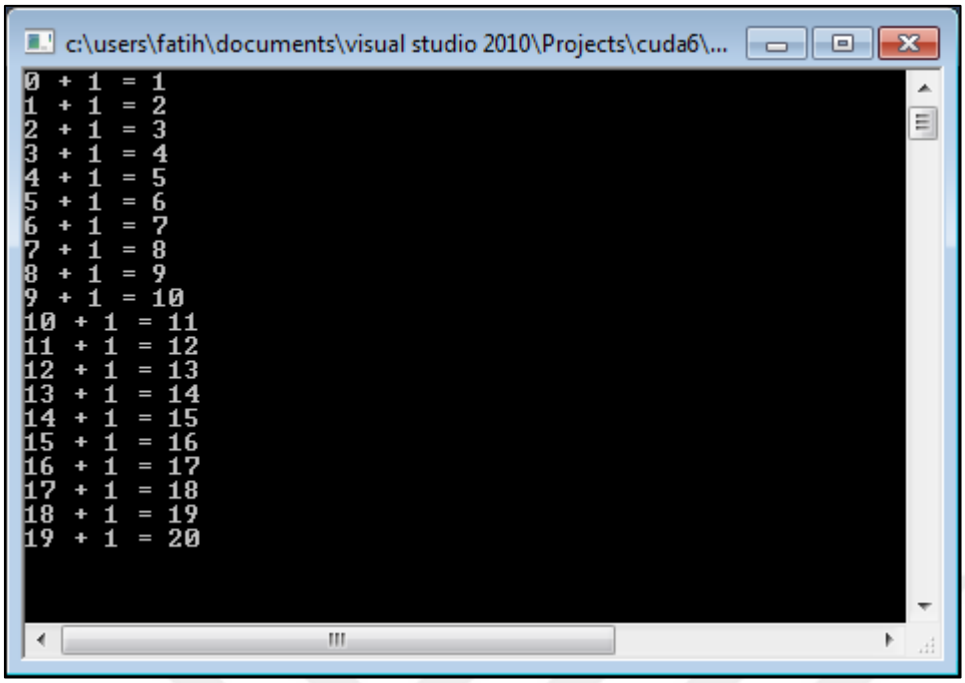
Şekil 5.19. Ekran kartının teknik özellikleri

CUDA programlamaya örnek olması bakımından iki matrisin toplamını hesaplayan CUDA programı verilmektedir. CUDA kodlarını kullanabilmek için *cuda.h* kütüphanesinin eklenmesi gerekmektedir. Burada a ve b matrislerini toplayıp, toplam değeri c matrisine yazan bir fonksiyon yazılmıştır. *threadIdx* değeri çalışan iş parçacıklarının numaralarıdır. *cudaMalloc* komutu ekran kartının hafızasından yer ayırmaktadır. *cudaMemcpy* komutu ise parametrelerine göre, işlemci belleğinde ekran kartı belleğine, ekran kartı belleğinden işlemci belleğine veri kopyalamak için kullanılmaktadır.

add<<<1,N>>>(dev_a, dev_b, dev_c); komut satırı GPU üzerinde *kernel*in çalıştırıldığı komuttur. Bu komutta bir blokta N adet iş parçacığı çalıştırılmaktadır. Programın çıktısı Şekil 5.20'de verilmiştir. Çalıştırılan kod parçacığı aşağıdaki gibidir:

```
#include <cuda.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>/* printf, scanf, puts, NULL */
#include <conio.h>

#define N 20
__global__ voidadd(int *a, int *b, int *c)
{
	int tID = threadIdx.x;
	if (tID< N)
		{
			c[tID] = a[tID] + b[tID];
		}
}
int main()
{
	int a[N], b[N], c[N];
	int *dev_a, *dev_b, *dev_c;
	cudaMalloc((void **) &dev_a, N*sizeof(int));
	cudaMalloc((void **) &dev_b, N*sizeof(int));
	cudaMalloc((void **) &dev_c, N*sizeof(int));
	// FillArrays
	for (int i = 0; i < N; i++)
		{
			a[i] = i;
			b[i] = 1;
		}
	cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
	cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);
	add<<<1,N>>>(dev_a, dev_b, dev_c);
	cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);
	for (int i = 0; i < N; i++)
		{
			printf("%d + %d = %d\n", a[i], b[i], c[i]);
		}
	getch();
	return 0;
}
```



Şekil 5.20. CUDA program çıktısı

5.6.8. CUDA kütüphaneleri

NVIDIA, CUDA'yı giriş seviyesinde kullanan kullanıcılar için bazı kütüphaneler geliştirmiştir. CUDA araç kitinin içerisinde olan kütüphaneler şunlardır;

- CUFFT : Fourier dönüşümleri
- CUBLAS : Lineer cebir yoğunluklu
- CUSPARSE: Aralıklı lineer cebir
- LIBM : Standart C Matematik Kütüphanesi
- CURAND: Pseudo-random and Quasi-random numbers
- NPP: Görüntü ve sinyal işleme
- Thrust: Temel kütüphane

Diğer açık kaynak veya ticari kütüphaneler ise şunlardır;

- MAGMA: Lineer cebir - OpenVidia: Bilgisayarlı görme
- CULA Tools: Lineer cebir - OpenCurrent: CFD
- CUSP: Aralıklı lineer çözümler - NAG: Hesaplamalı finans

Bunlardan iki tanesi genel bilgi vermek amacıyla incelenecektir. İncelenecek olan kütüphaneler Thrust ve NPP kütüphaneleridir.

Thrust Kütüphanesi

Thrust, C++ programlama dilinde kullanılan paralel uygulamalar geliştirmek için hazırlanan bir kütüphanedir. CUDA, TBB, OpenMP gibi platformları destekleyen ve paralel uygulamalar geliştirmeye yarayan araçları içermektedir.

Vektörler

Thrust'ta verilerin tanımlanmasını ve hafızada yer ayrılmasını sağlayan "*container*" adı verilen yapılar vardır. Bunlar; "*host_vector*" ve "*device_vector*" adını alırlar. *Host_vector* CPU'nun hafızasında belirtilen veri tipinde ve belirtilen sayıda bir dizi ya da sayı verilmezse değişken için yer ayırmaktadır. *Device_vector* ise aynı işlemi GPU'nun genel hafızasında yer ayırarak yapmaktadır. Bu iki komutu kullanmak için kütüphanelerinin eklenmesi gerekmektedir.

Kullanımı şu şekildedir;

```
#include<thrust/host_vector.h>
#include<thrust/device_vector.h>
...
thrust::host_vector<veri tipi>degisken_ismi(boyut);
thrust::device_vector<veri tipi>degisken_ismi(boyut);
```

Örnek:

```
thrust::host_vector<int> H(4);
//CPU'da H adında 4 elemanlı tamsayı bir dizi tanımlamaktadır
thrust::device_vector<int> D = H;
//GPU'da D isimli bir değişken tanımlamakta ve H dizisini buna eşitlemektedir.
```

Yineleyiciler (Iterators)

Vektörlerin elemanlarına ulaşırken yineleyiciler (iterators) adı verilen yapılar kullanılır. Bu yapılar dizilerde ilk ve son elemanı göstererek ilk elemandan son elemana kadar verilere ulaşmak için kullanılırlar. Tanımlanan bir vektörün *.begin* ile ilk elemanına, *.end* ile son elemanına ulaşılır.

D dizisinin boyutunu hesaplayan örnek kodu şöyledir;

```
thrust::device_vector<int>::iteratorbegin= D.begin();
thrust::device_vector<int>::iteratorend=D.end();
int D_uzunluk = end-begin;
```

Fonksiyonlar

Thrust kütüphanesi 75 adet fonksiyon içermektedir. Bu fonksiyonlar genel amaçlı temel bazı fonksiyonların CUDA ile paralelleştirilmiş halidir. Bazıları C++ kütüphanesinde bulunan fonksiyonlarla aynı isimli fonksiyonlardır. Bu fonksiyonlardan en çok kullanılanlar şu şekildedir;

- *reduce*: Bir dizinin toplamını alır.
- *sort*: Bir diziyi küçükten büyüğe sıralar.
- *copy:* Bir diziyi başka bir diziye kopyalar.
- *fill*: Bir diziyi verilen değerle doldurur.
- *sequence*: Bir diziyi sıralı değerlerle doldurur.
- *count*: Bir dizideki bir değerin sayısını verir.
- *min_element*: Dizideki en küçük elemanın pozisyonunu verir.
- *max_element*: Dizideki en büyük elemanın pozisyonunu verir.
- *equal*: İki dizinin eşit olup olmadığını bulur.
- *is_sorted*: Dizinin sıralı olup olmadığını bulur.
- *mismatch*: İki dizinin farklı olduğu ilk pozisyonu bulur.
- *transform_reduce*: Dizinin transformunun toplamını bulur.

Örneğin; GPU ve CPU arasında veri kopyalama işlemi şu şekilde gerçekleştirilir;

thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

Burada d_vec vektörü başlangıçtan sona kadar h_vec vektörüne kopyalanır. Burada d_vec GPU'da, h_vec CPU'dadır. GPU'dan bir vektör CPU'ya kopyalanmaktadır. CUDA kütüphanesinde bulunan cudaMemcpy kullanımına ihtiyaç duyulmadan CPU ve GPU'dan yer ayrılıp veri kopyalama sağlanmaktadır.

Thrust kütüphanesi kullanan örnek CUDA platformu kodu aşağıdaki gibidir;

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/sequence.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/replace.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {  // 10 elemanlı üç vektör için yer ayır
thrust::device_vector<int>X(10);
thrust::device_vector<int>Y(10);
thrust::device_vector<int>Z(10);
thrust::sequence(X.begin(), X.end());//  X’e değer ver 0,1,2,3, ....
thrust::transform(X.begin(), X.end(), Y.begin(), thrust::negate<int>()); //  Y = -X
thrust::fill(Z.begin(), Z.end(), 2); //  Z’nin tüm değerlerini 2 yap
//  Y = X mod 2
thrust::transform(X.begin(), X.end(), Z.begin(), Y.begin(), thrust::modulus<int>());
thrust::replace(Y.begin(), Y.end(), 1, 10); ///Y’deki tüm birleri onla değiştir
//  Y’yi yazdır
thrust::copy(Y.begin(), Y.end(), std::ostream_iterator<int>(std::cout, "\n"));
return 0;
}
```

NVIDIA Performans Temelleri (NPP) kütüphanesi

NPP (NVIDIA Performance Primitives) GPU hızlandırmalı görüntü sinyal ve video işleme fonksiyonlarını içeren bir kütüphanedir. Aritmetik ve lojik operasyonlar, istatistiksel operasyonlar ve filtre fonksiyonlarını içermektedir. CPU uygulamalarına göre 5 ile 10 kat arasında kazanç sağlamaktadır. GPU üzerinde çalışan 4000 civarında işlem içermektedir [58].

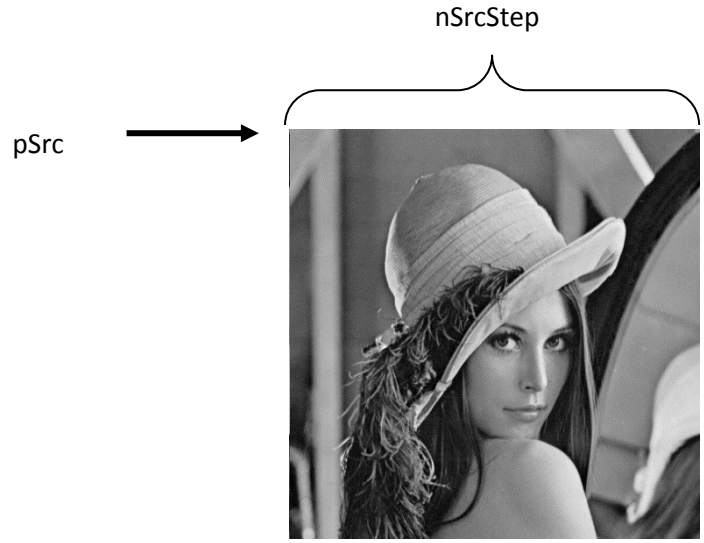
Bir görüntü üzerindeki işlemler için şu adımlar takip edilmelidir;

- Görüntünün CPU'ya yüklenmesi
- Görüntünün GPU"ya kopyalanması
- NPP fonksiyonları ile işlemlerin gerçekleştirilmesi
- Sonucun CPU'ya transfer edilmesi
- Sonucun alınması

NPP görüntü gösterimi

NPP'de görüntü pSrc ve nSrcStep olmak üzere iki parametre ile gösterilmektedir (Bkz. Şekil 5.21);

- pSrc: Görüntünün ilk pikselinin işaretçisi
- nSrcStep : Görüntünün bir satırının kaç bayt yer kapladığı



Şekil 5.21. NPP'de görüntü parametreleri

Fonksiyon isimlendirme

Kullanılan fonksiyonlar aşağıdaki isimlendirme kuralına uymaktadırlar. Bu kural o fonksiyonun hangi özellikleri taşıdığını ve nerede ve hangi veriler için kullanılacağını gösterdiğinden önemlidir.

İsimlendirme kuralı: nppiMulC_8u_C1IRSfs (Fonksiyon adı olarak kullanılmaktadır)

- npp: Kütüphane adı
- i: Görüntü modülü (s: Sinyal modülü)
- MulC: Temel isim (Add, Sum, vb.)
- 8u: Görüntünün veri tipi (16u, 32s, 32f, 64f, etc.)
- C1: Tek kanal (C3R, C4R, AC4R, etc.)
- I: in-place (belirtilmemişse out-of-place)
- R: Çalışma alanı (ROI)
- Sfs: Sonucu büyütüp küçültmeye izin veren

6. SİSTEM TASARIMI, HATA BULMA ALGORİTMALARI GELİŞTİRME VE PARALELLEŞTİRME

Bu bölümde sistem tasarımı, ağız hatalarının bulunması algoritması, kapsül hataları, GPU ile paralelleştirilen yan yüzey hataları, kovan hatalarının OpenMP ile paralelleştirlmesi ve elde edilen sonuçlar verilmektedir.

6.1. Sistem Tasarımı

Kameralı kontrol sistemleri, insan gözü ile yapılan kalite kontrol işlemlerini kamera ve ilgili yazılımlar ile gerçekleştiren bilgisayarlı görme sistemleridir. Bir diğer deyişle bilgisayarlı görme, görsel kontrol ile gerçekleştirilen operasyonların kameralı sistemlerle çözümlenmesidir. Kameralı kontrol sistemleri kullanılarak çok çeşitli sektörlerde endüstriyel projeler gerçekleştirilmektedir. Otomotiv, robotik, paketleme, gıda, ilaç sağlık ve askeri sistemler bunlardan bazılarıdır. Bilgisayarlı görme ile tüm sektörlerde eksik ve hatalı parça kontrolü, boyut ölçme, karakter ve desen tanıma, robot kol yönlendirme, yüzey analizi, renk analizi, ürün sayma, ürün hata tespiti, montaj, baskı kontrolü, barkod okuma, kare kod okuma gibi pek çok işlem yapılabilmektedir.

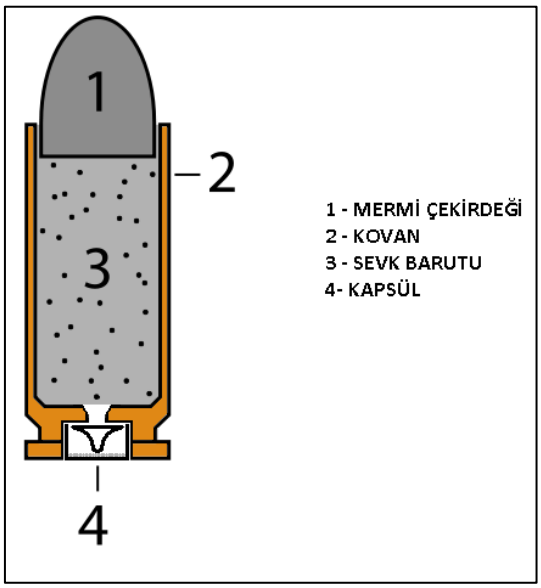
Bu tez çalışmasında kovan üretim aşamasında, hatalı kovanların ayrılmasını sağlayan bir otomasyon sistemi gerçekleştirilmektedir. Bunun için, MKE Gazi Fişek Fabrikasında en fazla üretimi yapılan 7.62 mm çaplı fişeklerin (Bkz. Şekil 6.1) üretim aşamalarından birisi olan kovan muayene adımında hatalı kovanları üretim bandından çıkaran otomatik bir makine tasarımı ve üretimi yapılmaktadır. Sistem gerçek zamanlı olarak dakikada 300 kovanın muayenesini yapacak şekilde tasarlanmaktadır. Bu çalışma 0018.STZ.2013-1 numaralı SANTEZ Projesi ile desteklenmektedir. Bu tez çalışmasının kapsamı ise ağız hatalarının bulunması algoritmalarının geliştirilmesi ile ağız hataları, kapsül hataları ve yan yüzey hataları algoritmalarının OpenMP ile paralelleştirilmesi ve yan yüzey hatalarının GPU ile paralelleştirilmesidir.



Şekil 6.1. 7.62X51 mm fişek

Fişek dört bölümden oluşmaktadır (Bkz. Şekil 6.2). Bunlar;

- Mermi çekirdeği,
- Kovan,
- Sevk barutu,
- Kapsül'dür.



Şekil 6.2. Fişeğin bölümleri [1]

Fişek üretimi sırasında çeşitli aşamalarda kalite kontrol işlemi yapılmaktadır. Bunlardan biri de kovan üretimi sırasında yapılan kontroldür. Kovan üretimi aşamasında hatalı kovanların üretim bandından çıkarılması ile hatalı kovanların diğer aşamalara geçmeleri önlenmekte ve bu sayede maliyet düşürülmektedir. MKE Gazi Fişek Fabrikası'nda kovanların hata kontrolü gözle muayene (Bkz. Resim 6.1) ile yapılmaktadır.



Resim 6.1. Gözle muayene tezgahları

Operatör işçiler ile yapılan gözle muayenede çeşitli dezavantajlar ortaya çıkmaktadır. Bunlar;

- Düşük kontrol hızı (125 kovan/dakika),
- Hata oranlarının yüksekliği,
- Üretim maliyeti (işçilik maliyeti),
- İnsan faktörü (yorgunluk, hastalık, vb.),
- İstatiksel bilgilerin tutulmamasıdır.

Bu dezavantajları ortadan kaldırmak amacıyla bu tez çalışmasında hatalı kovanları tespit edip ayıran otomatik bir sistem gerçekleştirilmektedir.

Kovan hata çeşitleri NATO standartlarıyla (MIL-STD-636 ve MIL-STD-651) [3, 4] belirtilmektedir. Bu hatalar;

- Şapka hatası
- Ezik
- Boyuna çatlak
- Delik kovan
- Sürekli hadde çizgisi
- Enine, boyuna, helisel küçük çizgi
- Maden hatası
- Markalama hatası
- Kovan ağzında ezilme
- Ağız tavlama hatası
- Şapka çevresinde hata (bozukluk)
- Şapkada çevresel düzgünsüzlük hataları
- Ağız bölgesinde çatlak ve ezikler
- Şapka çevresi ve tırnak bölgesi hatası
- Boyun ve ağız bölgesinde şişkinlikler
- Kovan ağzında sıkma hatası
- Boyun bölgesinde eziklik
- Çökme boyuna derin çizikler

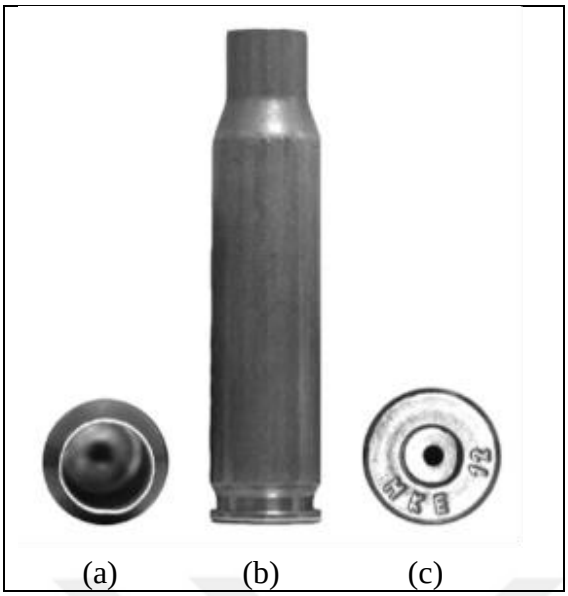
Kovan hataları üç ana kısımda incelenmektedir. Bunlar;

a) Ağız hataları,

b) Gövde hataları,

c) Kapsül hataları'dır.

Şekil 6.3'te kovanın bölümleri gösterilmektedir.



(a) (b) (c)

Şekil 6.3. Kovanın bölümleri: (a) ağız (b) gövde (yan yüzey) (c) kapsül

Kovan hataları aşağıda başlıklar halinde özetlenmiştir;

a) Ağız Hataları;

- Ağız kısmındaki daireselliğin düzgün olmaması,
- Ağız kısmında materyal eksikliği,
- Çapak vb. fazlalık olmasıdır.

b) Kapsül Hataları;

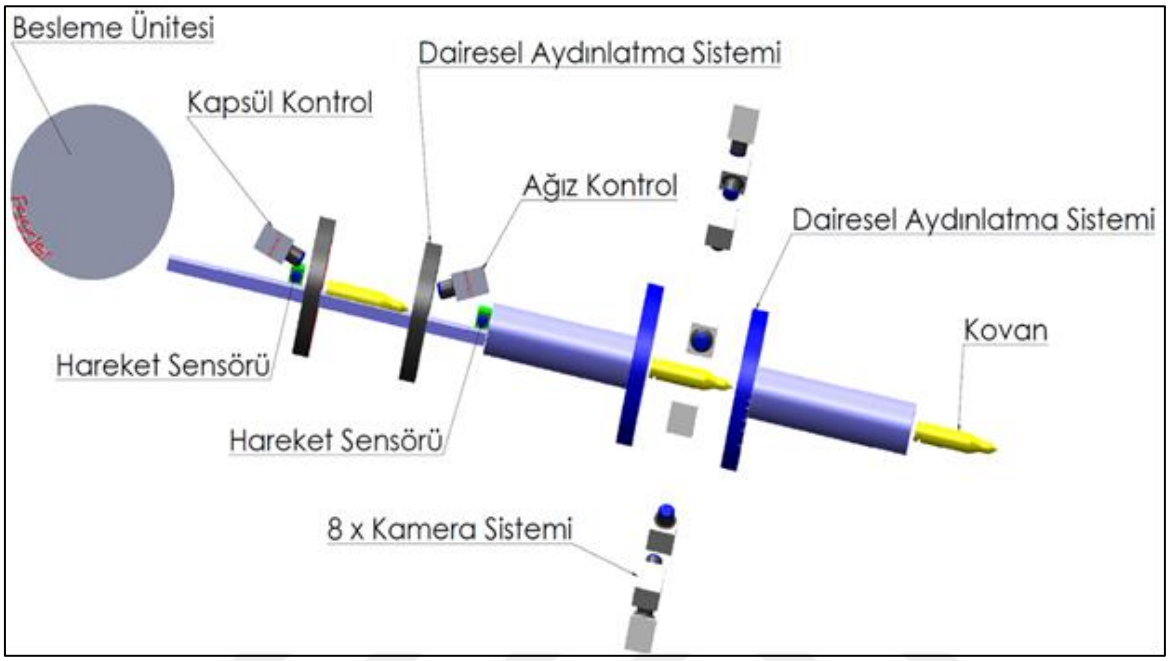
- Kapsül kısmında bozulmalar,
- Alev deliğinin varlığı,
- Yazıların kontrolü.

c) Gövde Hataları;

- Enine boyuna çizikler,
- Ezik, delik kovan,
- Şişkinlik veya çökme,
- Maden hatası,
- Tavlama hatası.

Bu hatalar incelendiğinde her bir bölüm için farklı hata tipleri olduğu görülmektedir. Bunların her birinin çözümü için de farklı algoritmalar ve yaklaşımlar uygulanmıştır.

Geliştirilen sistem ile kovanın ağız, kapsül ve yan yüzey kısımlarındaki hataların kontrolünün otomatik olarak yapılması hedeflenmiştir. Bu hedefe ulaşmak için tasarlanan mekanik sistemin şematik gösterimi Şekil 6.4'te verilmektedir.



Şekil 6.4. Geliştirilen sistemin şematik gösterimi

Sistemin mekanik kısmı üç üniteden oluşmaktadır;

a) Kovan besleme ünitesi,

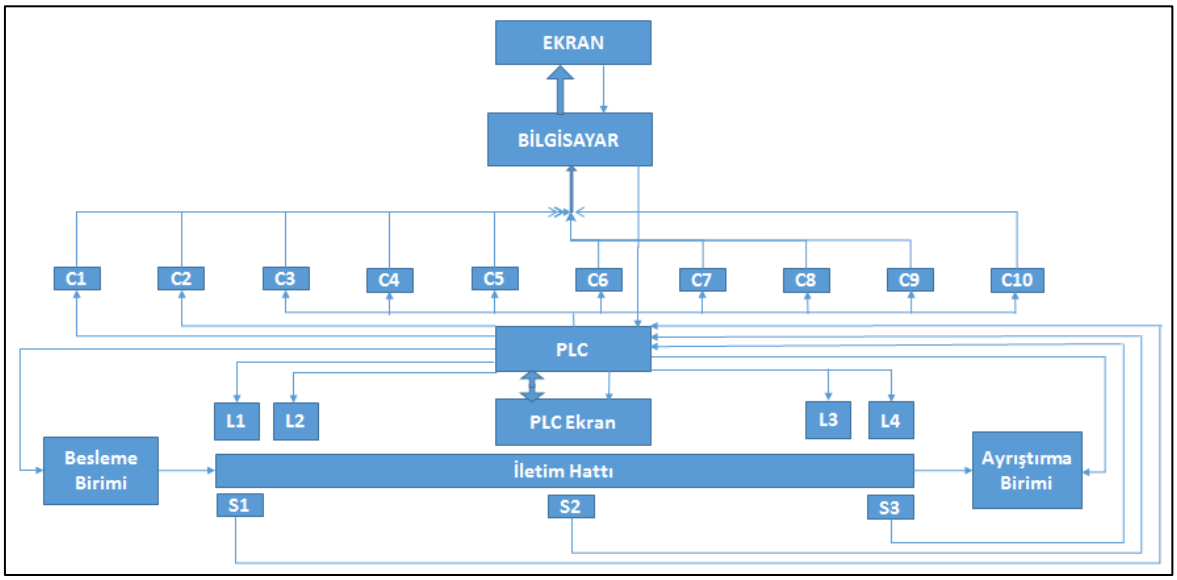
b) Kovan aydınlatma ve görüntü alma ünitesi,

c) Hatalı kovan ayrıştırma ünitesi.

Kovanlar ilk olarak besleme ünitesinden geçerek aydınlatma ve görüntü alma ünitesine gelmektedirler. Burada önce ağız ve kapsül görüntüleri alınmakta daha sonra da yan yüzey görüntüleri alınmaktadır. Alınan tüm görüntüler işlenerek bir kovanın tüm görüntülerinde hatasız sınıflandırması yapıldıysa ayrıştırma ünitesinde hatasız bölümüne gelmektedirler Hatalı olan kovanlar ise ayrıştırma bölümünde ayrılmaktadırlar. Bu nedenle kovan ayrıştırma birimine gelene kadar kovanla ilgili olarak hatalı/hatasız bilgisinin belirlenmesi ve kovan ayrıştırma ünitesine yollanması gerekmektedir.

Kovan hata ayıklama algoritmaları yan yüzey, kapsül ve ağız olmak üzere üç bölümden oluşmaktadır. Geliştirilen algoritmalar gerçek zamanlı sistem üzerinde çalıştırılmıştır. Uzun

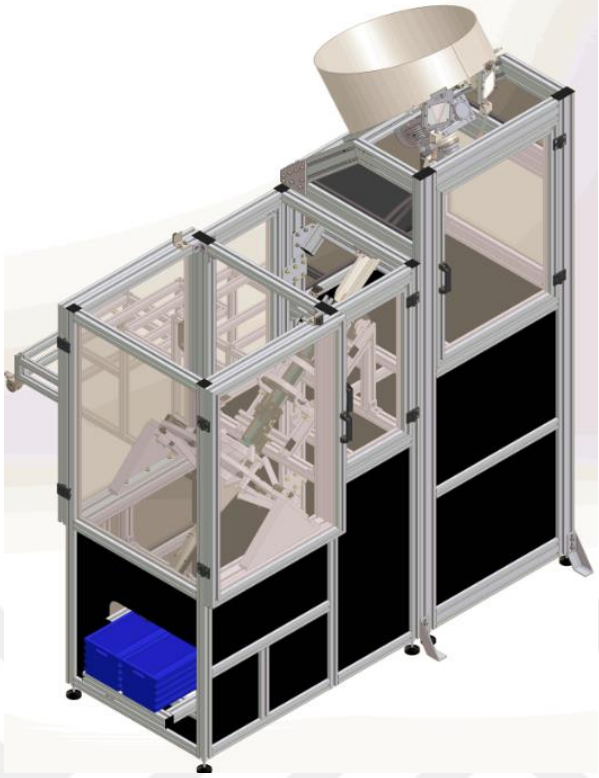
bir süreçte doğruluk oranlarının arttırılması amacıyla hem mekanik hem de yazılımsal olarak iyileştirmeler yapılmıştır. Algoritmaların geliştirme sürecinde pek çok farklı yöntem test edilmiş ve doğruluk oranı en yüksek olanlar sistemde çalışır hale getirilmiştir. Geliştirilen sisteme ait blok diyagramı Şekil 6.5'te verilmektedir. C1, C2, …, C10 kameraları, L1, L2, L3 ve L4 aydınlatmaları, S1, S2 ve S3 sensörleri göstermektedir. Geliştirilen sistemin görüntüsü ise Resim 6.2'de verilmektedir.



Şekil 6.5. Geliştirilen sisteme ait blok diyagramı

Her bir kovan için ağız kapsül ve yan yüzey olmak üzere görüntüler alınmakta ve görüntü işleme teknikleri ile hatalı kovanlar tespit edilmektedir. Ağız ve kapsül görüntüsü birer adet, kovan yan yüzey görüntüsü ise 8 adet olmak üzere her bir kovana ait 10 adet görüntü alınmakta ve işlenmektedir. İki kovanın geçişi arasındaki süre, dakikada 300 kovan muayenesi hedefini tutturmak için en fazla 200 ms olmalıdır. Bu süre içerisinde sensörlerin kovanı algılaması, tetik vermesi, kameradan görüntülerin alınması, görüntülerin işlenmesi, sonucun PLC'ye gönderilmesi ve ayrıştırma birimine tetik vermesi ve kovanın ayrılması bulunmaktadır. Geliştirilen sistemin görüntüsü Resim 6.2'de görülmektedir.

Kovan hataları ağız, kapsül ve yan yüzey olmak üzere üç farklı algoritma kullanılarak bulunmaktadır. Ağız ve kapsül için birer görüntü, yan yüzey için ise 8 adet görüntü olmak üzere her bir kovan için 10 adet görüntü işlenmektedir.



(a)



(b)

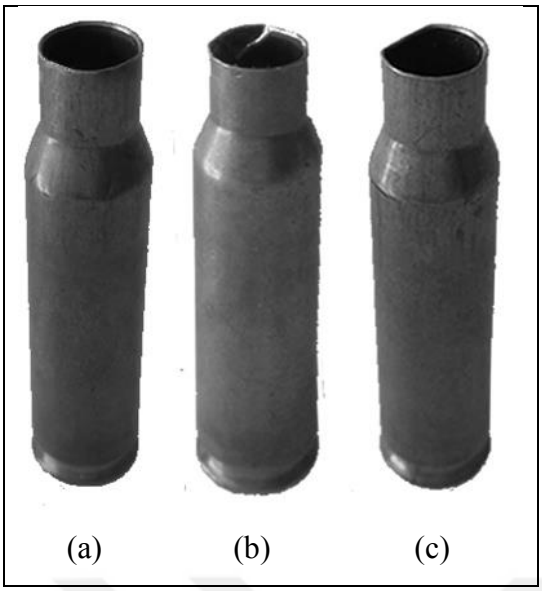
Resim 6.2. Geliştirilen sistemin görüntüsü (a) prototip teknik çizimi (b) çalışan sistemin fotoğrafı

Gerçekleştirilen sistemde kovan hatalarını tespit için, ağız ve kapsül görüntüleri CPU çekirdeklerini kullanarak, yan yüzey görüntülerini ise GPU kullanarak paralelleştiren hibrit bir yapı kullanılmıştır, İşlemci çekirdeklerinin paralelleştirmesinde OpenMP, GPU paralelleştirmesinde ise CUDA platformu kullanılmıştır. Gerçekleştirilen hibrit sistem CPU işlemci çekirdeklerinden birincisinde ağız hataları algoritması, ikincisinde kapsül hataları algoritması, üçünsünde ise yan yüzey hataları algoritması çalıştırılacak şekilde tasarlanmıştır. Sekiz adet görüntüden oluşan yan yüzey hataları algoritması aynı zamanda CUDA kullanılarak GPU çekirdeklerinde paralelleştirimiştir.

6.2. Kovan Ağız Hatalarının Bulunması

Görüntü işleme, görüntüleri işlemek, mevcut resim ve grafikleri değiştirmek ya da iyileştirmek için kullanılır. En yaygın kullanımı, görüntülerin incelenmesi, görüntülerdeki nesnelerin özelliklerinin belirlenmesi ve ayrıştırılmasıdır. Biçimleri çevreleyen konturları işleyerek geometrik ve istatistiksel bulgular elde etmek nesne tespit ve tanıma çalışmalarında yaygın olarak kullanılan yöntemlerdir.

Bu tez çalışmasında da dairesel yapılar tespit edilmekte ve tam daire olup olmadığına bakılmaktadır. Kovan hatalarından biri olan ağız hatalarından ezilme, ağız bölgesinde çatlak ve yarıkların (Bkz. Şekil 6.6) tespit edilmesi için bir algoritma geliştirilmiş ve kovanın hatalı olup olmadığına karar verilmiştir. Bu yazılımın çalışma prensibi ise kovanın ağız kısmındaki yuvarlağın elde edilmesi ve bu yuvarlağın ne kadar düzgün bir daire olduğuna bakılarak kovanın hatalı ya da hatasız olmasına karar verilmesidir.



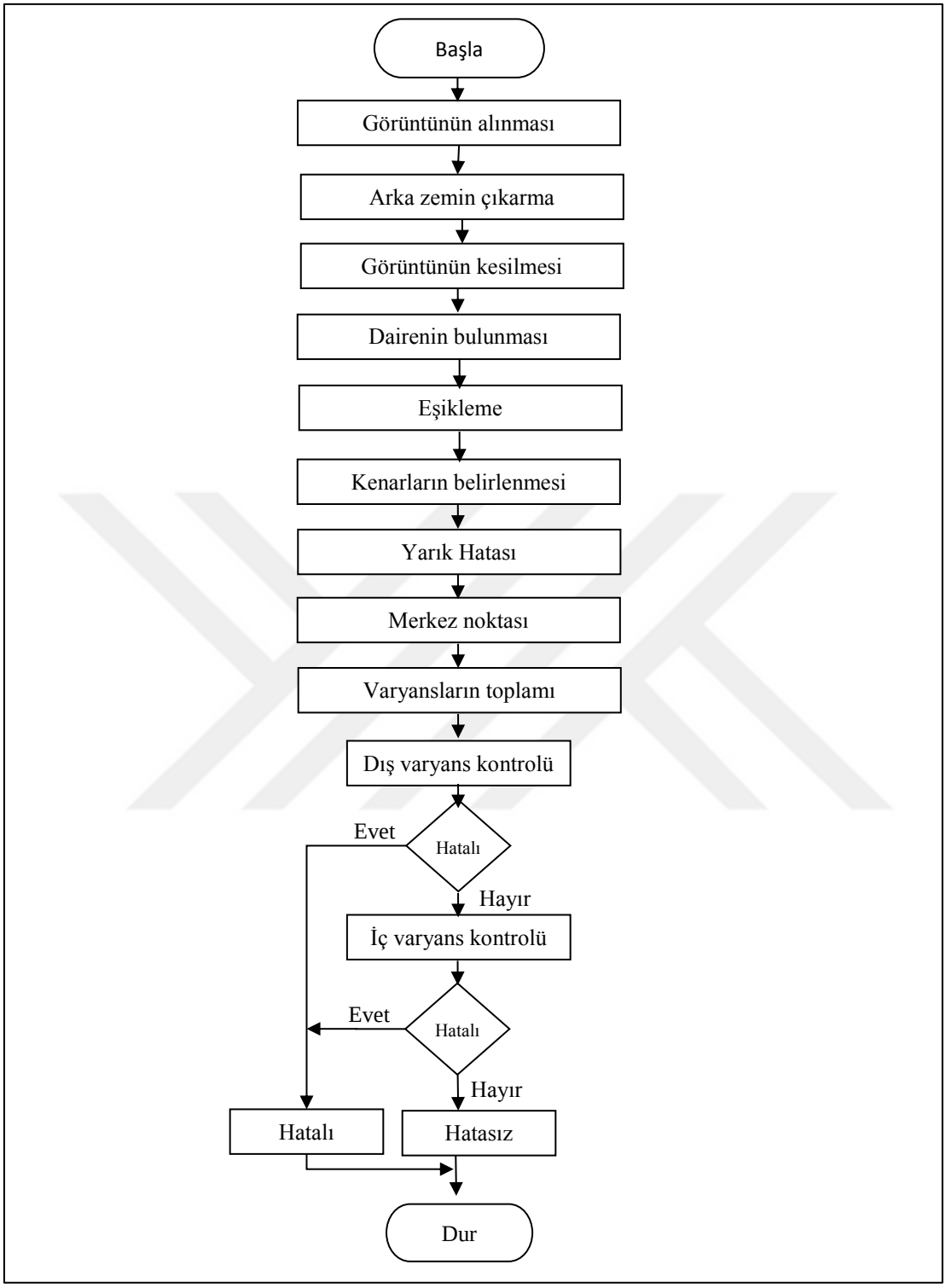
Şekil 6.6. Kovan ağız görüntüleri (a) hatasız (b) yarık hatası (c) ezik

Algoritma elde edilen görüntü üzerinde bazı işlem adımlarının uygulanma sonucu elde edilen verilere göre kovanın hatalı veya hatasız olduğuna karar vermektedir. Algoritmanın doğruluk derecesini artırmak amacıyla algoritmada pek çok kez düzenlemeler yapılarak son haline getirilmiştir. Elde edilen algoritma gerçek sistem üzerinde test edilmiş ve elde edilen sonuçlar verilmiştir.

Ağız hatalarının bulunması için kullanılan algoritmanın işlem basamakları aşağıdaki gibidir;

a) Arka zemin çıkarma
b) Görüntünün kesilmesi
c) Dairelerin bulunması
d) Eşikleme işlemi
e) Kenarların belirlenmesi
f) Yarık hatasının tespit edilmesi
g) Yeni görüntüde merkez noktasının tespit edilmesi
h) Varyansların toplamının hesaplanması
i) Sınıflandırma

Ağız hatalarını bulma algoritmasının blok şeması Şekil 6.7'de verilmektedir.



Şekil 6.7. Ağız hatası bulma algoritması blok diyagramı

Gerçekleştirilen algoritma gerçek zamanlı sistem üzerinde alınan görüntüleri işleyerek elde edilen sınıflandırma sonuçlarına göre hatalı olanları ayırmaktadır. Kovan muayene sistemi

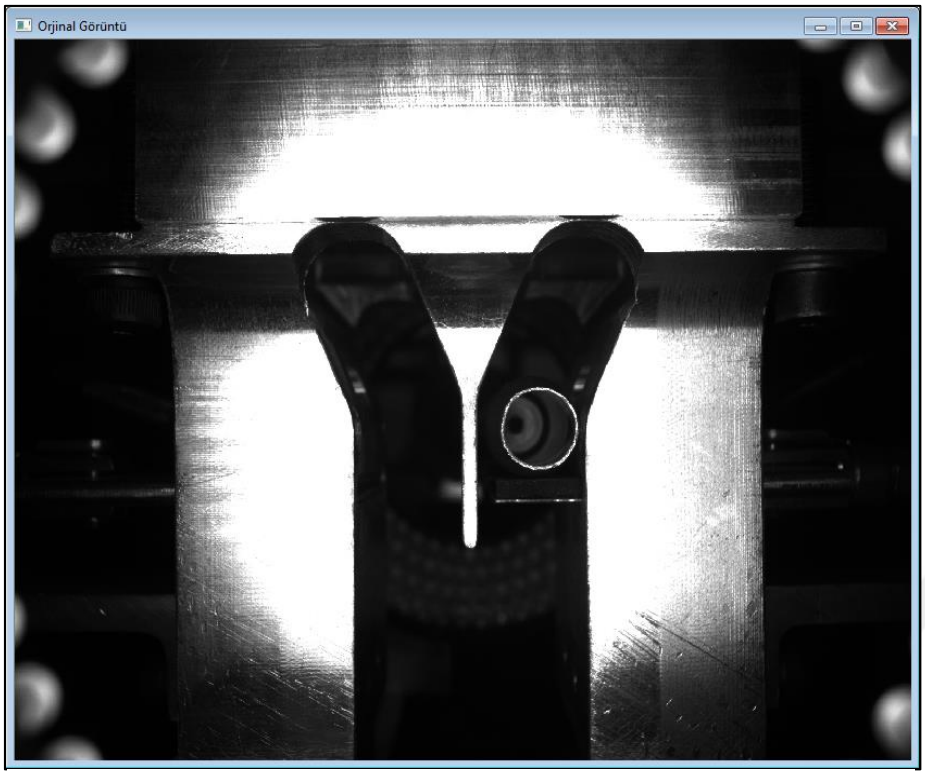
üzerinde bulunan bekleme istasyonundaki kovandan alınan ham görüntü örneği Şekil 6.8'de görülmektedir.



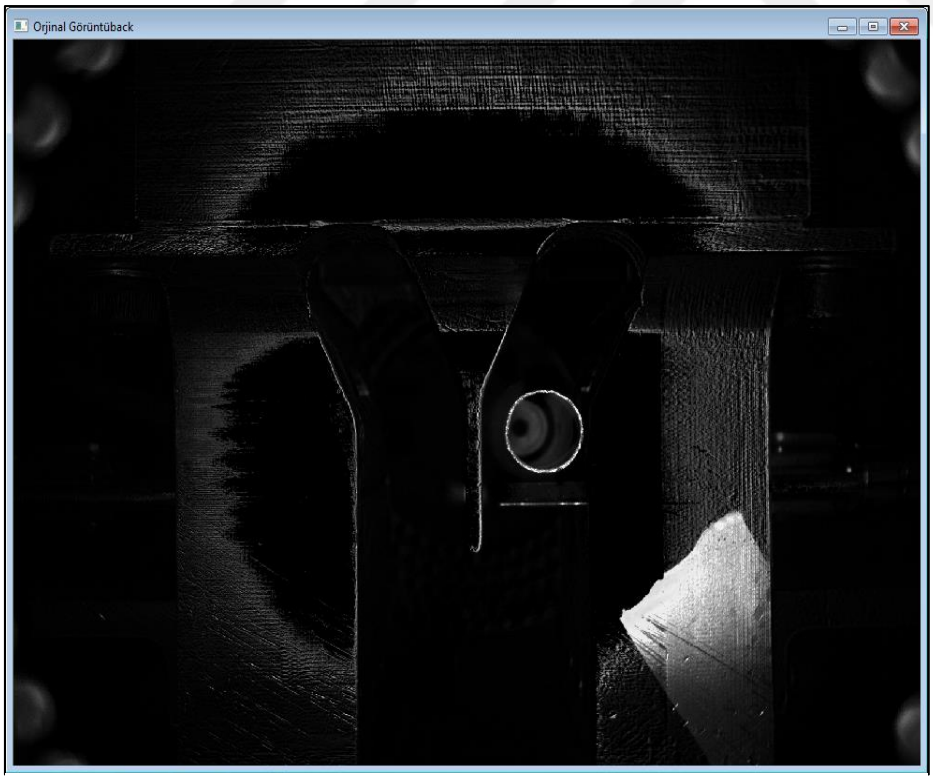
Şekil 6.8. Bekleme istasyonu ile alınan görüntü

6.2.1. Arka zemin çıkarma

Kameradan alınan görüntülerde kovan ağzı dışında makine parçalarında da parlama yüksek olmaktadır. Bu görüntülerde eşikleme sonunda kovan ağzının belirlenmesi gerekirken istenmeyen şekilde eşik değerinin yüksek çıkması sonucu kovan ağzı belirlenememektedir. Bu nedenle ilk adım olarak arka zemin çıkarma işlemi uygulanmaktadır. Elde edilen görüntüden kovansız olarak alınan görüntü arka zemin kabul edilerek çıkarılmaktadır. Şekil 6.9'da alınan görüntü ve ilk işlem olan arka zemin çıkarmadan sonra elde edilen görüntü verilmektedir.



(a)



(b)

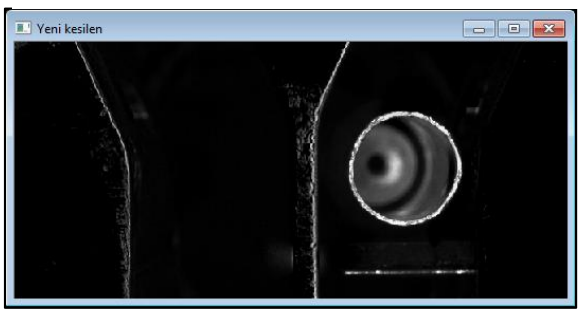
Şekil 6.9. Arka zemin çıkarma (a) kameradan alınan görüntü (b) işlem sonrası görüntü

6.2.2. Görüntünün kesilmesi

Elde edilen görüntüden 1600 X 1200 boyutlarındaki görüntüde işlem zamanını azaltmak amacıyla ilgi bölgesi (ROI) belirlenmiş ve görüntü 650 X 300 boyutlarında kesilerek kesilen görüntü üzerinde ilerleyen işlem adımları gerçekleştirilmiştir. Kovan ağız görüntüleri alınırken kaliteli görüntü alınması için gerekli ayarlar yapılmaktadır. Özellikle Gamma ve Gain değerleri en kaliteli görüntüyü almak için ayarlanmaktadır. Buna rağmen elde edilen görüntülerde gürültülerin oluşma ihtimaline karşı gürültüleri yok etmek amacıyla Gaussian filtresi uygulanmıştır. Görüntünün kesilmesi ve filtre sonucu Şekil 6.10'da verilmektedir.



(a)

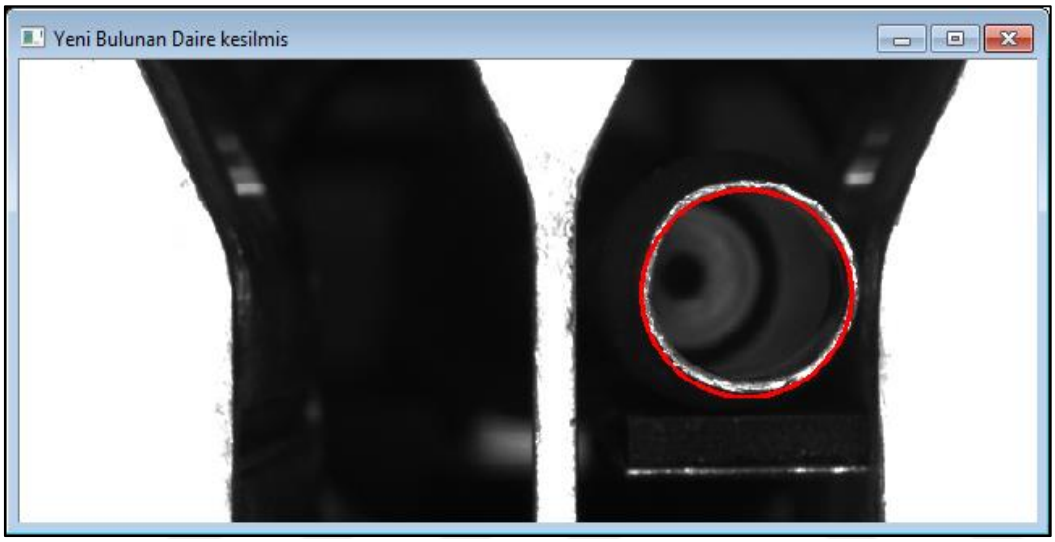


(b)

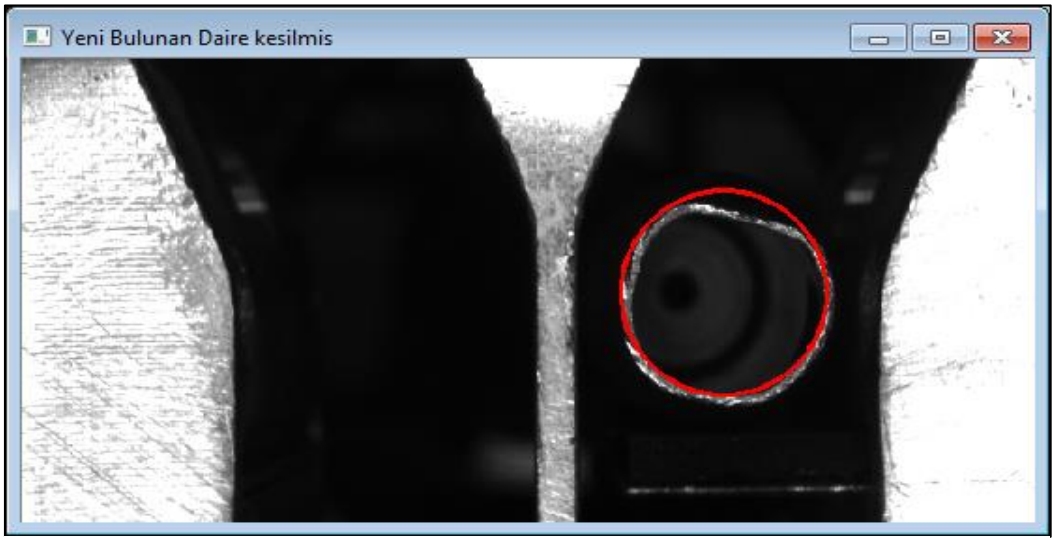
Şekil 6.10. Kesilen görüntü örneğinin önceki ve sonraki görüntüsü (a) 1600 X 1200 boyutlarındaki görüntü (b) 650 X 300 boyutlarındaki görüntü

6.2.3. Dairelerin bulunması

Daire bulma algoritması tam bir daire şekli olan nesneleri görüntü üzerinde işaretlemektedir. Çalışmada Hough daire bulma algoritması kullanılarak kovan ağzındaki daire bulunmaktadır. Bu algoritma sonucu elde edilen dairenin merkez noktası ve yarıçapı bulunmaktadır. Hough algoritması sonucu elde edilen daire Şekil 6.11'de gösterilmektedir.



(a)

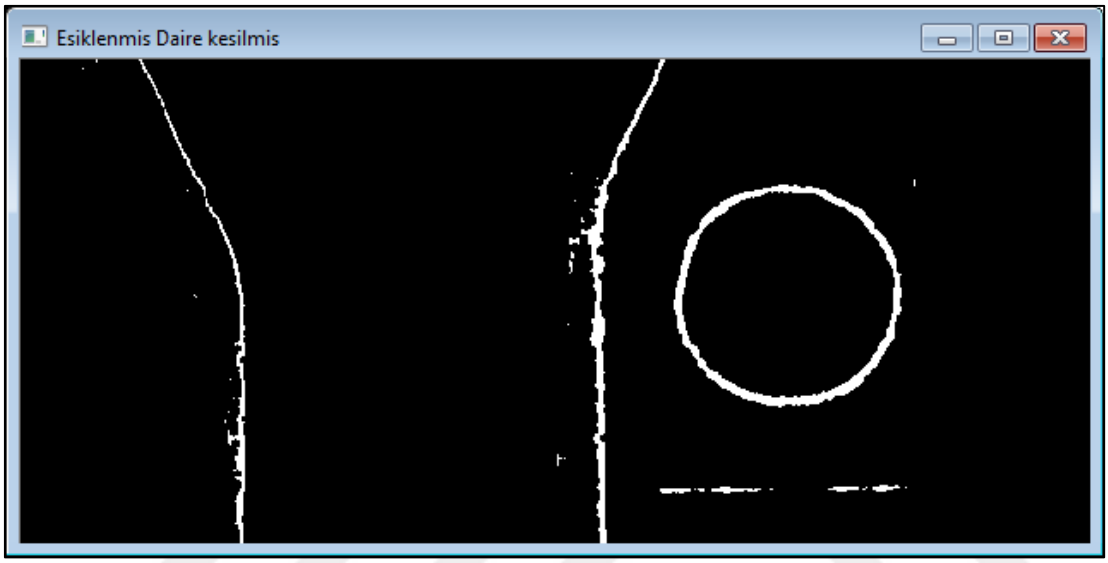


(b)

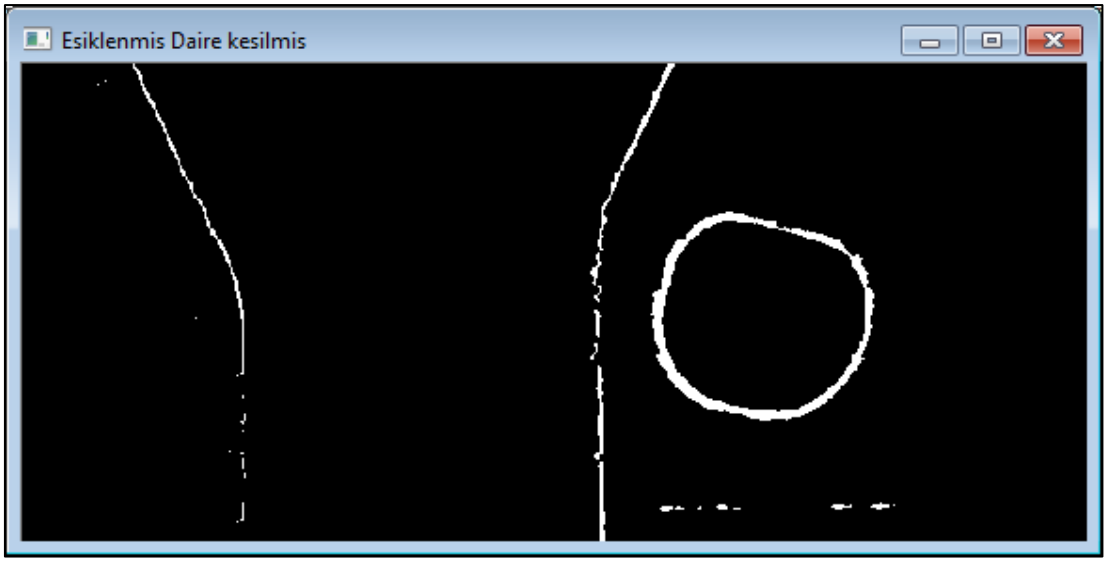
Şekil 6.11. Hough algoritması sonucu işaretlenmiş örnek ağız görüntüleri (a) hatasız kovan (b) hatalı kovan

6.2.4. Eşikleme işlemi

Kovan ağız görüntüsü üzerinde eşikleme (threshold) işlemi gerçekleştirilmiştir. Eşik değeri Otsu eşik hesaplama yöntemi ile belirlenmiştir. Eşikleme işleminin ardından morfoloji ile kapama (closing) işlemi gerçekleştirilmiş ve kovan ağzında küçük birleşmemiş alanlar kapatılmıştır. Eşiklenmiş ve morfoloji uygulanmış görüntü Şekil 6.12'de verilmektedir.



(a)



(b)

Şekil 6.12. Eşiklenmiş ve morfoloji uygulanmış örnek görüntüler (a) hatasız kovan (b) hatalı kovan

6.2.5. Kenarların belirlenmesi

Kovanın merkez noktasının koordinatları Hough daire bulma algoritması sonucunda elde edilmektedir. Bu koordinatlar alınarak eşiklenmiş görüntü üzerinde 360 derece taranarak merkez noktanın kovan kenarlarına olan uzaklıkları ve bu noktaların koordinatları belirlenmektedir. Eşiklenmiş görüntüde Şekil 6.12'de görüldüğü gibi kovanın içi siyah ve çevresinde beyaz bir halka olarak kenarı görülmektedir. Bu şekilde elde edilen eşiklenmiş görüntüde her derece için noktalar taranarak siyahtan beyaza geçilen yer aranmaktadır. Beyaz piksele rastlandığında bu noktanın koordinatları bir diziye yazılmaktadır. Ayrıca merkez noktasına olan uzaklığı diziye yazılmaktadır. 360 derece taranan kovanın iç çevresi belirlenmektedir (Bkz. Şekil 6.13).

Bu durum matematiksel olarak ifade edildiğinde; üçüncü adımda elde edilen dairenin merkez noktası $c(x, y)$, tarama açısı φ $(\varphi = 1,2,...,360)$, merkez noktasından olan uzaklığı i_φ $(i_\varphi = 1,2,..., I)$, kovan çeperinin koordinatlar $C(X_\varphi, Y_\varphi)$ olarak gösterilirse. X_φ ve Y_φ değerleri Eş. 6.2 ve Eş. 6.3 formülleriyle hesaplanır.

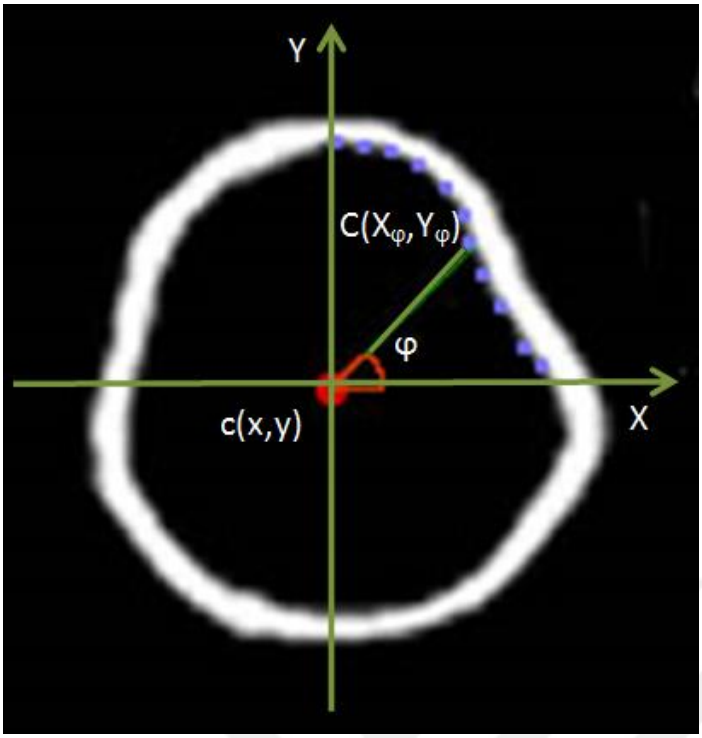
$$X_\varphi = x + (i_\varphi * \cos(\varphi * pi \div 180)) \tag{6.2}$$

$$Y_\varphi = y + (i_\varphi * \sin(\varphi * pi \div 180)) \tag{6.3}$$

Sonuç olarak $C(X_\varphi, Y_\varphi)$ ve i_φ değerleri diziye yazılarak sonraki adımlarda kullanılmaktadır.

Aynı şekilde kovanın dış çeperi de belirlenmekte ve koordinatları ve merkeze olan uzaklığı bir başka diziye yazılmaktadır. Önceki algoritmadan farklı olarak bundan sonraki işlem basamakları hem kovanın iç çeperi hem de dış çeperi için gerçekleştirilmektedir.

Kovan iç çeperi koordinatları Cd, kovan dış çeperi koordinatları Ci isimli dizide yazılmıştır. Örneğin; iç çeperde 0 dereceden 360 dereceye kadar i. dereceye karşılık gelen noktanın koordinatları Ci(i) olmaktadır.



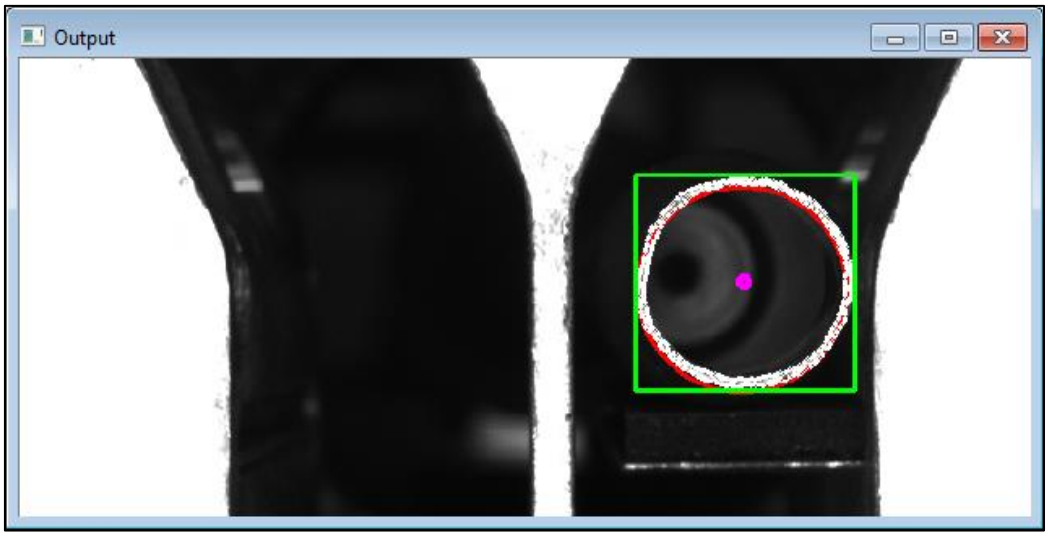
Şekil 6.13. Kovan ağzı iç çeperinin tespiti

6.2.6. Yarık hatasının tespit edilmesi

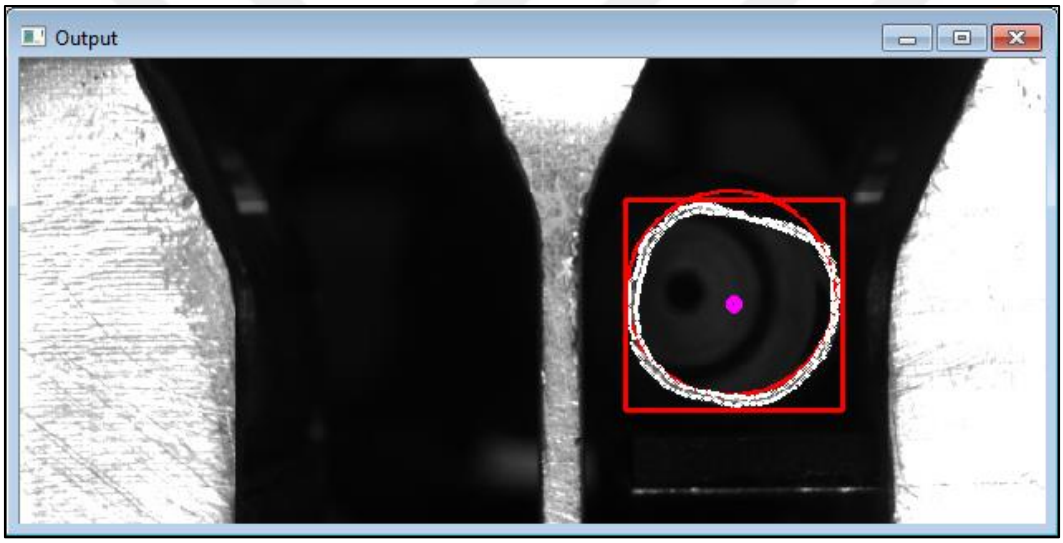
Kenarların belirlenmesi adımında 360 derece taranarak elde edilen noktaların koordinatları kullanılarak her bir noktanın yanındaki noktaya olan uzaklığına bakılmakta ve bu uzaklıklar belirlenmektedir. 360 derece için $C(X_\varphi, Y_\varphi)$ ve $C(X_\varphi + 1, Y_\varphi + 1)$ noktaları arasındaki Öklit uzaklığı hesaplanır. Görüntüde bir kesinti yoksa yan yana olacak olan noktaların arasındaki uzaklık küçük hatta 1 piksel ile 2 piksel arasında olacaktır. Eğer bir kesinti varsa bu değer fazla olacaktır. Bu nedenle arada belirlenen değerden fazla uzaklık olması durumunda kovanda yarık hatası belirlenmekte ve hatalı olarak işaretlenmektedir.

6.2.7. Yeni görüntüde merkez noktasının tespit edilmesi

Bu adımda elde edilen "*mükemmel daire*" karşılaştırma için kullanılmıştır. Elde edilen görüntüde minimum dikdörtgen çizilerek dairenin tespit edilmesi elde edilen en iyi sonucu vermektedir. Çizilen minimum dikdörtgenin ağırlık merkezi elde edilecek dairenin merkez noktası olmaktadır. Çapı ise kenarlarının ortalaması alınarak hesaplanmaktadır. Burada hesaplanan "*mükemmel daire*"nin merkez noktası *M* ile yarıçapı *R* ile gösterilebilir. Çizilen minimum dikdörtgen Şekil 6.14'te gösterilmektedir.



(a)

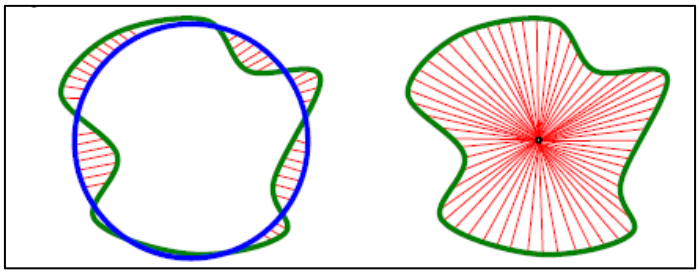


(b)

Şekil 6.14. Bulunan minimum dikdörtgenin görüntü üzerinde işaretlenmesi (a) hatasız kovan (b) hatalı kovan

6.2.8. Varyansların toplamının hesaplanması

Varyans, özellikle dairesel nesnelerin tespit edilmesinde ve sınıflandırılmasında kullanılmaktadır. Gerçekleştirilen algoritmada önceki adımda elde edilen dairenin merkez noktası kullanılarak kovan kenarlarının merkez noktasına olan uzaklıkları bulunmaktadır. Varyans hesaplanırken minimum dikdörtgen ile elde edilen merkez noktası ve hesaplanan ortalama yarıçap kullanılmaktadır. Bu daire ile elde edilen kovan iç ve dış çeperinin varyanslar toplamı hesaplanmaktadır (Bkz. Şekil 6.15).



Şekil 6.15. Dairesel varyansların hesaplanması

Herhangi bir açı için kovan çeperinin "*mükemmel daire*"nin kenarına olan uzaklığı U ile gösterilirse iç ve dış varyansların toplamı Eş. 6.4 ve Eş. 6.5 formülleriyle hesaplanır.

$$Var_{iç} = \sum_{\theta=0}^{\theta=359} \left(Eğer\ U_{iç}(\theta) > 2\ ise\ U_{iç}(\theta) \right)^2 \quad (6.4)$$

$$Var_{dış} = \sum_{\theta=0}^{\theta=359} \left(Eğer\ U_{dış}(\theta) > 2\ ise\ U_{dış}(\theta) \right)^2 \quad (6.5)$$

Uzaklıklar ise iç çeper için Eş. 6.6 ve dış çeper için Eş. 6.7 formülleriyle hesaplanır.

$$U_{iç}(\theta) = \sqrt{(Ci(\theta)(x) - M(x)R\cos\theta)^2 + (Ci(\theta)(y) - M(y)R\sin\theta)^2} \quad (6.6)$$

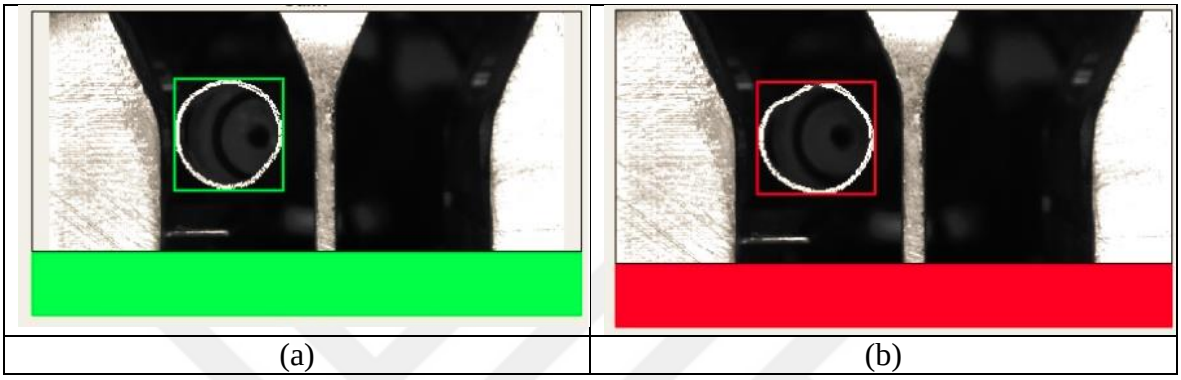
$$U_{dış}(\theta) = \sqrt{(Cd(\theta)(x) - M(x)R\cos\theta)^2 + (Cd(\theta)(y) - M(y)R\sin\theta)^2} \quad (6.7)$$

6.2.9. Sınıflandırma

Sağlam kovanların ağzı "*mükemmel daire*"ye yakın olduğundan varyanslar toplamı küçük olacaktır ve kovan sağlam olarak belirlenecektir. Kovan ağzında herhangi bir ezik, yamukluk veya bozulma olduğunda varyansların toplam değeri büyük olmakta ve kovan hatalı olarak işaretlenmektedir. Kovandaki ağız hatası oranı arttıkça varyanslar toplamı büyüyecektir. Buna göre varyansların toplamı belli bir değeri aşan kovanlar hatalı diğerleri hatasız olarak belirlenmektedir. Sınıflandırmada kovan iç çeperi için ve kovan dış çeperi için ayrı ayrı varyanslar toplamı hesaplanmaktadır. Burada iç ve dış varyanslar toplamından herhangi biri belirlenen orandan düşükse kovan hatasız olarak sınıflandırılmaktadır. Bunun dışındaki durumlarda kovan hatalı olarak sınıflandırılmaktadır. Belirlenen varyans değeri Δ ile gösterilirse sınıflandırma Eş. 6.8'e göre gerçekleştirilir.

$$Sınıf = \begin{cases} eğer\ Var_{iç} < \Delta\ \ veya\ Var_{dış} < \Delta,\ \ hatasız \\ değilse,\ \ hatalı \end{cases} \quad (6.8)$$

Geliştirilen arayüz yazılımının ağız hataları ile ilgili ekran görüntüleri Şekil 6.16'da görülmektedir.



(a) (b)

Şekil 6.16. Geliştirilen arayüz yazılımında ağız hataları (a) hatasız kovan (b) hatalı kovan

6.2.10. Performans ölçütleri

Algoritmanın performansını değerlendirirken kullanılan temel kavramlar doğruluk oranı, hata oranı, kesinlik, duyarlılık ve F-ölçütüdür. Algoritmanın başarısı, doğru sınıfa atanan örnek sayısı ve yanlış sınıfa atılan örnek sayısı nicelikleriyle alakalıdır. Test sonucunda ulaşılan sonuçların performans bilgileri karışıklık matrisi ile ifade edilebilir (Bkz. Çizelge 6.1). Karışıklık matrisinde satırlar test kümesindeki örneklere ait gerçek sayıları, kolonlar ise program çıktılarını ifade eder. Performans değerlendirme ölçütlerinden biri karışıklık matrisi şu şekilde oluşturulur:

Çizelge 6.1. Karışıklık matrisi

	Kovanların Durumu		
Programın Sonucu		Hatalı	Hatasız
	Hatalı	TP (Doğru Pozitif)	FP (Yanlış Pozitif)
	Hatasız	FN (Yanlış Negatif)	TN (Doğru Negatif)

Doğruluk ve Hata Oranı

Performansın ölçülmesinde kullanılan en popüler ve basit yöntem doğruluk oranıdır. Doğru sınıflandırılmış örnek sayısının toplam örnek sayısına oranıdır. Hata oranı ise yanlış sınıflandırılmış örnek sayısının toplam örnek sayısına oranıdır.

$$Doğruluk = \frac{TP + TN}{TP + FP + FN + TN} \tag{6.9}$$

$$Hata\ Oranı = \frac{FP+FN}{TP+FP+FN+TN} \tag{6.10}$$

Keskinlik ve Duyarlılık

Keskinlik sınıfı hatalı olarak bulunan doğru örnek sayısının hatalı olarak bulunan tüm örnek sayısına oranıdır. Duyarlılık ise doğru sınıflandırılmış hatalı örnek sayısının toplam hatalı örnek sayısına oranıdır.

Karışıklık matrisinden elde edilen veriler kullanılarak performans ölçütlerinden keskinlik ve duyarlılık Eş. 6.11 ve Eş. 6.12 formülüyle hesaplanmaktadır:

$$Keskinlik = \frac{TP}{TP + FP} \tag{6.11}$$

$$Duyarlılık = \frac{TP}{TP + FN} \tag{6.12}$$

F Ölçütü

Keskinlik ve duyarlılık ölçütleri tek başlarına bir sonuç çıkartmaya yeterli değildir. Her iki ölçüt birlikte değerlendirilerek bunların harmonik ortalaması ile F ölçütü hesaplanmaktadır. Elde edilen değer 0 ile 1 arasındadır. F ölçütü 1'e ne kara yakınsa sınıflandırıcı o kadar başarılıdır. F ölçütü aşağıdaki formülle elde edilmektedir:

$$F - \text{Ölçütü} = \frac{2 \times Duyarlılık \times Keskinlik}{Duyarlılık + Keskinlik} \tag{6.13}$$

6.2.11. Değerlendirme

Kovan ağzı hatalarını tespit etmek için geliştirilen algoritma görüntüler üzerinde test edilmiş ve en yüksek doğruluk oranını verecek şekilde geliştirilmiştir. Makinadan gerçek zamanlı olarak alınan görüntüler test edilmiş ve sonuçlar Çizelge 6.2'de verilmiştir.

Geliştirilen algoritma 500 adet kovan üzerinde uygulanmıştır. Bu kovanlardan 250 adedi hatasız, 250 adedi de hatalı olarak seçilmiştir. Kullanılan görüntüler çalışan bant üzerinde otomatik olarak alınan görüntülerdir. Geliştirilen algoritma 500 adet kovan üzerinde çalıştırılarak elde edilen sonuçlar değerlendirilmiştir.

Çizelge 6.2. Karışıklık matrisi sonuçları

	Kovanların Durumu		
Programın Sonucu		Hatalı	Hatasız
	Hatalı	239	11
	Hatasız	5	245

Elde edilen karışıklık matrisine göre doğruluk, hata oranı, keskinlik, duyarlılık ve F ölçütü değerleri aşağıdaki gibi bulunmuştur.

$$Doğruluk = \frac{TP + TN}{TP + FP + FN + TN} = \frac{239 + 245}{500} = 0.968$$

$$Hata\ Oranı = \frac{FP + FN}{TP + FP + FN + TN} = \frac{16}{500} = 0.032$$

$$Keskinlik = \frac{TP}{TP + FP} = \frac{239}{250} = 0.956$$

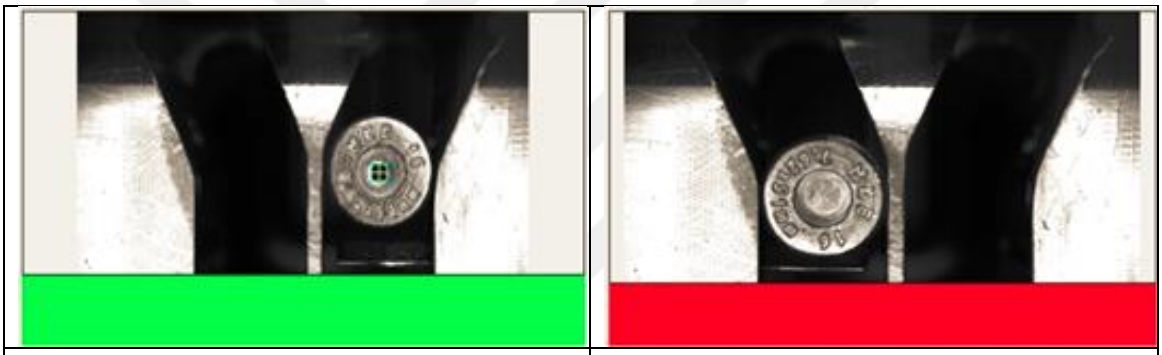
$$Duyarlılık = \frac{TP}{TP + FN} = \frac{239}{244} = 0.979$$

$$F - \text{Ölçütü} = \frac{2 \times Duyarlılık \times Keskinlik}{Duyarlılık + Keskinlik} = \frac{2 \times 0.979 \times 0.956}{0.979 + 0.956} = 0.967$$

Elde edilen sonuçlar kovan ağzı hatalarının yaklaşık olarak %97 doğruluk oranı ile tespit edildiğini göstermektedir. F ölçütü 0,967 olarak hesaplanarak yüksek bir değer elde edilmiştir. Oluşturulan algoritma bu sonuçlara göre başarılı bir şekilde çalışmaktadır.

6.3. Kapsül Hatalarının Bulunması

Kovan üzerinde oluşan hatalardan alev deliğinin bulunmaması kapsül görüntüsü üzerinden tespit edilmektedir. Alev deliğinin tespit edilmesi dört adımda gerçekleştirilmektedir. Bunlar görüntünün kesilmesi, eşikleme, Hough daire tespit algoritması ve sınıflandırmadır.
Geliştirilen arayüz yazılımının kapsül hataları ile ilgili ekran görüntüleri Şekil 6.17'de görülmektedir.



(a) (b)

Şekil 6.17. Geliştirilen arayüz yazılımında kapsül hataları (a) hatasız kovan (b) hatalı kovan

6.4. Yan Yüzey Hatalarının Bulunması

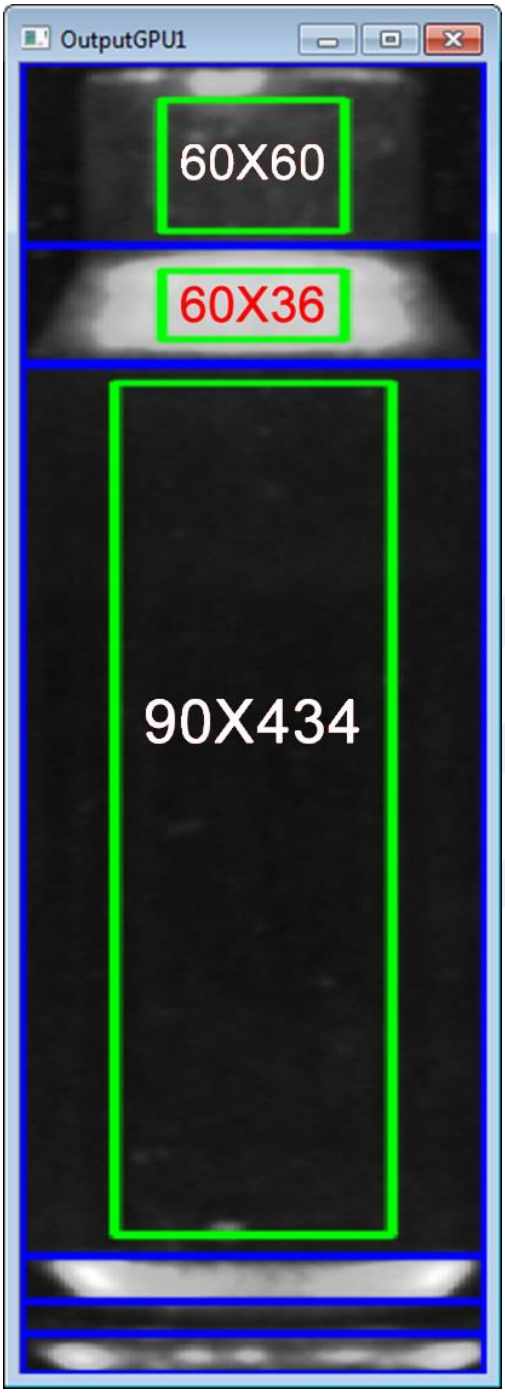
Yan yüzey hata bulma algoritması GPU ile paralelleştirilmekte ağız ve kapsül algoritmaları ise CPU çekirdeklerinde çalıştırılmaktadır. Yan yüzey hatalarının bulunmasında kullanılan standart algoritmalar Gaussian filtresi, şablon eşleme (template matching), eşikleme, Sobel filtresi, iki yanlı filtre (bilateral filtre) ve morfolojik operasyonlardır. Bu algoritmalar GPU ile paralelleştirilerek algoritmada hızlanma sağlanmıştır. Geliştirilen algoritma sistem üzerinden elde edilmiş gerçek görüntüler üzerinde çalıştırılmış ve elde edilen sonuçlar bu bölümde verilmiştir.

Yan yüzey algoritmasında ilk kullanılan algoritma Gaussian filtresidir. Arkasından şablon eşleme algoritması uygulanmaktadır. Gaussian filtresi ile gürültüler temizlemekte ardından

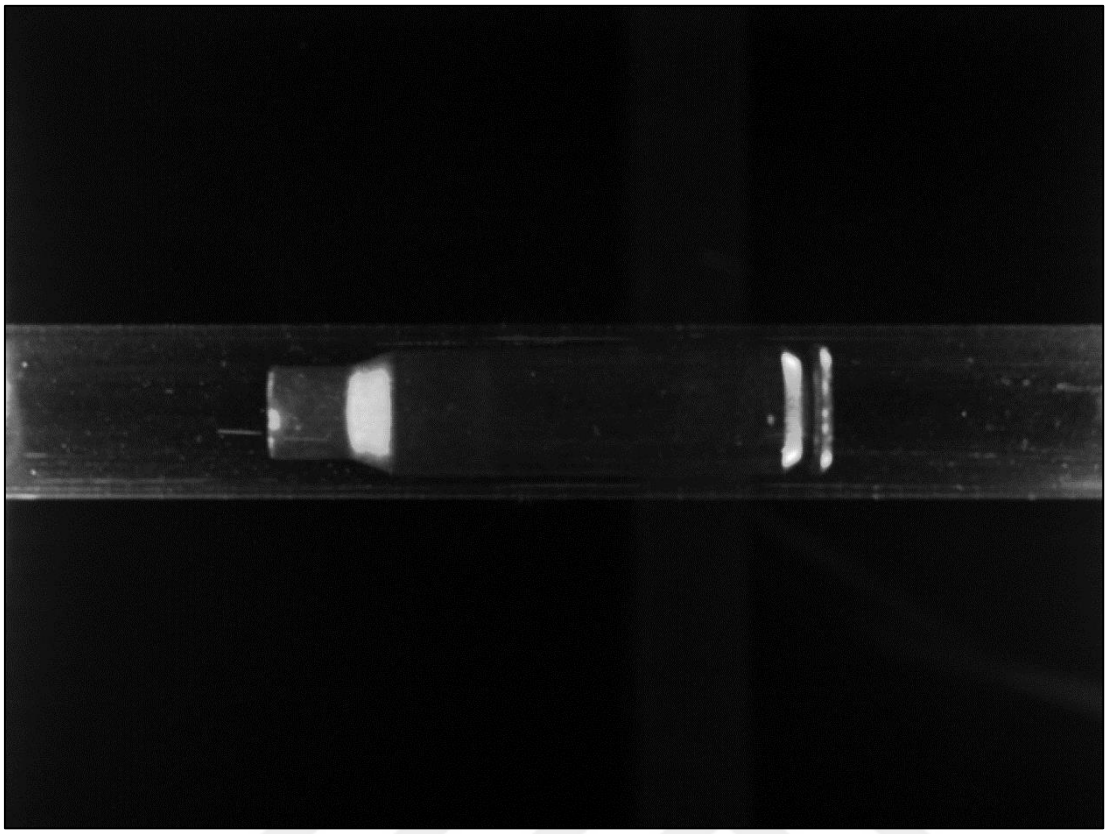
kovanın görüntü üzerindeki yerini bulmak için şablon eşleme algoritması kullanılmaktadır. Bu iki algoritma GPU ile paralelleştirilmiştir. Kovanın yeri bulunduktan sonra kovan parçalara ayrılmaktadır. Kovan üzerinde üç bölge belirlenmekte ve bu bölgeler üzerinde hata olup olmadığına bakılmaktadır. Her bir bölge için iki yanlı filtre, Sobel filtresi, eşikleme ve morfolojik işlemlerden kapama işlemi uygulanmaktadır. Elde edilen sonuçlardan o bölgede hata olup olmadığına bakılmaktadır. Her bir görüntü için dört adet algoritmanın GPU uygulaması gerçekleştirilmiştir. Algoritmaların uygulandığı üç bölge Şekil 6.18'de gösterilmektedir.

6.4.1. Algoritmanın GPU ile paralelleştirilmesi

Geliştirilen algoritma GPU'da paralelleştirilmiştir. Karşılaştırma yapmak için bu algoritmalar CPU'da da çalıştırılarak, her algoritma ayrı ayrı incelenmekte ve bu algoritmaların CPU ve GPU çalışma süreleri ile elde edilen hızlanmalar verilmektedir. Kameradan alınan görüntü Şekil 6.19'da görülmektedir.



6.18. Yan yüzeyin bölgelere ayrılması



Şekil 6.19. Orijinal görüntü

Yan yüzey algoritması içerisinde kullanılan algoritmalarda kullanılan görüntü boyutları, CPU süreleri ve GPU süreleri verilmektedir. Geliştirilen algoritmalar GeForce GT 635M, Tesla C2070 ve Tesla K40 ekran kartlarında çalıştırılmış ve elde edilen sonuçlar verilmiştir. Uygulamada kullanılan ve işlem süreleri ölçülen NVIDIA ekran kartlarının isimleri ve çekirdek sayıları Çizelge 6.3'te verilmektedir.

Çizelge 6.3. Uygulamada test edilen NVIDIA ekran kartları ve çekirdek sayıları

GPU No	**GPU İsmi**	**Çekirdek Sayısı**
GPU1	GeForce GT 635M	96
GPU2	Tesla C2070	448
GPU3	Tesla K40	2880

6.4.2. Gaussian filtresi

Görüntü filtreleme resmin üzerinde bir filtre varmış gibi düşünüp her piksel değerinin yeniden hesaplanmasıdır. Filtreleme sayesinde görüntü üzerinde netleştirme, belirli

ayrıntıları ortaya çıkarma, görüntüyü yumuşatma, kenar keskinleştirme veya kenar bulma gibi işlemler gerçekleştirilir. Filtre boyutları 3×3, 5×5, 7×7, 9×9, 11×11 matrisler şeklinde olabilir.

Gaussian filtreleme bir Fourier dönüşümüdür. Bu filtre ile görüntü yumuşatma işlemi yapılarak orjinalden daha net filtrelenmiş bir görüntü elde edilir. 2B Gauss filtresi Eş. 6.14 formülüyle elde edilmektedir.

$$g(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}} \tag{6.14}$$

Fonksiyonda *g*, *x* ve *y* koordinatlarındaki Gaussian çekirdeği (fonksiyonu)'dir. σ ise Gauss fonksiyonunun keskinliğini ya da yumuşaklığını belirleyecek olan Gauss dağılımının standart sapma değeridir. $1 / 2\pi\sigma^2$ ifadesi ise düzeltme (normalizasyon) katsayısıdır. Gaussian filtresi nokta dağılım fonksiyonu gibi ayrık pikseller üzerinde işlenir.

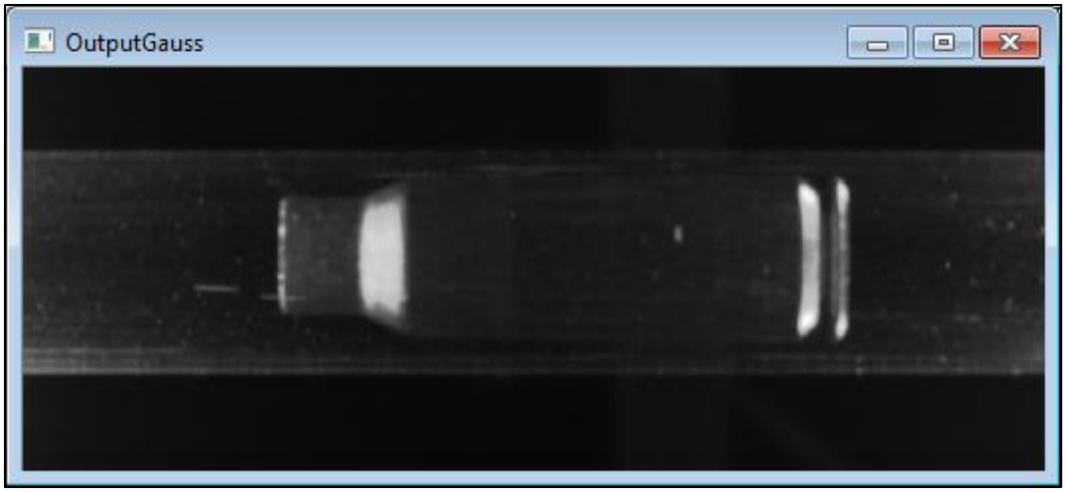
Bu tez çalışması kapsamında görüntüdeki gürültüleri temizleme maksadıyla gerçekleştirilen uygulamada 5X5'lik Gauss filtresi GPU ile hızlandırılmıştır. Kullanılan filtre Eş 6.15'teki gibidir.

$$G(5X5) = \frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \tag{6.15}$$

Uygulamalar 1280X420 boyutlarındaki Şekil 6.17'de verilen siyah beyaz yan yüzey görüntüsü üzerinde uygulanmıştır. Uygulama sonuçları Çizelge 6.4'te verilmektedir. GPU ile elde edilen görüntü ise Şekil 6.20'de verilmektedir.

Çizelge 6.4. 5X5'lük Gaussian filtresi işlem süreleri

GPU No	Seri İşlem Süresi (ms))	GPU Kernel Süresi (ms)	Hızlanma
GPU1	1,79	2,93	0,61
GPU2	2,40	0,86	2,79
GPU3	2,10	0,18	11,67



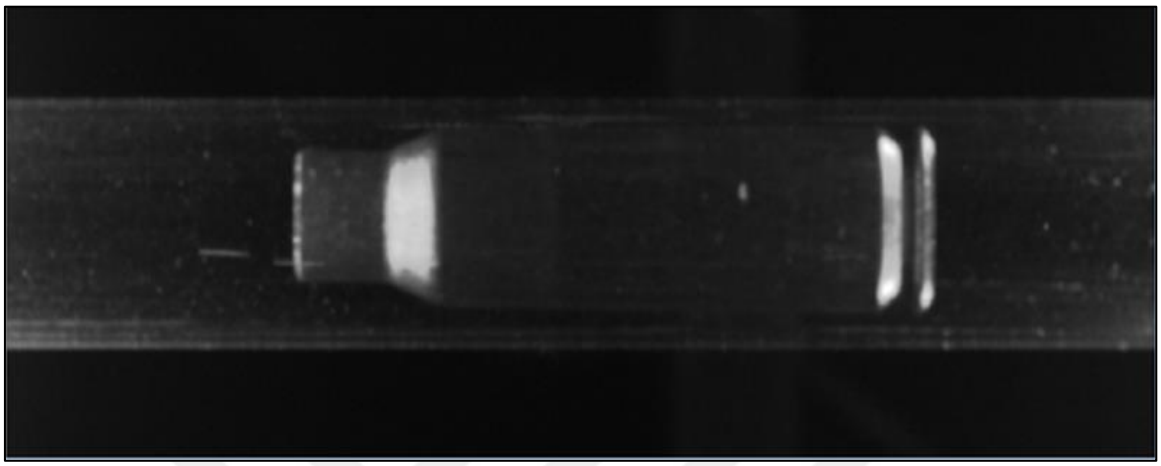
Şekil 6.20. 5X5 Gaussian filtresi sonucu

Gaussian filtresi GPU algoritması Çizelge 6.5'te görüldüğü gibi GeForce 635M ekran kartı ile hızlanma elde edilememiştir. Tesla C2070 ekran kartı ile 2,79 kat, Tesla K40 ekran kartı ile 11,67 kat hızlanma sağlamıştır.

6.4.3. Şablon eşleme

Şablon eşleme ile bir kaynak görüntü üzerinde bir şablon aranır. Bu yöntem görüntü üzerinde şablonu tüm piksellerde dolaşarak aramaktadır. Aranan şablonun bulunabilmesi için görüntü üzerinde birebir aynısının olması gerekmektedir. Aranan şablon görüntü üzerinde kaydırma yöntemi ile gezdirilerek arama yapılır.

Kovan muayene yan yüzey algoritmasında kovanın resim üzerindeki yerini belirlemek için şablon eşleme algoritması kullanılmaktadır. Şekil 6.21'deki görüntüde Şekil 6.22'deki şablon aranmaktadır.



Şekil 6.21. Şablon eşleme kaynak görüntü



Şekil 6.22. Kullanılan şablon

Kovan yan yüzey hatalarını bulmak için kullanılan algoritmada en fazla işlem zamanını şablon eşleme algoritması almaktadır. Algoritmanın CPU ve GPU işlem süreleri Çizelge 6.5'te verilmektedir.

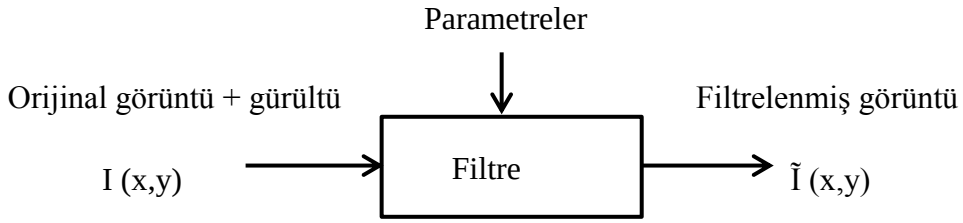
Çizelge 6.5. Şablon eşleme işlem süreleri

GPU No	Seri İşlem Süresi (ms)	GPU Kernel Süresi (ms)	Hızlanma
GPU1	27,65	28,55	0,97
GPU2	32,26	7,42	4,35
GPU3	30,14	1,50	20,09

Elde edilen sonuçlara göre CPU işlem süresi yaklaşık 30 ms ölçülmüştür. Elde edilen sonuçlara göre GeForce 635M ekran kartı ile hızlanma elde edilememiştir. Tesla C2070 ekran kartı ile 4,35 kat hızlanma sağlanmıştır. Tesla K40 ekran kartında GPU işlem süresi 1,5 ms ölçülmüş ve 20 kat hızlanma elde edilmiştir. Şablon eşleme algoritması ile elde edilen bu hızlanma bu tez çalışması kapsamında uygulanan algoritmalar içerisinde elde edilen en yüksek hızlanmadır.

6.4.4. İki yanlı filtre (Bilateral Filtre)

Görüntü üzerindeki parlaklık ve renk bilgilerinin rasgele değişmesi ile oluşan bozukluklar ve lekeler gürültü olarak tanımlanır. Gürültü sensör ve dijital kameralardaki görüntü kirliliğidir. Gürültü görüntülerdeki piksel değerlerini etkiler ve gerçek görüntülerin bozulmasına neden olurlar. Sayısal görüntülerde farklı özelliklerde gürültüler bulunmaktadır. En yaygın gürültü türleri Gauss ve tuz-biber gürültüleridir. Dijital görüntüler üzerindeki gürültülerin ortadan kaldırılması için çeşitli filtreler kullanılmaktadır. Her bir gürültü türünün ortadan kaldırılması için farklı filtreler kullanılmalıdır. Tipik bir görüntü filtresi Şekil 6.23'te görülmektedir.



Şekil 6.23. Tipik görüntü filtresi

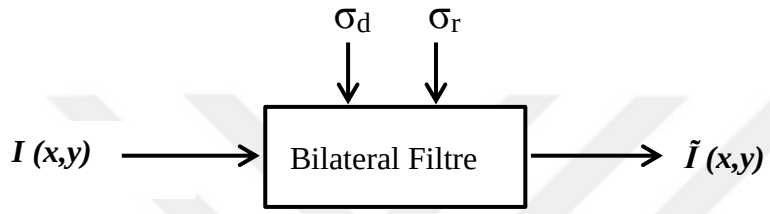
Sayısal görüntü filtresi MxN boyutlarındaki I(x,y) ile gösterilen bir resmin, mxn boyutlarındaki g(x,y) ile gösterilen filtre çekirdeği ile konvolüsyonudur ve I(x,y)*g(x,y) şeklinde gösterilir. Bu işlem Eş. 6.16 ile gösterilir:

$$\tilde{I}(x,y) = g(x,y) * I(x,y) = \sum_{\tau_1=-a}^{a} \sum_{\tau_2=-b}^{b} g(\tau_1,\tau_2) I(x-\tau_1, y-\tau_2) \qquad (6.16)$$

a ve b değerleri maske boyutlarını, $\tilde{I}(x,y)$ maskeleme sonucunda elde edilen görüntüyü ifade etmektedir.

Filtreleme ile görüntüyü netleştirme, yumuşatma, keskinleştirme, kenarları bulma, gürültüyü giderme gibi pek çok işlem gerçekleştirilebilir. Görüntülerden gürültüleri kaldırmak için çok farklı yöntemler geliştirilmiştir.

1998 yılında Tomasi ve Manduchi tarafından iki yanlı filtre önerilmiştir. Komşu piksellerin geometrik yakınlıklarına ve fotometrik benzerliklerine dayalı yapısı vardır. Bu iki parametrenin doğru belirlenmesi gerekir. Bu durumda iki yanlı filtre gürültüleri yok ederken aynı zamanda kenarları da koruyan bir yapıya sahiptir.



Şekil 6.24. İki yanlı filtre şeması

q	q	q
q	p	q
q	q	q

Şekil 6.25. 3x3 boyutlarındaki filtre maskesi

İki yanlı filtrede giriş görüntüsünde p konumunda olan pikselin çıkış değeri Eş. 6.17 formülü ile hesaplanır.

$$\tilde{I}(p) = \frac{1}{C} \sum_{q \in N(p)} w_g(p,q) w_c(p,q)\, I(q) \tag{6.17}$$

Burada $w_g(p,q)$ değeri geometrik benzerlik olup Eş. 6.18 ile hesaplanmaktadır.

$$w_g(p,q) = e^{\frac{-\|q-p\|^2}{2\sigma_d^2}} \tag{6.18}$$

Formülde $\|q - p\|$ merkezle diğer nokta arasındaki uzaklığı göstermektedir. σ_d ise kullanıcı tarafından belirlenen bir değerdir.

$w_c(p, q)$ ise komşu pikselin fotometrik benzerlik olup Eş. 6.19 ile hesaplanmaktadır.

$$w_{gc}(p, q) = e^{\frac{-|I(q)-I(p)|^2}{2\sigma_r^2}} \quad (6.19)$$

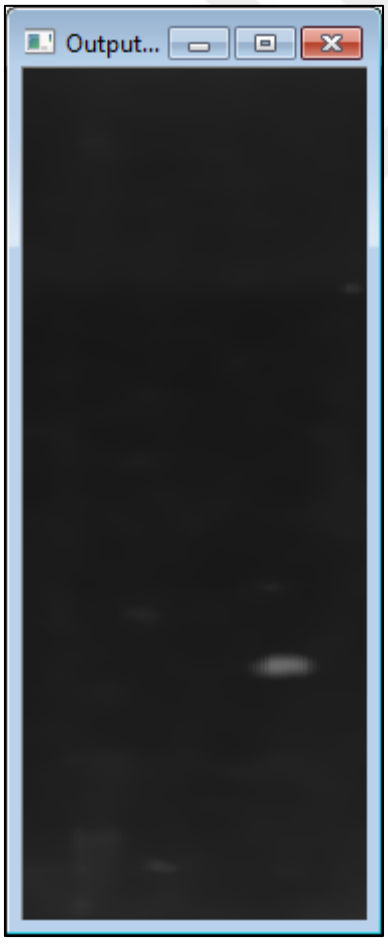
Formülde $|I(q) - I(p)|$ değeri p ve q piksel değerleri arasındaki fark olup σ_{dr} ise kullanıcı tarafından belirlenen ikinci değerdir.

İki yanlı filtre σ_d sonsuza doğru giderse Gauss filtresine, σ_r sonsuza doğru giderse filtre mekânsal olmayan filtrelere yaklaşır. Eğer bu iki parametre ikisi de sonsuza doğru giderse, iki yanlı filtre ortalama filtreye dönüşür. Dolayısıyla kullanıcı tarafından parametre seçimi önemlidir [59].

Bu tez çalışmasında iki yanlı filtrenin CPU ve GPU uygulaması ayrı ayrı gerçekleştirilmiş ve kovan yan yüzey görüntüleri üzerinde çalıştırılmıştır. Her bir yan yüzey görüntüsünde üç parça üzerinde çalıştırılmıştır. Kullanılan filtre boyutu 5X5'tir. Görüntü boyutları ve işlem süreleri Çizelge 6.6'da verilmektedir. Kovan gövdesinin iki yanlı filtre sonucu elde edilen görüntüsü Şekil 6.26'da verilmektedir.

Çizelge 6.6. İki yanlı filtre işlem süreleri

Görüntü Boyutu	GPU No	Seri İşlem Süresi (ms)	GPU Kernel Süresi (ms)	Hızlanma
60X60	GPU1	2,12	1,56	1,36
	GPU2	1,49	0,62	2,40
	GPU3	1,96	0,32	6,13
60X36	GPU1	1,18	0,92	1,28
	GPU2	0,76	0,43	1,77
	GPU3	0,87	0,26	3,35
90X434	GPU1	19,05	11,32	1,68
	GPU2	13,95	2,72	5,13
	GPU3	14,63	0,81	18,06



Şekil 6.26. İki yanlı filtre sonucu

İki yanlı filtre algoritması CPU'da ikinci en yüksek işlem zamanı ölçülen algoritmadır. Boyutları en yüksek olan (90X434) görüntüde iki yanlı filtrenin CPU süresi ortalama 16 ms'dir. GPU uygulamasında elde edilen sonuçlara göre GeForce 635M ekran kartı ile tüm görüntü boyutlarında hızlanma elde edilmiştir. Tesla C2070 ekran kartı ile CPU'ya göre 5,13 kata kadar hızlanma sağlanmıştır. Tesla K40 ekran kartında en yüksek boyutlu görüntüde 18 kat hızlanma sağlanmıştır. Bu hızlanma uygulamada elde edilen en iyi ikinci hızlanmadır.

6.4.5. Sobel filtresi

Sobel filtresi görüntü işlemede en çok bilinen ve kullanılan kenar algılama algoritmalarındandır. Kenar algılama görüntüler içerisindeki nesnelerin algılanmasını sağlar. Sobel algoritmasında iki adet konvolüsyon kerneli kullanılır. Bunlardan birisi yatay kenarları bulmaya yararken diğeri dikey kenarları bulmaya yarar. Sobel filtresinde en çok 3X3'lük filtreler kullanılmaktadır.

Yatay Sobel kernel:

$$s_x(3X3) = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{6.20}$$

Dikey Sobel kernel:

$$s_y(3X3) = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{6.21}$$

Her iki kernel herhangi bir pixel için hesaplandıktan sonra büyüklükleri hesaplanarak kenarlar bulunmuş olur. Gradyan büyüklüğü iki gradyanın kareleri toplamının karekökü olarak hesaplanır (Eş. 6.22).

$$|s| = \sqrt{{s_x}^2 + {s_y}^2} \tag{6.22}$$

Burada kareler toplamının karekökünü almak yerine daha az maliyetli bir işlem olan mutlak değerleri toplamı da alınabilir (Eş. 6.23).

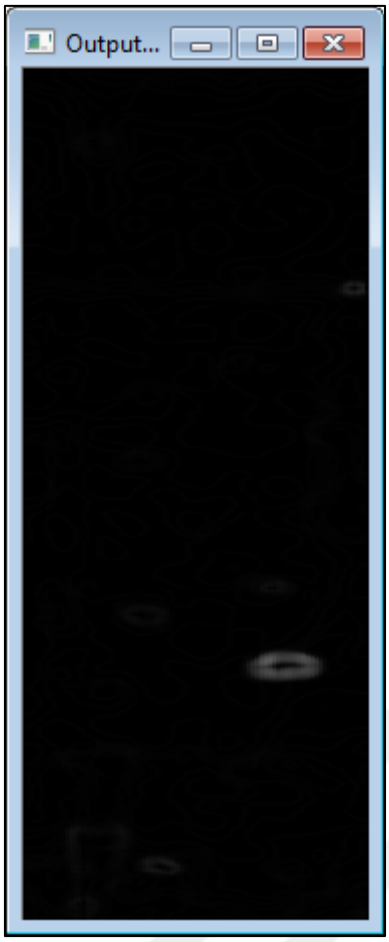
$$|s| = |s_x| + |s_y| \tag{6.23}$$

Bu tez çalışması kapsamında 3X3 lük Sobel filtresi CPU uygulaması ve GPU uygulaması gerçekleştirilmiş ve karşılaştırılmıştır. Elde edilen sonuçlar Çizelge 6.7'de verilmektedir. Kovan gövdesinin Sobel filtresi sonucu elde edilen görüntüsü Şekil 6.27'de verilmektedir.

Çizelge 6.7. Sobel filtresi işlem süreleri

Görüntü Boyutu	GPU No	Seri İşlem Süresi (ms)	GPU Kernel Süresi (ms)	Hızlanma
60X60	GPU1	0,19	2,35	0,08
	GPU2	0,012	0,54	0,02
	GPU3	0,15	0,35	0,43
60X36	GPU1	0,17	2,23	0,08
	GPU2	0,026	0,53	0,05
	GPU3	0,08	0,25	0,32
90X434	GPU1	0,51	2,75	0,19
	GPU2	0,16	0,74	0,22
	GPU3	0,3	0,15	2,00

Uygulanan Sobel filtresi sonuçlarına göre sadece 90 X 434 çözünürlüğündeki görüntüde Tesla K40 ekran kartı ile 2 kat hızlanma elde edilmiştir. Diğer görüntü boyutlarında hızlanma elde edilememiştir. Ekran kartlarında elde edilen süreler karşılaştırıldığında çekirdek sayısı en düşük olan GeForce ekran kartının işlem süresinin en yüksek, çekirdek sayısı en yüksek olan Tesla K40 ekran kartının en düşük sürede algoritmayı tamamladığı görülmektedir. Ekran kartlarında elde edilen sonuçlar çekirdek sayısı artıkça işlem süresinin azaldığını göstermektedir. Uygulamada ölçülen seri işlem süreleri OpenCV kütüphanesinin fonksiyonlarında ölçülen sürelerdir. Hem CPU kodlarının optimize olması hem de çözünürlüğün düşük olması nedeniyle hızlanma sadece en yüksek görüntü boyutunda ve en iyi ekran kartında elde edilmiştir.



Şekil 6.27. 3X3 Sobel filtresi sonucu

6.4.6. Eşikleme

Görüntü bölütleme, bir görüntüyü her biri içerisinde farklı özelliklerin tutulduğu anlamlı bölgelere ayırmak olarak tarif edilebilir. Eşikleme, görüntü bölütleme için kullanılan en önemli ve temel yaklaşımlardan birisidir. Eşikleme işleminden amaç, görüntü içerisindeki nesneleri görüntü arka planından ayırmaktır. Nesneleri arka plandan ayırmak için en kolay yol bir T eşik değeri ile görüntüdeki piksel değerlerini karşılaştırmak olacaktır. Buna göre, görüntüdeki herhangi bir (i, j) pikseli için; f(i, j) > T ise (i, j) pikseli nesneye ait bir nokta, f(i,j) ≤ T ise (i, j) pikseli arka plana ait bir nokta olacaktır. Tüm eşikleme tekniklerinin en basiti, tek bir eşik değeri kullanarak bölütlemedir. Bu tür eşikleme işlemine global eşikleme denir.

Global eşiklemenin uygulanacağı görüntülerde, aydınlatmanın önemli bir payı vardır. Farklı aydınlatmalara maruz kalmış nesnenin bütünsel eşikleme sonucunda üreteceği ikili

görüntüler farklı olacaktır. Bu çerçeveden bakıldığında, büyük oranda kontrol edilebilen ortamlara bütünsel eşiklemenin başarıyla uygulanabileceği açıktır. Global eşiklemenin kullanılabileceği alanlar, endüstriyel muayene uygulamaları olarak verilebilir çünkü bu tür uygulamalarda aydınlatmanın kontrol edilebilmesi genellikle mümkündür. Global eşiklemede uygun eşik değeri farklı algoritmalara göre belirlenebilir. Otsu ve Kapur en çok bilinen eşik değer hesaplama yaklaşımlarındandır.

Bu tez çalışması kapsamında yapılan uygulamada eşik değeri Otsu'ya göre hesaplanmaktadır. Eşikleme sonucu elde edilen işlem süreleri Çizelge 6.8'de verilmektedir.

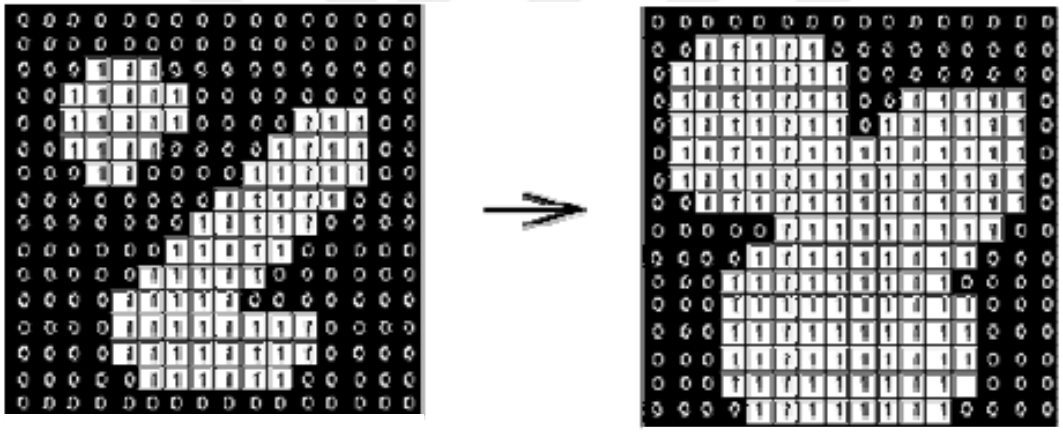
Çizelge 6.8. Eşikleme işlem süreleri

Görüntü Boyutu	GPU No	Seri İşlem Süresi (ms)	GPU Kernel Süresi (ms)	Hızlanma
60X60	GPU1	0,026	0,1	0,26
	GPU2	0,016	0,022	0,73
	GPU3	0,02	0,015	1,33
60X36	GPU1	0,007	0,1	0,07
	GPU2	0,003	0,02	0,15
	GPU3	0,006	0,01	0,60
90X434	GPU1	0,11	0,13	0,85
	GPU2	0,05	0,02	2,50
	GPU3	0,07	0,01	7,00

Elde edilen sonuçlara göre görüntü boyutu yükseldikçe GPU ile elde edilen hızlanmalar artmaktadır. Eşikleme algoritmasında en iyi hızlanma 7 kat olarak hesaplanmıştır. Bu hızlanma en yüksek görüntü boyutu (90X434) ve en iyi ekran kartında (Tesla K40) elde edilmiştir. En düşük görüntü boyutu olan 60 X 36 boyutlarında elde edilen sonuçlarda hızlanma elde edilememiştir. 60 X 60 boyutlarındaki görüntüde ise Tesla K40 ekran kartında 1,33 kat hızlanma elde edilmiştir.

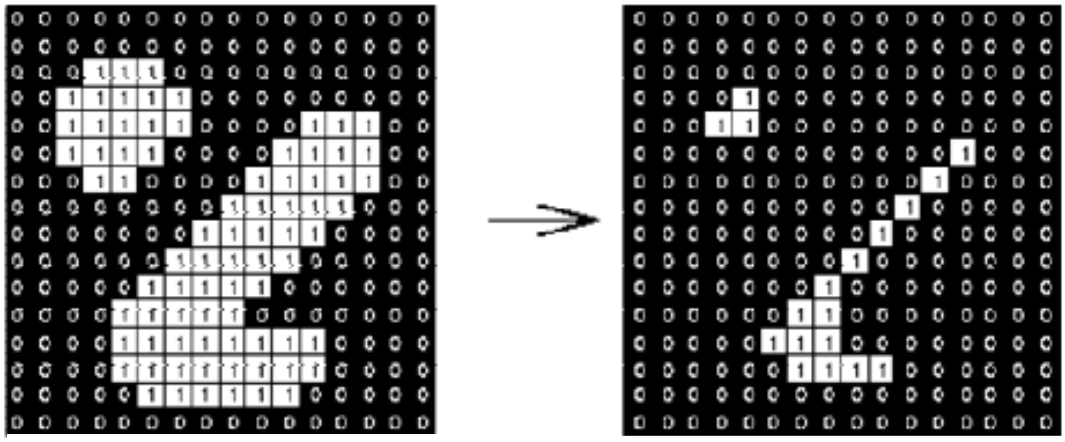
6.4.7. Morfolojik operasyonlar

Morfoloji görüntü işlemede temel küme işlemlerine dayanan sınır kenar gibi görüntü bileşenlerinin ayrıştırılmasına yarayan uygulamalar için kullanılan bir yaklaşımdır. Morfolofik görüntü işlemede aşındırma (erosion) ve genişletme (dilation) isimli iki temel yaklaşım kullanılır. Uygulamada genellikle görüntü ikili moda çevrilir. Aşındırma ikili bir görüntüde bulunan nesnelerin boyutunu seçilen yapısal elemente bağlı olarak küçültürken, genişletme nesnenin alanını artırır. Bu işlemlerden aşındırma işlemi birbirine ince bir gürültü ile bağlanmış iki veya daha fazla nesneyi birbirinden ayırmak için kullanılırken, genişletme işlemi ise aynı nesnenin bir gürültü ile ince bir şekilde bölünerek ayrı iki nesne gibi görünmesini engellemek için kullanılır. Görüntü üzerinde genişletme işlemi Şekil 6.28'de, aşındırma işlemi de Şekil 6.29'da şematik olarak gösterilmektedir.



(a) (b)

Şekil 6.28. Genişletme işlemi (a) öncesi (b) sonrası

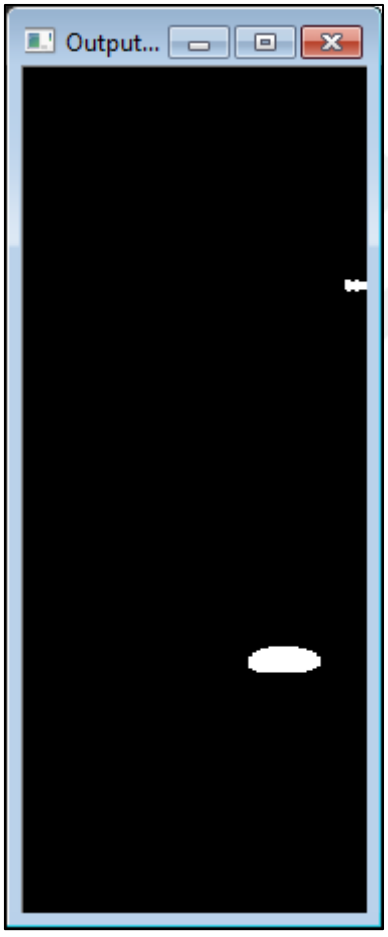


(a) (b)

Şekil 6.29. Aşındırma işlemi (a) öncesi (b) sonrası

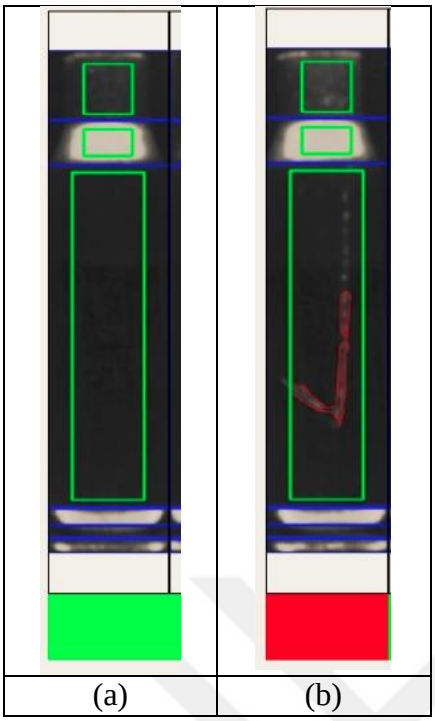
Morfolojik işlemler açma ve kapama olmak üzere iki uygulama vardır. Görüntünün önce aşındırma sonra genişletmeye tabi tutulması işlemine açma denir. Görüntünün önce genişletme sonra aşındırma işlemine tabii tutulmasına ise kapama ismi verilir. Açma işlemi küçük nesneleri görüntüden kaldırırken, kapama işlemi küçük boşlukları görüntüden kaldırır.

Bu tez çalışması kapsamında kovan yan yüzey algoritmasında görüntüler üzerinde kapama işlemi uygulanmaktadır. 7X7'lik bir yapısal element kullanılmaktadır. Elde edilen işlem sonucunda yan yüzey görüntüsünün kapama işlemi sonucu Şekil 6.30'da verilmektedir.



Şekil 6.30. Kapama işlemi sonucu

Geliştirilen arayüz yazılımının yan yüzey hataları ile ilgili ekran görüntüleri Şekil 6.31'de görülmektedir.



(a) (b)

Şekil 6.31. Geliştirilen arayüz yazılımında yan yüzey hataları (a) hatasız kovan (b) hatalı kovan

Görüntüler üzerinde çalışan GPU algoritmalarından sonuncusu olan kapama işleminin işlem süreleri Çizelge 6.9'da verilmektedir. Elde edilen sonuçlara göre en yüksek görüntü boyutlarındaki (90 X 434) görüntüde Tesla K40 ekran kartında seriye göre 2,83 kat hızlanma elde edilmiştir. Diğer ekran kartlarında ve görüntü boyutlarında hızlanma elde edilememiştir.

Çizelge 6.9. Morfolojik operasyonlardan kapama işlem süreleri

Görüntü Boyutu	**GPU No**	**Seri İşlem Süresi (ms)**	**GPU Kernel Süresi (ms)**	**Hızlanma**
60X60	GPU1	0,11	1,31	0,08
	GPU2	0,13	0,51	0,25
	GPU3	0,13	0,21	0,62
60X36	GPU1	0,12	0,94	0,13
	GPU2	0,06	0,41	0,15
	GPU3	0,1	0,3	0,33
90X434	GPU1	0,61	4,76	0,13
	GPU2	0,77	1,06	0,73
	GPU3	0,68	0,24	2,83

6.4.8. GPU'da koşturulan algoritmaların değerlendirilmesi

Çizelge 6.10'da GPU kullanılarak paralelleştirilen tüm algoritmaların üç farklı GPU için çalışma süreleri ve hızlanma oranları verilmektedir. Elde edilen sonuçlara göre GPU uygulaması yapılan algoritmalarda GeForce ekran kartında hızlanma elde edilememiştir. Tesla C2070 ekran kartında algoritmaların hızlanmalarının aritmetik ortalaması 1,52 kat olarak hesaplanmıştır. Tesla K40 ekran kartında elde edilen sonuçlara göre hızlanmaların aritmetik ortalaması 5,34 kat olarak hesaplanmıştır.

Çizelge 6.10. Tüm algoritmaların GPU işlem süreleri

Görüntü Boyutu	**Kullanılan algoritmalar**	**Geforce (ms)**	**Tesla C2070 (ms)**	**Tesla K40 (ms)**	**Hızlanma Geforce**	**Hızlanma C2070**	**Hızlanma K40**
1188X370	Gauss Filtresi	2,93	0,86	0,18	0,61	2,79	11,67
1188X370 665X149	Template Matching	28,55	7,42	1,5	0,97	4,35	20,09
60X60	Bilateral	1,56	0,62	0,32	1,36	2,4	6,13
	Sobel	2,35	0,54	0,35	0,08	0,02	0,43
	Threshold	0,1	0,022	0,015	0,26	0,73	1,33
	Morphology	1,31	0,51	0,21	0,08	0,25	0,62
60X36	Bilateral	0,92	0,43	0,26	1,28	1,77	3,35
	Sobel	2,23	0,53	0,25	0,08	0,05	0,32
	Threshold	0,1	0,02	0,01	0,07	0,15	0,6
	Morphology	0,94	0,41	0,3	0,13	0,15	0,33
90X434	Bilateral	11,32	2,72	0,81	1,68	5,13	18,06
	Sobel	2,75	0,74	0,15	0,19	0,22	2
	Threshold	0,13	0,02	0,01	0,85	2,5	7
	Morphology	4,76	1,06	0,24	0,13	0,73	2,83
	Toplam	59,95	15,90	4,60			
	Ortalama				0,56	1,52	5,34

Yapılan ölçümlerde GPU uygulaması geliştirilen algoritmaların başlangıç ve bitişleri dikkate alınarak hız ölçümü yapılmıştır. Kovan hatalarının tespitinde kullanılan ekran kartı Tesla K40 ekran kartıdır. Diğer iki ekran kartı hızlanmaların test edilmesi amacıyla kullanılmıştır. Gerçek sistem üzerinde çalışan CPU ve Tesla K40 ekran kartı ile elde edilen süreler Çizelge 6.11'de verilmektedir. Tabloya göre algoritmaların toplam CPU süresi 51,24 ms, Tesla K40 GPU süresi 4,60 ms olarak ölçülmüştür. Buna göre algoritmalarda elde edilen hızlanma 11,14 kat olarak hesaplanmıştır.

Çizelge 6.11. Tesla K40 sonuçları

Görüntü Boyutu	**Kullanılan algoritmalar**	**CPU süre (ms)**	**Tesla K40 süre (ms)**	**Hızlanma**
1188X370	Gauss Filtresi	2,1	0,18	11,67
665X149	Template Matching	30,14	1,5	20,09
60X60	Bilateral	1,96	0,32	6,13
	Sobel	0,15	0,35	0,43
	Threshold	0,02	0,015	1,33
	Morphology	0,13	0,21	0,62
60X36	Bilateral	0,87	0,26	3,35
	Sobel	0,08	0,25	0,32
	Threshold	0,006	0,01	0,60
	Morphology	0,1	0,3	0,33
90X434	Bilateral	14,63	0,81	18,06
	Sobel	0,3	0,15	2,00
	Threshold	0,07	0,01	7,00
	Morphology	0,68	0,24	2,83
	Ortalama			5,34
	Toplam	51,24	4,60	11,80

Burada ölçülen süreler verilen algoritmaların başlangıç ve bitiş zamanları arasındaki fark alınarak hesaplanmıştır. Yan yüzey algoritmasında bu algoritmalar dışındaki işlemler CPU'da yaklaşık 4,7 ms sürede tamamlanmaktadır. Yan yüzey GPU algoritmasında ise GPU'daki işlemler dışındaki süre 7,4 ms'de tamamlanmaktadır. Bu süre içerisinde CPU'da

gerçekleştirilen işlemler, CPU'dan GPU'ya ve GPU'dan CPU'ya veri iletim süresi bulunmaktadır. Burada ölçülen işlem süreleri donanıma göre farklılık gösterebilir.

6.5. OpenMP ile Paralelleştirme

Yan yüzey algoritması tek görüntü için CPU ile 56 ms sürede tamamlanmaktadır. Tesla K40 ekran kartı ile paralelleştirlmiş GPU algoritması yaklaşık 12 ms'de tamamlanmaktadır. GPU paralelleştirmesi sonucu 4,67 kat hızlanma elde edilmiştir. 8 adet görüntü için algoritma CPU'da seri olarak çalıştırıldığında 448 ms süre alırken GPU ile yaklaşık 96 ms'de işlenmektedir.

OpenMP ile algoritma işlemci çekirdeklerinde paralelleştirilmiştir. Hata bulma algoritması için üç iş parçacığı oluşturulmuş ve ağız, kapsül ve yan yüzey görüntüleri bu iş parçacıklarında çalıştırılmıştır. OpenMP ile CPU paralelleştirmesi yapılmadan sadece GPU paralelleştirmesi yapılarak, ağız hataları 70 ms, kapsül hataları 65 ms, yan yüzey hataları ise 96 ms süre alan hata bulma algoritmaları toplamda 231 ms sürede almaktadır. OpenMP ile CPU paralelleştirlmesi sonucunda kovan hataları bulma algoritmalarının tümü 99 ms.'de tamamlanmaktadır. OpenMP ile 2,3 kat hızlanma sağlanmıştır.

Tüm algoritmalar seri olarak işlenmesi durumunda 583 ms'de tamamlanmaktadır. Geliştirilen GPU ve CPU'nun birlikte kullanıldığı hibrit algoritma sonucunda işlem süresi 99 ms olmaktadır. Elde edilen hızlanma yaklaşık 5,9 kat olmuştur. Dakikada 300 kovan işleme kapasitesine ulaşmak için bir kovanın 200 ms'de görüntülerinin işlenmesi gerekmektedir. Tesla K40 GPU've 4 çekirdekli işlemciye sahip bir bilgisayar kullanılarak geliştirilen hibrit algoritma ile bu süre 99 ms'ye indirilmiş ve çalışmanın başında konulan hedefe ulaşılmıştır.

7. SONUÇ VE ÖNERİLER

Bu çalışmada görüntü işleme teknikleri ve paralel veri işleme algoritmaları kullanarak hatalı olan kovanları ayıran hızı yüksek ve maliyeti düşük otomatik bir sistem geliştirilmiştir. Kovan hatalarının bulunması için gerçek zamanlı sistemde elde edilen görüntüler kullanılmış ve işlenmiştir. Elde edilen görüntülerin çözünürlüklerinin yüksek olması ve görüntü sayısının fazla olması işlenen veri miktarını da artırmaktadır. Ayrıca işlenen verinin silindirik metal bir nesne olması nedeniyle hatanın tespit edilmesi için karmaşık görüntü işleme adımlarının uygulanması zorunludur. Bu nedenle geliştirilen algoritmalarda pek çok görüntü işleme tekniği bir arada kullanılmıştır. Bu algoritmalarda hedef doğruluk oranının artırılması olmuş ve algoritmaların geliştirilmesi aşamasında üretilen yaklaşımlar ile hatalı kovanlar %97 gibi yüksek bir doğruluk oranı ile tespit edilmiştir.

Bu tez çalışması ile birlikte yürütülen SANTEZ projesi kapsamında üretim kapasitesi en fazla olan 7.62 mm çapındaki hatalı kovanları otomatik olarak tespit edip ayıran bir kovan muayene tezgâhının prototipi geliştirilmiştir. Sistemin kapasitesi dakikada 300 kovan olarak hedeflenmiş ve bu hedefe ulaşılmıştır. Her bir kovan için sekiz adet yan yüzey, bir adet ağız ve bir adet kapsül olmak üzere toplam 10 adet görüntü alınmış ve işlenmiştir. Alınan kovan görüntüleri görüntü işleme teknikleri ile paralel veri işleme algoritmaları kullanılarak işlenmiş ve hatalı kovanlar tespit edilerek ayıklanmışlardır.

Gerçek zamanlı çalışan bir sistemde belli bir süre içerisinde tüm verilerin işlenerek sonucun elde edilmesi zorunluğu bulunmaktadır. Bu süre gerçekleştirilen kovan muayene sisteminde iki kovan arasında geçen süredir ve hedeflenen süre 200 ms'nin altına inmektir. Burada bir kovan için görüntülerin alınması işlenmesi, hatalı ya da hatasız olarak işlem sonucunun belirlenmesi, sonucun sisteme aktarılarak hatalı kovanın ayrıştırılması 200 ms içerisinde gerçekleştirilmelidir. Gerçekleştirilen sistem ile hedeflenen kapasiteye hem donanım hem de yazılım olarak ulaşılmıştır.

Gerçekleştirilen sistem ile her bir kovan için alınan görüntüler eşzamanlı olarak işlenmiştir. İşlemlerin daha kısa sürede bitirilmesi için paralel veri işleme teknikleri kullanılmıştır. Sistemde kovan hatalarını tespit algoritmasında yan yüzey görüntülerini GPU kullanarak, ağız ve kapsül görüntüleri ise CPU çekirdeklerini kullanarak paralelleştiren hibrit bir yapı kullanılmıştır. GPU uygulamasında CUDA platformu, işlemci çekirdeklerinin

paralelleştirmesinde ise OpenMP kullanmıştır. Hibrit sistem CPU işlemci çekirdeklerinden birincisinde ağız hataları algoritması, ikincisinde kapsül hataları algoritması, üçünsünde ise sekiz adet yan yüzey hataları algoritması çalıştırılacak şekilde tasarlanmıştır. Sekiz adet görüntüden oluşan yan yüzey hataları algoritması aynı zamanda CUDA kullanılarak GPU çekirdeklerinde paralelleştirimiştir.

Yan yüzey hatalarını bulmak için kullanılan algoritmada eşikleme, Gauss filtresi, Sobel filtresi, iki yanlı filtre, şablon eşleme ve morfolojik operasyonlar kullanılmıştır. Bu algoritmaların tümü GPU ile paralelleştirilmiş ve işlem süreleri ölçülmüştür. GPU ile Tesla K40 ekran kartında ortalama 4,67 kat hızlanma elde edilmiştir. OpenMP ile paralelleştirme sonucu elde edilen hızlanma 2,3 kat olmuştur. Hem OpenMP hem GPU uygulaması sonucu elde edilen toplam hızlanma ise 5,9 kat olarak hesaplanmıştır. Böylece seri olarak 583 ms'de tamamlanan kovan hatalarını bulma algoritması paralel olarak 99 ms'de tamamlanarak çalışmanın başında belirlenen hedefe ulaşılmıştır.

MKE Gazi Fişek fabrikasında mevcut durumda 7.62 mm çapındaki kovanların muayenesi için kullanılan gözle muayene tezgâhlarında üretim kapasitesi dakikada 125 kovandır. Yani her bir üretim bandı için iki adet gözle muayene tezgâhı kullanılmaktadır. Her bir gözle muayene tezgâhında iki personel çalıştığından her bir üretim bandında dört personel çalışmaktadır. Geliştirilen sistemle bu personellere ihtiyaç duyulmaması işçilik maliyetini düşürmekte, insan faktörüne bağlı olmayan üretim sayesinde kalitede standardın yakalanmasını ve kalitenin arttırılmasını sağlamaktadır. Ayrıca NATO standartlarına göre en fazla %4 lük hata kabul edilmektedir. Geliştirilen sistemde hatalı kovanlar %97 gibi bir doğruluk oranı ile tespit edilmiş ve böylece NATO standartlarına uygun üretim yapılması da sağlanmıştır.

KAYNAKLAR

1. Algan, S. (2016). *Fişek üretiminde risklerin tespiti ve kimyasal maruziyetin değerlendirilmesi,* İş Sağlığı ve Güvenliği Uzmanlık Tezi, Ankara.

2. Akpolat, M.S. (2014). *Türkiye`de ve Avrupa'da kazanların güvenli çalışma ve periyodik kontrol kriterleri,* İş Sağlığı ve Güvenliği Uzmanlık Tezi, Ankara.

3. MIL-STD-636. (1958, June) *Military Standard: Visual inspection standards for small arms ammunition through caliber 0.50.* Department of Defense, United States of America.

4. MIL-STD-651. (1962, October) *Military Standard: Visual inspection standards for 20mm ammunition and components, Department of Defense.* United States of America.

5. Çomak, B., Beycioğlu, A., Başyiğit, C., ve Kılınçarslan, Ş. (2011, 16-18 Mayıs). Beton Teknolojisinde Görüntü İşleme Tekniklerinin Kullanımı, 6th International Advanced Technologies Symposium, 220-228. Elazığ.

6. Jarp, S., Lazzaro, A., Nowak, A., and Valsan, L., 2012, Comparison of Software Technologies for Vectorization and Parallelization, https://cds.cern.ch/record/1696945/files/CERN-OPEN-2014-029.pdf.

7. Temizel, A., Halici, T., Logoglu, B., Temizel, T.T., Omruuzun, F., and Karaman, E., (2011, December). Experiences on Image and Video Processing with CUDA and OpenCL, Wen-Mei W. Hwu (eds). *GPU Computing Gems*, 547-567.

8. Chakroun, I., Melab N., Mezmaz M., and Tuyttens D. (2013). Combining multi-core and GPU computing for solving combinatorial optimization problems. *Journal of Parallel and Distributed Computing,* 73, 1563–1577.

9. Sanchez, M.G.,Vidal, V., Bataller, J., and Arnal, J. (2013). A Parallel Method for Impulsive Image Noise Removal on Hybrid CPU/GPU Systems, Internatinal Conference on Computational Science, 18, 2504-2507.

10. Xue-wu, Z., Yan-qiong, D., Yan-yun, L., Ai-ye, S., and Rui-yu, L. (2011). A Vision inspection system for the surface defects of strongly reflected metal based on multi-class SVM, *Expert Systems with Applications*, 38(5), 5930-5939

11. Zheng, H., Kong, L.X., and Nahavandi, S. (2002). Automatic inspection of metallic surface defects using genetic algorithms, *Journal of Materials Processing Technology*, 125-126, 427-433

12. Rosati, G.,Boschetti, G., Biondi, A., and Rossi, A. (2009). Real-time defect detection on highly reflective curved surfaces, *Optics and Lasers in Engineering*, 47(3-4), 379-384.

13. Wang, H.,Yang, J., and Hu, Z. (2011). Current Status and Prospect of Roller Bearing Surface Defect Detection, *Procedia Engineering*, 15, 4331 – 4336.

14. Agarwal, K., Shivpuri, R., Zhu, Y., Chang, T., and Huang, H. (2011). Process knowledge based multi-class support vector classification (PK-MSVM) approach for surface defects in hot rolling, *Expert Systems with Applications*, 38, 7251-7262.

15. Laofor, C., and Peansuap, V. (2012). Defect detection and quantification system to support subjective visual quality inspection via a digital image processing: A tiling work case study, *Automation in Construction*, 24, 160-174.

16. Tsai, D., Wu, S., and Li, W. (2012). Defect detection of solar cells in electroluminescence images using Fourier image reconstruction, *Solar Energy Materials & Solar Cells*, 99, 250-262.

17. Samtaş, G., ve Gülesin, M. (2012). Sayısal görüntü işleme ve farklı alanlardaki uygulamaları, *Electronic Journal of Vocational Colleges*, 1(2), 85-97.

18. Çomak, B., Beycioğlu, A., Başyiğit, C., ve Kılınçarslan, Ş. (2011, 16-18 Mayıs). Beton Teknolojisinde Görüntü İşleme Tekniklerinin Kullanımı, 6th International Advanced Technologies Symposium, 220-228, Elazığ, Turkey.

19. Schenk O., Christen M., and Burkhart H. (2008). Algorithmic performance studies on graphics processing units, *Journal of Parallel and Distributed Computing*, 68(10), 1360-1369.

20. Diez D.C., Moser D., Schoenegger A., Pruggnaller S., and Frangakis A.S. (2008). Performance evaluation of image processing algorithms on the GPU, *Journal of Structural Biology*, 164(1), 153–160.

21. Belleman R.G., Bedorf J., and Zwart S.F.P. (2008). High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA, *New Astronomy*, 13(2), 103-112.

22. Ram M.P.M., Kurfess T.R., and Tucker T.M. (2008). Least-squares fitting of analytic primitives on a GPU, *Journal of Manufacturing Systems*, 27(3), 130-135.

23. Ryoo S., Rodrigues C.I., Stone S.S., Stratton J.A., and Ueng S. (2008). Program optimization carving for GPU computing, *Journal of Parallel and Distributed Computing*, 68(10), 1389-1401.

24. Diez D.C., Mueller H., and Frangakis A.S. (2007). Implementation and performance evaluation of reconstruction algorithms on graphics processors, *Journal of Structural Biology*, 157(1), 288-295.

25. Stone S.S., Haldar J.P., Tsao S.C., Hwu W.W., Sutton B.P., and Liang Z.P. (2008). Accelerating advanced MRI reconstructions on GPUs, *Journal of Parallel and Distributed Computing*, 68(10), 1307-1318.

26. Che S., Boyer M., Meng J., Tarjan D., Sheaffer J.W., and SKadron K. (2008). A performance study of general-purpose applications on graphics processors using CUDA, *Journal of Parallel and Distributed Computing*, 68(10), 1370–1380.

27. Qi, M., Sun, G., and Chen, G. (2011). Parallel and SIMD Optimization Of Image Feature Extraction, *Procedia Computer Science*, 4, 489-498

28. Bergen, R.V., Lin, H., Alexander, M.E., and Bidinosti, C. (2015). 4D MR phase and magnitude segmentations with GPU parallel computing, *Magnetic Resonance Imaging*, 33(1), 134-145.

29. D'ambra, P., and Filippone, S. (2016). A parallel generalized relaxation method for high-performance image segmentation on GPUs, *Journal of Computational and Applied Mathematics*, 293, 35-44.

30. Swargha, K., and Rodrigues, P. (2012). Performance Amelioration of Edge Detection Algorithms Using Concurrent Programming, *Procedia Engineering*, 38, 2824-2831.

31. Zhang, J.,You, S., and Gruenwald, L. (2014). Parallel online spatial and temporal aggregations on multi-core CPUs and many-core GPUs, *Information Systems*, 44, 134-154.

32. Li, A., Kumar, A., Ha, Y., and Corporaal, H. (2015). Correlation ratio based volume image registration on GPUs, *Microprocessors and Microsystems*, 39 (8), 998-1011.

33. Shams, R.,Sadeghi, P., Kennedy, R., and Hartley, R. (2010). Parallelcomputation of mutualinformation on the GPU with application to real-time registration of 3D medical images, *Computer Methods and Programs in Biomedicine*, 99(2), 133–146.

34. Saha D., Darji K., Patel N., and Thakore D. (2016). Implementation of Image Enhancement Algorithms and Recursive Ray Tracing using CUDA, *Procedia Computer Science* 79, 516–524.

35. Qin L., Sheng B., Lin W., Wu W., and Shen R. (2015). GPU-Accelerated Video Background Subtraction Using Gabor Detector, *Journal of Visual Communication and Image Representation*, 32, 1–9.

36. Werff, H.M.A., and Bakker, W.H. (2014). Implementation and performance of a general purpose graphics processing unit in hyper spectral image analysis, *International Journal of Applied Earth Observation and Geoinformation*, 26, 312-321.

37. Ganiau, M., and Onchiş, D.M. (2014). Face and marker detection using Gabor frames on GPUs, *Signal Processing*, 96, 90-93.

38. Akgün, D., Sakoğlu, Ü., Esquivel, J., Adinoff, B., and Mete, M., (2015). GPU accelerated dynamic functional connectivity analysis forfunctional MRI data, *Computerized Medical Imaging and Graphics*, 43, 53–63.

39. Chouchene M., Sayadi F. E., Bahri H., Dubois J., and Miteran J. (2015). Optimized parallel implementation of face detection based on GPU component, *Microprocessors and Microsystems*, 39 (6), 393-404.

40. Lastra, M., Carabano, J., Gutierrez, P.D., Benitez, J.,M., and Herrera, F. (2015). Fast fingerprint identification using GPUs, Information Sciences, 301, 195–214.

41. Zhu, F.,Gonzales, D., Carpenter, T., Atkinson, M., and Wardlaw., J. (2012). Parallel perfusion imaging processing using GPGPU, *Computer Methods and Programs in Biomedicine*, 108, 1012–1021.

42. Arslan E. (2011, Ocak). *Hücresel Sinir Ağı Sistemleri Kullanılarak Kullanarak Hareketli Nesnelerin Görüntü İşleme Uygulamaları*, Doktora Tezi, İstanbul Üniversitesi, Fen Bilimleri Enstitüsü, İstanbul.

43. Solak, S., ve Altınışık, U. (2018). Görüntü işleme teknikleri ve kümeleme yöntemleri kullanılarak fındık meyvesinin tespit ve sınıflandırılması, *Sakarya Üniversitesi Fen Bilimleri Enstitüsü Dergisi*, 22 (1), 56-65.

44. Kahraman, M. (2017). Akciğer BT görüntülerinin bülütlenmesi ve Matlab ile nodül tespiti, Yeni Yüzyıl Üniversitesi, Mühendislik ve Mimarlık Fakültesi, Elektronik-Elektronik Mühendisliği Bitirme Projesi.

45. Demirhan A., ve Güler İ. (2010). Özörgütlemeli harita ağları ve gri düzey eş oluşum matrisleri ile görüntü bölütleme, *Journal of the Faculty of Engineering and Architecture of Gazi University*, 25(2), 285-291.

46. Uslu R. (2007) Elektronik bir hücresel yapay sinir ağı gerçekleşmesi olan ACE16K üzerinde görüntü bölütleme, Yüksek Lisans Tezi, İstanbul Teknik Üniversitesi, Fen Bilimleri Enstitüsü, İstanbul.

47. İnternet: GISGeography. Image Classification Techniques in Remote Sensing. URL: http://www.webcitation.org/query?url=https%3A%2F%2Fgisgeography.com%2Fimage-classification-techniques-remote-sensing%2F&date=2019-02-10, Son Erişim Tarihi: 10 Şubat 2019.

48. İnternet: Abtikan, Saygın. BEÜ Mühendislik Fakültesi Geomatik Mühendisliği Bölümü Fotogrametri II Ders Notları. *DocPlayer*. URL: http://www.webcitation.org/query?url=https%3A%2F%2Fdocplayer.biz.tr%2F39470972-Beu-muhendislik-fakultesi-geomatik-muhendisligi-bolumu-jdf-330-336-fotogrametri-ii.html&date=2019-02-10, Son Erişim Tarihi: 10 Şubat 2019.

49. Ercan U., Akar H., ve Koçer A. (2013, 23 Ocak). Paralel Programlamada Kullanılan Temel Algoritmalar, *Akademik Bilişim 2013–XV. Akademik Bilişim Konferansı Bildirileri*, 861-865.

50. Chapman, B.,Jost, G., and Pas, R. (2008). Using OpenMP: Portable Shared Memory Parallel Programming, *The MIT Press*, London.

51. İnternet: Pachghare, S. OpenMP Schedule Clause Parallel Matrix Multiplication. *OpenSource*, URL: http://www.webcitation.org/query?url=http%3A%2F%2Fwww.opensourceforu.com%2F2013%2F01%2Fopenmp-schedule-clause-parallel-matrix-multiplication%2F&date=2019-02-10, Son Erişim Tarihi: 10 Şubat 2019.

52. Çelik, A., ve Özmen, A. (2009, 13 Mayıs) Dağıtık paralel sistemler hakkında kıyaslamalı bir çalışma: PVM ve MPI, *5. Uluslararası İleri Teknolojiler Sempozyumu (IATS'09)*, Karabük, Türkiye.

53. Brothkorb, A.,Hagen, T., and Saetra, M.L. (2013). Graphics processing unit (GPU) programming strategies and trends in GPU computing, *Journal of Parallel and Distributed Computing*, 73(1), 4-13.

54. Cook, S. (2013). CUDA Programming A Developer's Guide to Parallel Computing with GPUs, Elsevier, USA.

55. Kaçar A. (2018). *İyonkürede dalga yayılımı modellemesi için ışın izleme algoritmalarının grafik işlemci birimleri ile paralel işlenmesi*, Yüksek Lisans Tezi, Hacettepe Üniversitesi Fen Bilimleri Enstitüsü, Ankara.

56. Çetin, N. M. ve Hacıömeroğlu, M. (2013) GPU Hızlandırmalı Veri Demetleme Algoritmalarının İncelenmesi, *Online Academic Journal of Information Technology*, 4 (12), 19-59.

57. Kirk, D.B. and Hvu, W.W. (2010), Programming Massively Parallel Processors, Elsevier, USA

58. NVIDIA Performance Primitives (NPP) (2014). Version 6.5 June 20, Sayfa sayısı:3030.

59. Jamal B. (2016) *Bulanık Mantık Uyarlamalı İki Yanlı (Bilateral) Görüntü Filtresi Tasarımı,* Yüksek Lisans Tezi, Gazi Üniversitesi Fen Bilimleri Enstitüsü, Ankara.

ÖZGEÇMİŞ

Kişisel Bilgiler



Soyadı, adı	: AYDIN, Semra
Uyruğu	: T.C.
Doğum tarihi ve yeri	: 26.02.1974 Bandırma
Medeni hali	: Evli
Telefon	: 0 (312) 490 67 32
e-mail	: semraaydin@gazi.edu.tr

Eğitim Derecesi	Okul / Program	Mezuniyet Tarihi
Doktora	Gazi Üniversitesi / Elk-Bil. Eğitimi	Devam Ediyor
Yüksek Lisans	Gazi Üniversitesi / Elk-Bil. Eğitimi	2006
Lisans	Gazi Üniversitesi / Bilgisayar Sis. Öğrt.	1996
Lise	Bursa Ali Osman Sönmez Fen Lisesi	1992

İş Deneyimi, Yıl	Çalıştığı Yer	Görev
2011-Halen	MEB Gölbaşı Mesleki ve Teknik A. L.	Teknik Öğretmen
1997-2011	TED Ankara Koleji	Bilgisayar Öğrt.

Yabancı Dil

İngilizce

Yayınlar

1. Aydın, S., Samet R., and Bay O.F. (2018). Real-time parallel image processing applications on multicore CPUs with OpenMP and GPGPU with CUDA, *The Journal of Supercomputing*, 74(6), 2255-2275.

2. Aydin, S., Samet R., and Bay, O.F. (2017), August 26-27). Real-time mouth defects detection on military cartridge, Computer Science & Information Technology (CS & IT), *3rd International Conference on Artificial Intelligence and Soft Computing*, 149-162, Dubai, Birleşik Arap Emirlikleri.

3. Bay, O.F., Aydın, S., and Samet R. (2017, May 9-12). Implementation of defect detection on military cartridge with OpenMP, *5th International Conference on Advanced Technology & Sciences (ICAT'17)*, 292-297, İstanbul, Turkey.

4. Samet, R., Bayram, A., Tural, S., and Aydın S. (2016, July 6-8) Primer Defects Detection on Military Cartridge Cases, *2016 Nicograph International (NicoInt)*, The 15th Annual International Conference, Hanzhou, China.

5. Bay, O.F., Samet R., Aydın, S., Tural, S., and Bayram A. (2015, August). Performance Analysis of GPU-Based Parallel Image Segmentation Using CUDA, *International Conference on Advanced Technology & Sciences (ICAT'15)*, 426-429, Antalya, Turkey.

6. Samet, R., Bay, O.F., Korkut, İ., Sazlı, M.H., Tural, S., Aydın, S., ve Bayram, A. (2014, 25-27 Haziran). Gerçek Zamanlı Paralel Görüntü İşleme Tabanlı Kovan Muayene Sistemi, SAVTEK-2014, *7.Savunma Teknolojileri Kongresi*, Bildiriler, pp.1201-1208, ODTÜ, Ankara, Türkiye.

7. Aydin, S., Bay, O.F. (2009). Building a high performance computing clusters to use in computing course applications, *Procedia-Social and Behavioral Sciences*, 1 (1), 2396-2401.

Hobiler

Ahşap boyama, kitap okuma.



GAZİLİ OLMAK AYRICALIKTIR.