T.C.
BAHÇEŞEHİR ÜNİVERSİTESİ

HESAPLAMA AĞIRLIKLI ALGORİTMALARIN PROGRAMLANMASINDA GRAFİK İŞLEMCİ (GPU) KULLANIMININ İNCELENMESİ

Yüksek Lisans Tezi

ERSİN KUZU

İSTANBUL, 2014

T.C.

BAHÇEŞEHİR ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİ TEKNOLOJİLERİ

HESAPLAMA AĞIRLIKLI ALGORİTMALARIN PROGRAMLANMASINDA GRAFİK İŞLEMCİ (GPU) KULLANIMININ İNCELENMESİ

Yüksek Lisans Tezi

ERSİN KUZU

Tez Danışmanı: Doç. Dr. M. Alper TUNGA

İSTANBUL, 2014

T.C.
BAHÇEŞEHİR ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİ TEKNOLOJİLERİ

Tezin Adı: Hesaplama Ağırlıklı Algoritmaların Programlanmasında Grafik İşlemci (GPU) Kullanımının İncelenmesi
Öğrencinin Adı Soyadı: Ersin KUZU
Tez Savunma Tarihi: 01.09.2014

Bu tezin Yüksek Lisans tezi olarak gerekli şartları yerine getirmiş olduğu Fen Bilimleri Enstitüsü tarafından onaylanmıştır.

Doç. Dr. F. Tunç BOZBURA
Enstitü Müdürü

Bu tezin Yüksek Lisans tezi olarak gerekli şartları yerine getirmiş olduğunu onaylarım.

Doç. Dr. M. Alper TUNGA
Program Koordinatörü

Bu Tez tarafımızca okunmuş, nitelik ve içerik açısından bir Yüksek Lisans tezi olarak yeterli görülmüş ve kabul edilmiştir.

Jüri Üyeleri	İmzalar
Tez Danışmanı Doç. Dr. M. Alper TUNGA	
Üye Yrd. Doç. Dr. Tevfik AYTEKİN	
Üye Yrd. Doç. Dr. Y.Batu SALMAN	

ÖNSÖZ

Bu çalışmanın hazırlanmasında destek ve yardımlarını esirgemeyen tez danışmanım, değerli hocam Doç Dr. M. Alper TUNGA'ya bana karşı göstermiş olduğu sabrından ötürü, değerli hocam Uğur BOSTANCI'ya verdiği teknik bilgiler ve yönlendirmelerinden ötürü, abim Engin Kuzu'ya gösterdiği sabır ve sunduğu teknik destek ve imkanlardan ötürü, hayatım boyunca maddi ve manevi desteklerini esirgemeyen anneme ve babama sonsuz teşekkürü borç bilirim.

İSTANBUL, 2014 Ersin Kuzu

ÖZET

HESAPLAMA AĞIRLIKLI ALGORİTMALARIN PROGRAMLANMASINDA GRAFİK İŞLEMCİ (GPU) KULLANIMININ İNCELENMESİ

Ersin Kuzu

Bilgi Teknolojileri

Tez Danışmanı: Doç. Dr. M. Alper TUNGA

Ağustos 2014, 86 Sayfa

Teknoloji ile birlikte ihtiyaçların artmaya devam edeceği bir gerçektir. Dolayısıyla yüksek işlem gücüne ihtiyaç duyan uygulama alanlarında tek bir işlem birimi asla yeterli olmayacaktır. Geçmişte ve günümüzde problemler birden fazla işlem birimine paylaştırılarak ortaklaşa çözülmesi yöntemi ile toplam işlem gücü artırılmaya çalışılmaktadır. Merkezi işlem birimleri (yani CPU'lar) yaygın ve ucuz oldukları için bir tür işlemci çiftliği oluşturmak amacıyla ciddi olarak tercih edildiler. Bunlar, sadece hesaplama konusunda uzmanlaşmış donanımlar değildirler. Yakın geçmişte oyun sektörünün hızla büyümesi grafik işlemcilerin gelişimi için itici güç oldu. Diğer yandan grafik işlemciler tamamen hesaplama üstüne uzmanlaşmış donanımlardır. Bunların genel amaçlı olarak programlanabilmesi (GPGPU) ile birlikte hesaplama dünyası yeni bir donanıma daha sahip oldu. Çok çekirdekli yapısı, hesaplama odaklı mimarisi ve komut seti, yüksek bellek transfer hızları, üreticiden bağımsız yazılım geliştirmeye olanak sağlayan endüstri standardı programlama dili (OpenCL) ile öne çıkmaktalar. Düşük maliyeti ise ayrı bir tercih sebebidir.

Sunulan çalışmada, OpenCL iş parçacıkları ile gerçekleştirilen hesaplama ağırlıklı bazı algoritmaların CPU ve GPU üzerinde başarım analizi gerçekleştirilmiştir. Deneysel sonuçlar, 2 çekirdekli CPU (Intel Core 2 Duo T9550), 448x2 çekirdekli NVIDIA GPU (2 x Nvidia Tesla M2050) ve 1600x2 çekirdekli ATI GPU (Ati Radeon HD 5970 X2), OpenCL genel programlama platformu teknolojisini destekleyen donanımlara sahip bilgisayarlar kullanılarak elde edilmiştir. Bu amaçla 2 çekirdekli CPU hesaplama sonuçları referans alınarak hızlanma, performans, enerji tüketimi ve maliyet grafikleri karşılaştırmalı olarak elde edilmiştir. Sonuçlar yaklaşık olarak hesaplama ağırlıklı algoritmalarda en az 13 kat, en fazla 1001 kata kadar hızlanma, gelişmiş şifre kurtarma algoritmalarında en az 152 kat, en fazla 905 kata kadar hızlanma, 1190 kata kadar performans artışı, 119 kata kadar enerji tasarrufu, 618 kata kadar daha düşük maliyet elde edilmiştir.

Anahtar Kelime: Genel Amaçlı GPU Programlama (GPGPU), OpenCL, GPU Hesaplama, Yüksek Başarımlı Hesaplama (HPC), Gelişmiş Şifre Kurtarma

ABSTRACT

INVESTIGATION OF GPU USE IN PROGRAMMING COMPUTATIONAL BASED ALGORITHMS

Ersin Kuzu

Information Technology

Thesis Supervisor: Assoc. Dr. M. Alper TUNGA

August 2014, 86 Pages

It is obvious that the expectations will continue to increase along with the technology. Therefore, a single processing unit will never adeqaute in application areas in which high processing features are needed. The problems are shared with multiple processing units in both past and future to increase the total processing capacity. Because central processing units are cheap, it is preferred to use them to construct processor farms. In the recent past, the rapid growth of the game industry was the driving force for the development of the graphics processors. On the other hand, graphics processors are completely specialized hardware on computational calculations. Programming these graphics processors for general purposes (GPGPU) lets this hardware to be used in any kind of calculations. The standard programming language (OpenCL) independent of vendors lets these processors to be used as multi-core structure, calculation-oriented architecture and instruction set. Low cost is another feature to choose them.

In this study, the performance analysis of some compute-intensive algorithms was carried out on CPU and GPU with OpenCL threads. Experimental results were obtained using computers equipped with 2-core CPU (Intel Core 2 Duo T9550), 448x2-core NVIDIA GPU (2xNvidia Tesla M2050), 1600x2-core ATI GPU (Ati Radeon HD 5970 X2) and which support OpenCL general programming platform technology. For this purpose, with the reference of 2-core CPU calculation results, the acceleration, performance, power consumption and cost graphs were obtained. According to the results; at least 13 times, up to 1001 times acceleration is obtained in approximately compute-intensive algorithms. In advanced password recovery algorithms, at least 152 times, up to 905 times acceleration, up to 1190 times performance increase, up to 119 times energy saving, and up to 618 times lower cost were obtained.

Keywords: General-Purpose GPU Programming (GPGPU), OpenCL, GPU Computing, High Performance Computing (HPC), Advanced Password Recovery

İÇİNDEKİLER

TABLOLAR……………..………………………………………………………….ix

ŞEKİLLER………………..………………………………………………………x

KISALTMALAR…………….....……………………………………..……………..xii

1. GİRİŞ………………………………………….……………………………...…1

2. CPU – GPU MİMARİSİ……………………………..……..……………………3

3. PARALEL HESAPLAMA…………….....……………………………..……………..4

3.1 SERİ VE PARALEL YAKLAŞIM…...……………..…………………………4

3.2 FLYNN SINIFLANDIRMASI.............…………………..……………..6

3.2.1 SISD – Tek Komut Tek Veri……….……………………….…………………7

3.2.2 SIMD – Tek Komut Çoklu Veri……………………………….………………7

3.2.3 MISD – Çoklu Komut Tek Veri……………………...…...…………………8

3.2.4 MIMD – Çoklu Komut Çoklu Veri……………………...……….........…9

3.3 PARALEL BİLGİSAYAR HAFIZA YAPISI…......…………………………..10

3.3.1 Paylaşımlı Bellek………………………..………………..…………………10

3.3.2 Dağıtık Bellek……………………………………………......………………..11

3.3.3 Karma Bellek…………………………………….…………….………………12

3.4 DAĞITIK HESAPLAMA……………………....…...……………………….13

3.5 ÖBEK HESAPLAMA………….....………………......………………………….13

3.6 GRİD HESAPLAMA……….....………………..…………………………….14

3.7 PARALEL PROGRAMLAMA MODELLERİ...…………………...…………..15

4. GPGPU………………………..………………………………………….……16

4.1 GRAFİK KARTLARI…………………...…………….…………………………16

4.2 GRAFİK İŞLEM BİRİMİ MİMARİSİ……………………...………………..16

4.3 GPGPU KAVRAMI……….....………………………………………………20

4.4 GPGPU PROGRAMLAMA MODELİ………......................………………..21

4.5 GPGPU PROGRAMLAMA DİLLERİ……..……………...…………………23

5. OPENCL…………………………………….....………………………………..26

5.1 OPENCL ANATOMİSİ……………………………………26

5.1.1 Dil Spesifikasyonu……………………………………26

5.1.2 Platform Katmanı API……………………………………27

5.1.3 Çalışma Zamanı API……………………………………27

5.2 OPENCL TERİMLERİ……………………………………27

5.2.1 Aygıtlar……………………………………27

5.2.2 Çekirdek Fonksiyonları……………………………………27

5.2.3 Çekirdek Nesneleri……………………………………27

5.2.4 Programlar……………………………………28

5.2.5 Bağlantılar……………………………………28

5.2.6 Program Nesneleri……………………………………28

5.2.7 Komut Kuyrukları……………………………………28

5.2.8 Ev Sahibi Programlar……………………………………28

5.2.9 Bellek Nesneleri……………………………………28

5.3 OPENCL ÇALIŞMA MODELİ……………………………………29

5.3.1 OpenCL Platform Modeli……………………………………29

5.3.2 OpenCL Yürütme Modeli……………………………………29

5.3.3 OpenCL Bellek Modeli……………………………………31

5.3.4 OpenCL Programlama Modeli……………………………………32

5.4 ÖRNEK – VEKTÖR EKLEME ÇEKİRDEĞİ……………………………………32

5.5 OPENCL'E GİRİŞ……………………………………34

5.5.1 Host Uygulaması Geliştirmek……………………………………35

5.5.1.1 Poker kart oyunu……………………………………35

5.5.1.2 Beş veri yapısı……………………………………36

5.5.1.3 Benzetmedeki noksanlar……………………………………37

5.5.2 OpenCL Çekirdekleri……………………………………38

5.5.2.1 Matematik öğrencileri okulda……………………………………38

5.5.2.2 Bir aygıtta çekirdek işlemi……………………………………40

5.5.3 Örnek Host Uygulaması……………………………………42

5.5.4 Örnek Çekirdek Uygulaması……….…………..……....………………44

6. ÖRNEK UYGULAMALAR………..…………………………..…………46

6.1 BİNOM OPSİYON (BINOMIAL OPTION) FİYATLAMA MODELİ……49

6.2 ÖZDEĞERLER (EIGEN VALUE) PROBLEMİ....……………..……………50

6.3 FLOYD – WARSHALL ALGORİTMAASI…....……………………………51

6.4 K – MEANS KÜMELEME (CLUSTERING) ALGORİTMASI………..…..52

6.5 MANDELBROT KÜMESİ VE FRAKTAL SİMÜLASYONU…..………….53

6.6 MATRİS ÇARPMA…….………..………………..………..……………54

6.7 MONTE CARLO SİMÜLASYONU İLE OPSİYON FİYATLAMA……….55

6.8 CİSİM (N - BODY) PROBLEMİ……..……......……………………………56

6.9 KONVOLÜSYON İŞLEMİ (SIMPLE CONVOLUTION)………....……..57

6.10 AES ŞİFRELEME ALGORİTMASI…………………………..….………58

6.11 MD5 ALGORİTMASI………....……………………………………….59

6.12 SHA1 ALGORİTMASI……….....………………..………………………60

6.13 NTLM ALGORİTMASI……...…..…………………………………..61

7. SONUÇ………………..…………………………………………………….63

KAYNAKÇA………………...……………………………………...……65

TABLOLAR

Tablo 6.1: Hesaplama aygıtlarının özellikleri…..………………..........………………46

Tablo 6.2: Test sistemlerinin özellikleri……………………..……...…………………47

Tablo 6.3: Programcı sayıları………………………………………..…………………47

ŞEKİLLER

Şekil 2.1: CPU – GPU mimari yapısı……..………………...…………………………3

Şekil 3.1: Seri hesaplama………………...……………………………...…………4

Şekil 3.2: Paralel hesaplama……………………...…………….……………………..5

Şekil 3.3: Flynn Taksonomisi……………………...………………………………7

Şekil 3.4: SISD Mimarisi………………………….....……………………………..7

Şekil 3.5: SIMD Mimarisi…...……………………...……………………….………..8

Şekil 3.6: MISD Mimarisi………...…………………...………………..…………9

Şekil 3.7: MIMD Mimarisi……………………………..………..…………………9

Şekil 3.8: Paylaşımlı Bellek (UMA)………………………………..……….…….10

Şekil 3.9: Paylaşımlı Bellek (NUMA)……………………………...…..………………11

Şekil 3.10: Dağıtık Bellek………...……………………..………..…………………12

Şekil 3.11: Karma Bellek……………………………..…………...………………13

Şekil 4.1: Intel CPU – NVIDIA GPU işlem hızları………………..……...…..……..16

Şekil 4.2: Intel CPU – NVIDIA GPU bellek bant genişlikleri…………...…………17

Şekil 4.3: GPU işlem hattı mimarisi……………...……………………………………18

Şekil 4.4: İşlem hattında veri dönüşümü………………..…………...………………19

Şekil 4.5: NVIDIA GeForce 6800 blok diyagramı……………………......…………20

Şekil 4.6: DirectX 10 programlanabilir işlem hattı mimarisi……………………….…….21

Şekil 4.7: CUDA dili programlama modeli yapısı…………………….……….....….22

Şekil 4.8: Matris toplama işleminin C'de ve CUDA'da gerçeklenmesi……….....…….23

Şekil 5.1: OpenCL Platform Modeli……………..………………………………..30

Şekil 5.2: NDRange indeks alanı, iş grupları ve iş öğeleri…………………..………31

Şekil 5.3: OpenCL Bellek Modeli…...………………...…………………….……32

Şekil 5.4: Oyun masası………………..………………………………………..36

Şekil 5.5: Host uygulamasının çalışma şekli…………………………...…...……….37

Şekil 5.6: Okul yapısı…………………………...……………………………...….39

Şekil 5.7: OpenCL Aygıt Yapısı…..……………………...……………………......41

Şekil 6.1: Performans (GFlops) test sonuçları…………………………...………….48
Şekil 6.2: Maliyet ($) test sonuçları………………………...………………………..48
Şekil 6.3: Enerji tüketim (Watt) test sonuçları…………………………..…..………49
Şekil 6.4: Binom Opsiyon Fiyatlama test sonuçları……………………..……..……50
Şekil 6.5: Özdeğerler test sonuçları………………………………………..……..…..51
Şekil 6.6: Floyd – Warshall test sonuçları……………………………….…..……….52
Şekil 6.7: K – Means Kümeleme test sonuçları………………………….…………..53
Şekil 6.8: Mandelbrot Fraktal Simülasyonu test sonuçları…………………...……...…54
Şekil 6.9: Matris Çarpımı test sonuçları……………………..………….......……….55
Şekil 6.10: Monte Carlo Simülasyon test sonuçları………………………...……….56
Şekil 6.11: Cisim test sonuçları…………………………..……………..…...……….57
Şekil 6.12: Konvolüsyon İşlemi test sonuçları……………………………..…..…....58
Şekil 6.13: AES Şifreleme test sonuçları…………………………………….…………59
Şekil 6.14: SHA1 şifreleme yapısı……………………………………………..…..…..61
Şekil 6.15: MD5, SHA1, NTLM test sonuçları……………...……………...……….62

KISALTMALAR

API	:	Application Programming Interface
CPU	:	Central Processing Unit
CUDA	:	Compute Unified Device Architecture
GFLOPS	:	Giga Floating – point Operations Per Second
GPGPU	:	General Purpose Computing on Graphical Processing Unit
GPU	:	Graphics Processing Unit
MIMD	:	Multiple Instruction, Multiple Data
MISD	:	Multiple Instruction, Single Data
NUMA	:	Non – Uniform Memory Access
OpenCL	:	Open Computing Language
OpenGL	:	Open Graphics Library
SIMD	:	Single Instruction, Multiple Data
SISD	:	Single Instruction, Single Data
SMP	:	Symmetric Multiprocessing
UMA	:	Uniform Memory Access

1. GİRİŞ

IBM ve Intel 1980'li yıllarda Grafik İşlemci geliştirmeye başladılar. 1990'larda yeni grafik kartı geliştiren S3 Graphics, Nvidia, Ati gibi markalar piyasaya girdi. Grafik programlama kütüphanesi OpenGL (OpenGL (Open Graphics Library), 2014) grafik programlama standardı kullanılarak 2 boyutlu grafikler donanımsal olarak hızlandırılmaya başlandı.2000'lerde GPU'nun hesaplama performansı çok artmıştı. GPU'lar gerçek zamanlı ve yüksek çözünürlüklü 3 boyutlu grafikler işleyebilmek için değişime uğramışlardır. Bugünlerde ise yüksek hesaplama gücü, paralel programlanabilir, çok çekirdekli işlemcili, çok iş parçacıklı, programlanabilir, çok yüksek bellek bant genişliği sunan GPU'lar piyasada mevcuttur.

Grafik kartlarındaki gelişim işlemcilere oranla çok daha hızlı olmaktadır. Grafik kartları yoğun matematiksel hesaplamalar yapmak için tasarlanmışlardır. Genellikle bu hesaplamalar grafik canlandırmaları ve oyunlar üzerinde yapılmaktadır. Grafik kartlarındaki yüksek paralel hesaplama gücünden genel amaçlı programlamalar da yararlanmaya başlamıştır.

Genel amaçlı mühendislik ve bilimsel uygulamaların başarımlarının arttırılması için paralel işlemlerin GPU, seri işlemlerin CPU üzerinde hesaplanarak oluşturdukları heterojen yapıya "Genel Amaçlı GPU Programlama veya Hesaplama", yani GPGPU denir. Uygulamaların hesaplama ağırlıklı kısımları GPU üzerinde paralel olarak hesaplanır, diğer kısımları ise CPU üzerinde seri olarak çalıştırılır. Seri işlemler için tasarlanmış CPU'lar birkaç çekirdekten oluşurken, yüksek derecede paralel işleme için optimize edilmiş GPU'lar etkili ve küçük yüzlerce çekirdekten oluşmaktadır. Bundan ötürü GPU ve CPU'nun oluşturduğu heterojen yapıyı verimli ve güçlü kılmaktadır (Nvidia, 2014).

Genel amaçlı programların da GPU'lardan faydalanabilmesi için grafik API'leriyle (Uygulama Programlama Arayüzü) genel amaçlı uygulamalar geliştirilmeye çalışılmıştır. CPU'dan çok faklı olan GPU'nun mimari yapısı ve programlama modeli grafik API'leriyle bir problemi çözmeyi ve CPU'ya göre yazılmış bir seri kodu GPU'ya uyarlamayı çok zorlaştırmaktadır. Gelişmelere paralel olarak kullanıcı dostu arayüzler ortaya çıkmıştır;

a. Lib Sh (Lib Sh – Embedded Metaprogramming Language, 2014)
b. C for Graphics (Cg) (NVIDIA Developer, 2014)
c. Close to Metal (AMD's Close-to-the-Metal, 2014)
d. BrookGPU (BrookGPU, 2014)
e. DirectCompute (Compute Shader Overview, 2014)
f. OpenCL (OpenCL™ (Open Computing Language) Zone, 2014)
g. C++ AMP (C++ AMP (C++ Accelerated Massive Parallelism), 2014)
h. CUDA (CUDA (Compute Unified Device Architecture), 2014)

Fakat bu GPGPU programlama dilleri üretici marka veya aygıt modellerine özel olmaları nedeniyle heterojen aygıtların bulunduğu ortamlarda kullanılamamaktadırlar. Ayrıca bu programlama dillerinin özellikleri gereği, bu diller ile programlama yapan geliştiricilerin dile özel veya programladıkları aygıt mimarisine özel detaylara hakim olmaları gerekmektedir.

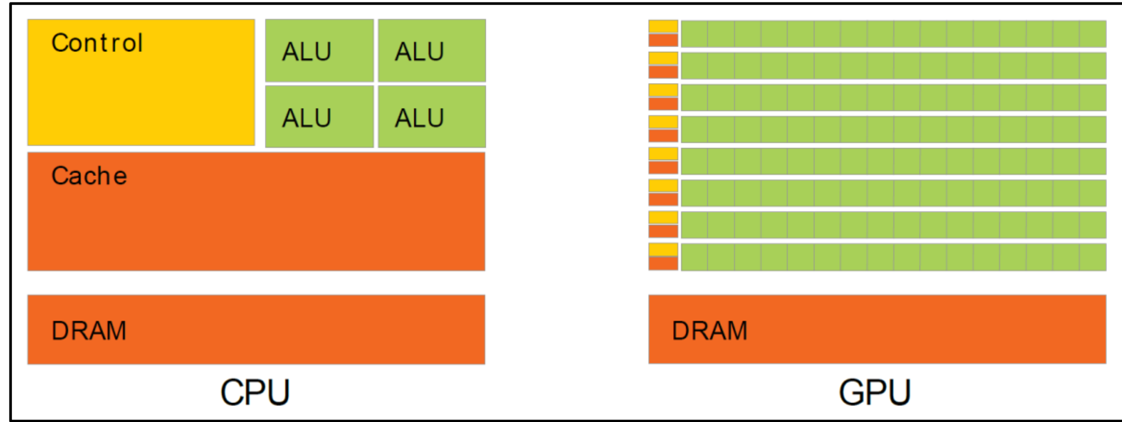
Bu problemlerin üstesinden gelmek için Apple, IBM, Intel, AMD ve NVIDIA gibi ana üreticilerin katkısı ve işbirliği ile kar amacı gütmeyen bir proje olarak Khoronos Group tarafından OpenCL çatısı (OpenCL Framework) geliştirilmiştir (OpenCL, 2014). OpenCL çatısı, C tabanlı programlama dili ve genel geçer bir uygulama programlama arayüzü (API- Application Programming Interface) barındırmaktadır. OpenCL çatısı ile programcıların çeşitli üreticilere ait, çeşitli modellerdeki CPU ve GPU aygıtlarının bulunduğu heterojen ortamlarda çalışabilen ve taşınabilir programlar yazmaları mümkün kılınmıştır. Her üretici OpenCL standartlarının uygulamasını kendine özel yapmasına rağmen, OpenCL veri tipleri ve API fonksiyonları imzalarında (method signatures) ortak bir standart sağlandığı için OpenCL çatısı ile gerçeklenen programlarla üretici ve aygıt bağımsız çalışabilmektedir. Bu ortak standartlar sayesinde OpenCL, geliştiricileri üreticiye özel veya aygıta özel teknik detaylardan da soyutlamaktadır. Bu nedenlerden dolayı OpenCL birçok üretici tarafından desteklenen ve giderek daha yaygın olarak kullanılan bir GPGPU çatısı haline gelmektedir.

2. CPU – GPU MİMARİSİ

GPU'ların işlemcileri özel olarak 3 boyutlu görüntüleri işlemek için tasarlanmıştır. Grafik uygulamalarının günümüzde yaygınlaşması, GPU'ların gelişmesini tetiklemiştir. GPU'ların asıl üretim amaçları grafik işlemlerini hesaplamaktır. GPU'lar CPU'lardan ön bellek ve akış kontrolü açısından üstünlüğü yerine yüksek kapasitede işlem yapmak için tasarlanmıştır. GPU üzerinde çok sayıda iş parçacığı çalıştırılarak hesaplanan piksel ve matris işlemleriyle 3 boyutlu görüntüler elde edilmektedir. GPU üzerinde koşut uygulamaların geliştirilmesine imkan veren, büyük veri kümelerinin işlenebilmesi ve maliyetin düşük olmasıdır.

Şekilde 2.1'de CPU – GPU mimari yapısı verilmiştir (NVIDIA, 2014). GPU ön belleklerinin düşük olmasına rağmen koşut veri işleme için tasarlanmalarından ötürü, çekirdeklerinde çok fazla aritmetik mantık birimi (ALU) içermektedir. Yapıları gereği GPU'lar CPU'lara nazaran çok fazla transistör içerir. Test donanımlarımızdan ATI Radeon HD 5970 X2 grafik kartında 4.3 milyar transistör, Intel Core 2 Duo T9550 işlemcisinde 410 milyon transistör bulunmaktadır. Grafik kartında işlemciye göre 10x daha fazla transistör bulunmaktadır.

Şekil 2.1: CPU – GPU mimari yapısı



Kaynak: (NVIDIA, 2014)

GPU'ların sahip olduğu mimarinin şekli SIMD (single instruction multiple data) mimari yapısında tasarlanmıştır. Bu mimari yapı çok sayıda veri kümesinde aynı uygulamayı çalıştırma imkanı sunmaktadır. Fakat bu mimari yapı problemlere etkili ve hızlı çözümler üretememektedir. Akış kontrolünün yoğun olduğu uygulamaların performansında artış beklenmemelidir. GPU mimarisi yapısı SIMD'a göre hazırlanmış komut kütüphanesi bulunan bir derleyici bulunmamaktadır. Bu da GPU için uygulama geliştirmeyi güçleştirmektedir. Bu zorluk koşutlaştırma işlemlerinin programcılar tarafından yazılmasını gerektirmektedir. Bilgisayar bilimlerindeki en popüler araştırmalardan biri de SIMD mimari yapısına uygun komutlar üretmektir. Buna vektörizasyon denir.

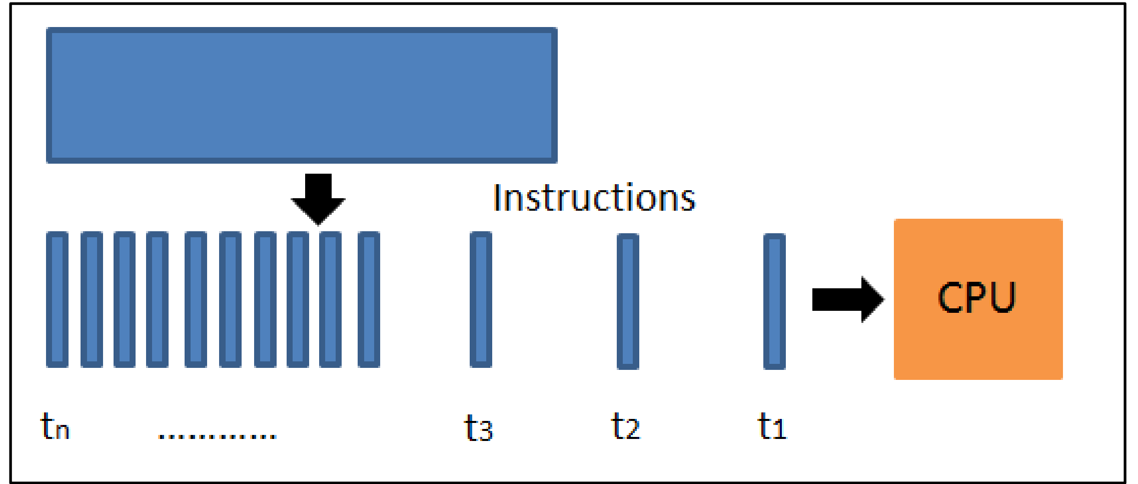
3. PARALEL HESAPLAMA

Paralel hesaplama, parçalara bölünmüş uygun yapıdaki bir görevin kısa sürede sonuçlandırılabilmesi için birçok çekirdekten oluşan işlemciler üzerinde eş zamanlı işlenmesidir. Bu yöntem çözüm aşamasında problemin küçük iş parçacıklarına bölünerek eş zamanlı koordine edilmesidir. Paralel hesaplama beraberinde performans artışı ve kısa sürede çözüm getirir. Bundan ötürü bilimdeki gelişmelerin paralel hesaplamaya ihtiyacı ortaya çıkmıştır. Örneğin, tek bir makine üzerinde matematiksel hesaplama yapmak uzun ve zor bir iştir. Bu makinenin fiziksel özellikleri arttırıldığında daha iyi sonuç alınır. Ancak fiziksel özelliklerin güçlendirilmesi sınırlıdır. Bundan dolayı paralel hesaplama öne çıkmıştır.

3.1 SERİ VE PARALEL YAKLAŞIM

Şekillerle bir problemin seri ve paralel yaklaşımda nasıl çözüldüğü gösterilmiştir. (Yan, K., 2014)

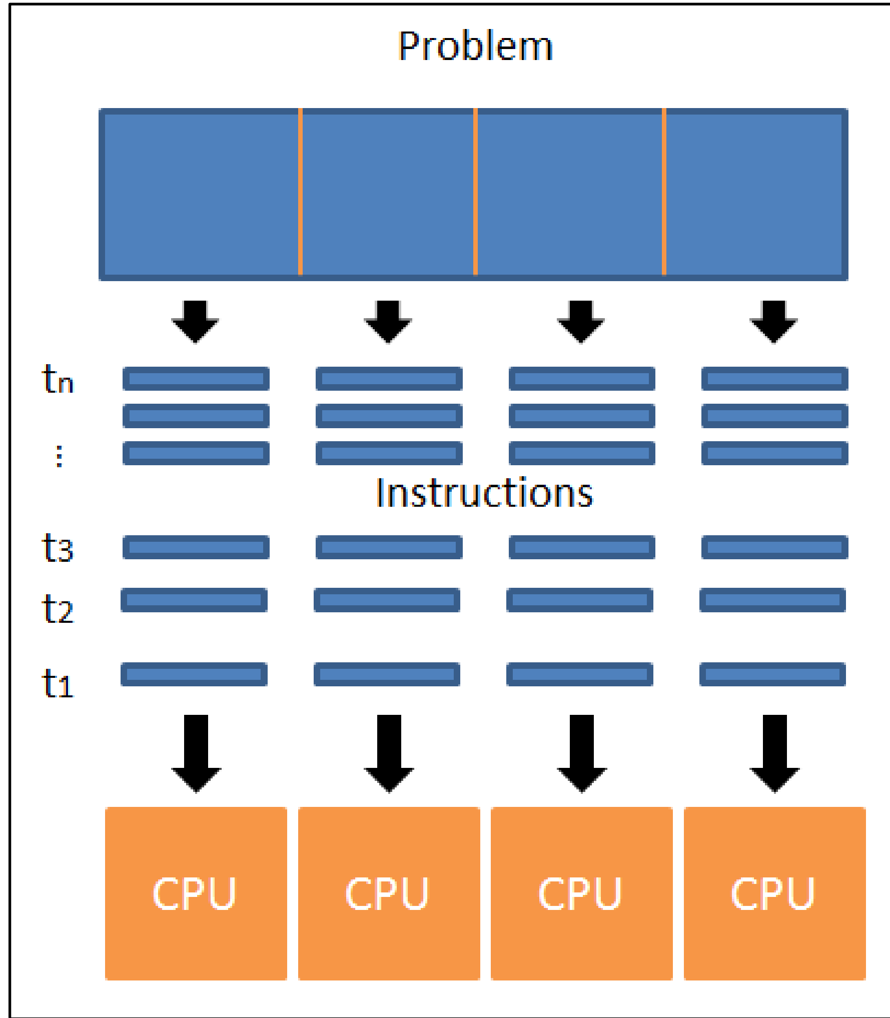
Şekil 3.1: Seri hesaplama



Kaynak: (Yan, K., 2014)

Seri Hesaplama: Problem her bir zaman aralığında bir komut olmak üzere CPU (Merkezi İşlem Birimi)'ne gönderilir.

Şekil 3.2: Paralel hesaplama



Kaynak: (Yan, K., 2014)

Paralel Hesaplama: Problem önce belli sayıda parçaya ayrılır. Örneğin, burada problem dört parçaya ayrılmıştır. Bundan sonra her bir problem parçası bir zaman aralığında bir komut olmak üzere farklı CPU'ya gönderilir.

İki sayının toplanmasının seri ve paralel yaklaşımla hesaplanması

Seri (1 işlemcili)

```
 1 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15 + 16
 2 3 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15 + 16
 3 6 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15 + 16
 4 10 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15 + 16
 5 15 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15 + 16
 6 21 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15 + 16
 7 28 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15 + 16
 8 36 + 9 + 10 + 11 + 12 + 13 + 14 + 15 + 16
 9 45 + 10 + 11 + 12 + 13 + 14 + 15 + 16
10 55 + 11 + 12 + 13 + 14 + 15 + 16
11 66 + 12 + 13 + 14 + 15 + 16
12 78 + 13 + 14 + 15 + 16
13 91 + 14 + 15 + 16
14 105 + 15 + 16
15 120 + 16
16 136
```

Paralel (8 işlemcili)

```
1 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15 + 16
2 10 + 26 + 42 + 58
3 136
```

Paralel hesaplamada hız artarken, çalışma zamanı azalır (Wilkinson, B., Allen, M., 2004). Seri hesaplama 16 adımda tamamlanırken paralel hesaplama 3 adımda tamamlanmaktadır. Bu örnekte 5,33x daha hızlı hesaplama sağlanmıştır. Daha fazla işlem olursa hızlanma kat kat artabilir.

3.2 FLYNN SINIFLANDIRMASI

Paralel bilgisayarlar sınıflandırılırken, farklı parametreler kullanılarak farklı sınıflandırmalar yapılmıştır. Ancak bunlardan en yaygın bilineni 1966'dan beri kullanılan ve Flynn Taksonomi olarak bilinenidir (Blaise Barney, Lawrence Livermore National Laboratory (LLNL), 2014). Michael J. Flynn, bu sınıflandırmayı yaparken parametre olarak komut (instruction) ve veriyi (data) seçmiştir. M. J. Flynn, bilgisayar sistemlerini aynı anda işlenebilen veri ve komut sayılarına göre dört gruba ayırmıştır. Bu bağlamda, Flynn Taksonomi'si şu şekildedir.

Şekil 3.3: Flynn Taksonomisi

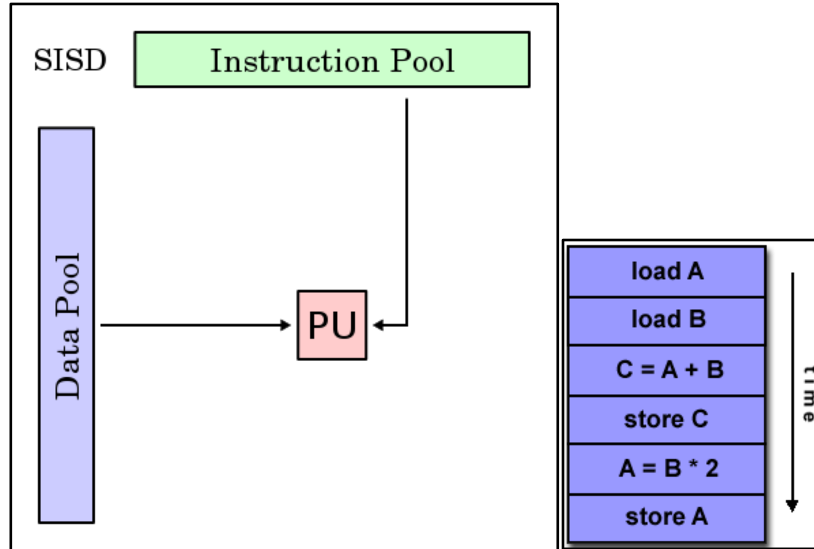
	Single Instruction	Multiple Instruction
Single Data	**SISD** (Traditional Uni-processor)	**MISD** (Voting schemes and active-active processing)
Multiple Data	**SIMD** (e.g. SSE 4.2, AltiVec, NEON, GP-GPU, Vector Instructions)	**MIMD** (Distributed systems with map-reduce (MPMD), Clusters with MPI, Mulit-Threaded Vector Processing (SPMD))

Kaynak: (IBM developerWorks, 2014)

3.2.1 SISD – Tek Komut Tek Veri (Single Instruction, Single Data)

Şekil 3.4'de görüldüğü gibi aynı anda tek bir komut, tek bir veriyi işlemektedir. Yani bilgisayarda bir işlemci (CPU) ve bir yönetim birimi (control unit) mevcuttur. Böyle bir yapı, klasik Von Neumann mimarisinin özelliğidir.

Şekil 3.4: SISD Mimarisi



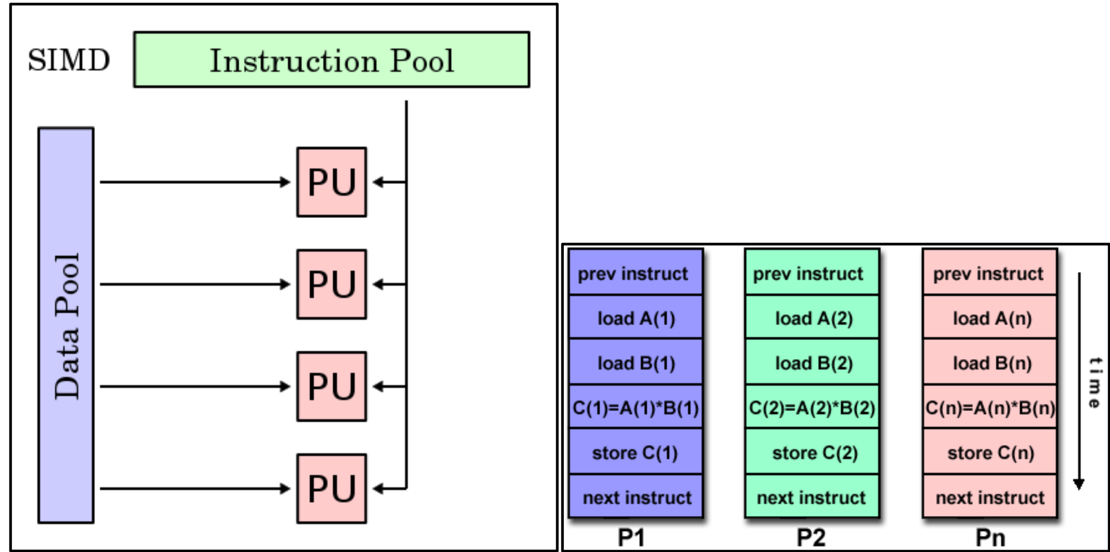
Kaynak: (Wikipedia, the free encyclopedia, 2014)

3.2.2 SIMD – Tek Komut Çoklu Veri (Single Instruction, Multiple Data)

Şekil 3.5'de görülen sistemde program yönetim birimi (control unit) tektir. İşlemci ise birden fazladır ve zamanın her anında bir komut birden fazla veri üzerinde işlem yapmaktadır. Bu tür mimarilere örnek olarak dizi veya matris işlemcisi gibi tasarlanmış olan ILLIAC – IV'ü göstermek mümkündür. SIMD yapılarının iki şekilde olduğunu söyleyebiliriz. Bunlardan birincisi dizi (matris veya vektör olarak da adlandırılır) bilgisayar olup her bir dizi elemanı üzerinde aynı işlemi yapar. İkincisi ise şekil 3.5'de görüldüğü gibi pipeline (boru hattı) olarak isimlendirilerek, çoklu veri,

boru hattına ardışık olarak gönderilmektedir ve zamanın her anında boru hattının çeşitli noktalarındaki çeşitli veriler üzerinde aynı veya farklı işlemler yapılmaktadır.

Şekil 3.5: SIMD Mimarisi

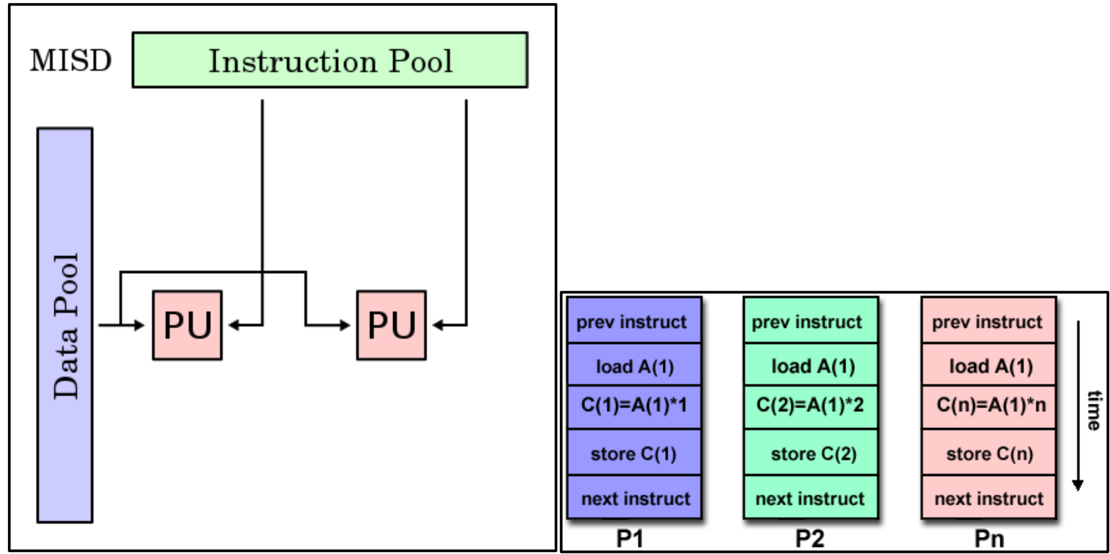


Kaynak: (Wikipedia, the free encyclopedia, 2014)

3.2.3 MISD – Çoklu Komut Tek Veri (Multiple Instruction, Single Data)

Şekil 3.6'da görüldüğü gibi bu yapıda birçok komut aynı anda bir tek veriyi işlemektedir. Yani, aynı anda bir işlemcideki veri üzerinde birden fazla yönetim biriminin (control unit) ürettiği çeşitli komutlar işlem yapmaktadır. Böyle mimariye sahip olan bir bilgisayar tasarlamak çok zordur. Çünkü bu türden işlemlere ihtiyaç duyulması başlı başına problemdir. Fakat bazı araştırmacılar boru hattı (pipeline) mimarisine MISD makine gibi bakmaktadırlar. Çünkü bu mimaride aynı veri üzerinde çeşitli işlemler yapmaya imkan sağlamaktadır.

Şekil 3.6: MISD Mimarisi

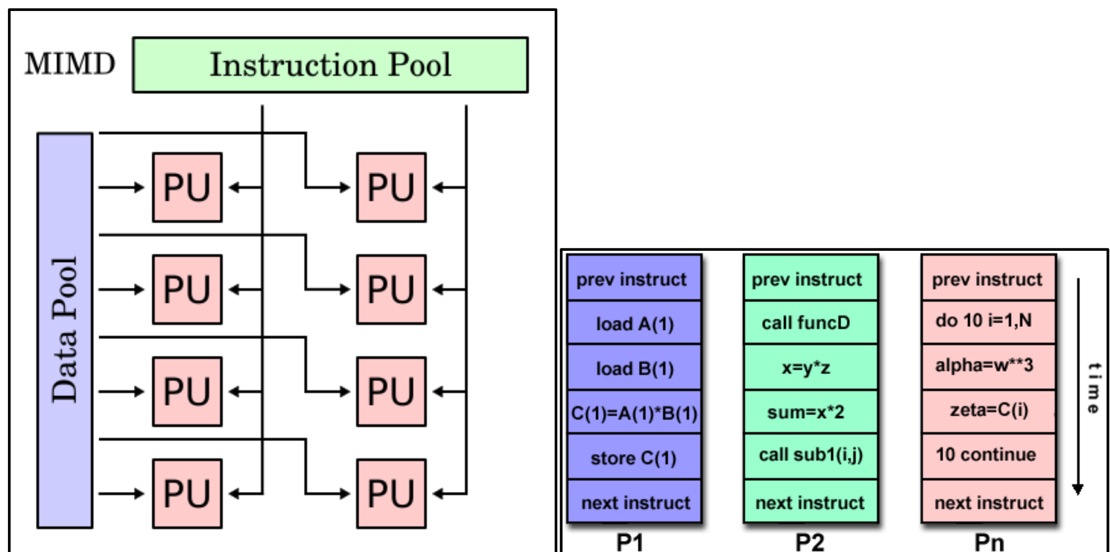


Kaynak: (Wikipedia, the free encyclopedia, 2014)

3.2.4 MIMD – Çoklu Komut Çoklu Veri (Multiple Instruction, Multiple Data)

Bu yapıda çeşitli işlemcilerde çeşitli veriler üzerinde çeşitli komutlara uygun işlemler yapılmaktadır. Yani, işlemci ve yönetim birimlerinin (control unit) sayısı şekil 3.7'de görüldüğü gibi birden fazla olmaktadır. İşlemciler ara sonuçları birbirine iletebilmektedirler. Aslında bu mimari daha önce sözünü ettiğimiz çok bilgisayarlı paralel sistemlere uymaktadır. Örnek olarak Cray – Z, Ncube (hypercube) vb. süper bilgisayarları gösterebiliriz.

Şekil 3.7: MIMD Mimarisi



Kaynak: (Wikipedia, the free encyclopedia, 2014)

Flynn'nin sınıflandırmasında görüldüğü gibi paralel işlem sistemleri SIMD ve MIMD olarak iki temel prensipte tasarlanmaktadır.

3.3 PARALEL BİLGİSAYAR HAFIZA YAPISI

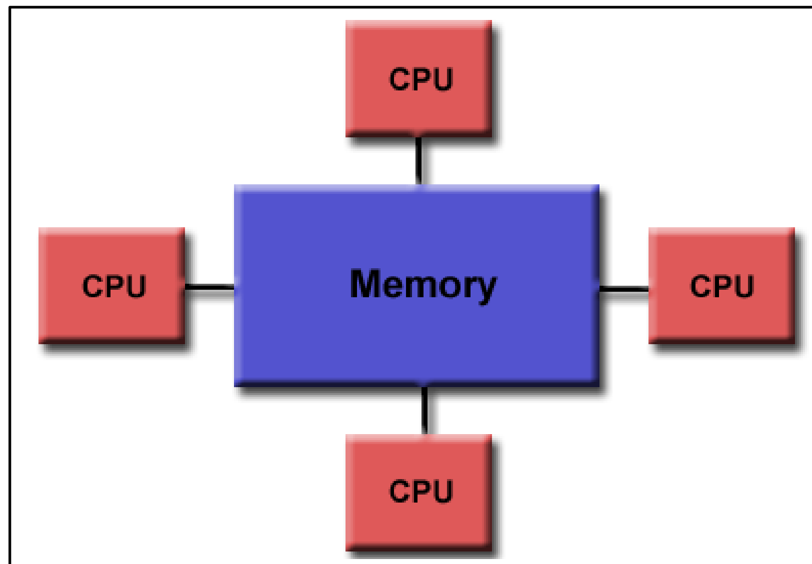
Paralel bilgisayar hafıza yapısı genel olarak üç şekilde yapılandırılır (Blaise Barney, Lawrence Livermore National Laboratory (LLNL), 2014). Bunlar paylaşımlı, dağıtık, melez paylaşımlı – dağıtık yapılardır.

3.3.1 Paylaşımlı Bellek (Shared Memory)

Paralel bilgisayarda çoğunlukla tüm işlemciler belleğe genel adres uzayından erişirler. Buna rağmen değişik paylaşımlı bellekler mevcuttur. Her işlemci kendi işlemlerini kendileri yapmalarına rağmen ortak bellek kaynağını kullanırlar. Bir işlemcinin bellek üzerindeki yaptığı değişikliği diğer işlemciler de görebilir. Paylaşımlı bellekli makineler iki temel sınıfa ayrılır. UMA (Uniform Memory Access) ve NUMA (Non – Uniform Memory Access).

Uniform Memory Access (UMA): UMA simetrik çoklu işlemcili (SMP – Symmetric Multiprocessor) bilgisayarlarda kullanılır. Dolayısıyla hem hafızaya erişimleri hem de erişim süreleri eşittir.

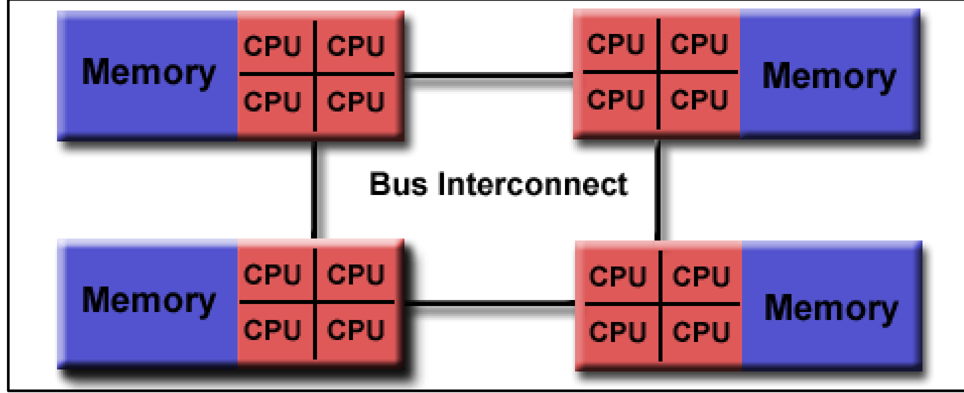
Şekil 3.8: Paylaşımlı Bellek (UMA)



Kaynak: (Blaise Barney, Lawrence Livermore National Laboratory (LLNL), 2014)

Non – Uniform Memory Access (NUMA): NUMA iki ya da daha fazla SMP tipi işlemcinin birbirine bağlanmasından meydana gelir. Bu durumda herhangi bir SMP diğer SMP'lerin belleklerine direkt ulaşabilir. Bu yapıda işlemcilerin her belleğe erişim süreleri aynı olmamaktadır.

Şekil 3.9: Paylaşımlı Bellek (NUMA)



Kaynak: (Blaise Barney, Lawrence Livermore National Laboratory (LLNL), 2014)

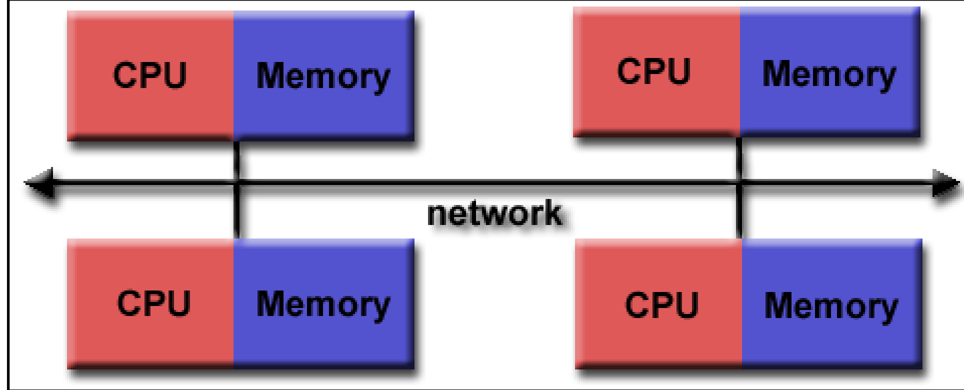
Programcılar için kullanıcı dostu olan paylaşımlı bellek yapısında bellek erişimi için genel adres uzayının kullanılması işlemciler arasındaki veri paylaşımını da hızlandırır. Buna karşın belleği paylaşan işlemci sayısı arttıkça bellek – işlemci trafiği çoğalacağından paylaşım hızı yavaşlayacaktır. Programcı belleğe erişim ve senkronizasyonundan sorumludur.

3.3.2 Dağıtık Bellek (Distributed Memory)

Dağıtık bellekli sistemlerde her işlemcinin kendi yerel hafızası mevcuttur. İşlemci diğer belleklerin adreslerini bilmediğinden bellek adresleri için genel bir adres uzayına gerek yoktur.

Dağıtık bellekli sistemlerde ise her makine kendi yerel hafızasına sahiptir ve makineler birbirlerine bir iletişim ağıyla bağlıdır. Hepsi bağımsız olarak işlem yapabilir. Yerel hafızada yapılan değişiklikler diğerlerini etkilemez. Dağıtık bellekli sistemde düşünülmesi gereken durum, bir işlemcinin diğer işlemci üzerindeki veriye ihtiyaç duyabileceğidir. Bunun için çözüm mesaj iletimidir. Dağıtık bellekli sistemin avantajı, işlemci sayısı arttırılarak kullanıcı için gerekli olan hafıza miktarının çoğaltılabilmesidir. Her işlemci ağ iletişimi olmaksızın kendine ait belleğe erişebilir. Buna karşın işlemciler arasındaki veri iletişiminden programcının kendisi sorumludur ve belleklerin hızları farklılık gösterebilir.

Şekil 3.10: Dağıtık Bellek

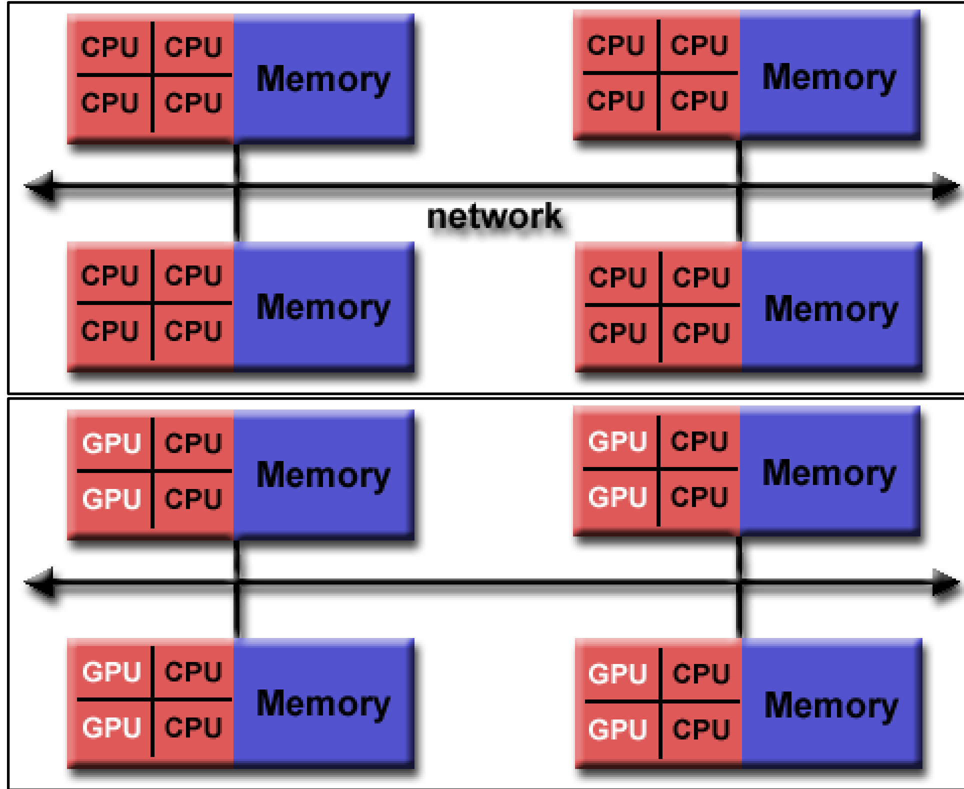


Kaynak: (Blaise Barney, Lawrence Livermore National Laboratory (LLNL), 2014)

3.3.3 Karma Bellek (Hybrid Distributed – Share Memory)

Karma bellek yapısı günümüzde daha çok kullanılmaktadır. Melez dağıtık – paylaşımlı sistemler günümüzde en büyük ve en hızlı sistemler (süper bilgisayarlar) için tasarlanmıştır. Her bellek için ayrılmış bir işlemci grubu vardır ve bu kısım paylaşımlı bellektir. Her işlemci – bellek grubu birbiriyle ağ üzerinden bağlıdır ve bu kısım dağıtık bellektir. Bu istem aynı zamanda dağıtık ve paylaşımlı bellek yapılarının avantaj ve dezavantajlarını da içermektedir.

Şekil 3.11: Karma Bellek



Kaynak: (Blaise Barney, Lawrence Livermore National Laboratory (LLNL), 2014)

Paylaşımlı bellek bileşeni bir paylaşımlı bellek makinesi ve/veya grafik işlem birimleri (GPU) olabilir. Dağıtık bellek bileşeni birden çok paylaşılan bellek ve/veya GPU içeren makinelerinden oluşan ağ olabilir.

3.4 DAĞITIK HESAPLAMA

Dağıtık hesaplama, büyük hesaplama problemlerinin parçalara ayrılıp, her parçanın birbirlerine bir bilgisayar ağıyla bağlı olan bir sistemdeki makineler üzerinde çözülmesidir (Wikipedia, the free encyclopedia, 2014). Dağıtık bir sistemde her makinenin kendi yerel hafızası vardır.

3.5 ÖBEK HESAPLAMA (CLUSTER)

Öbek bilgisayarlar, normalden daha iyi bir performans verebilmek için birbirine yakın olan bilgisayarların bir araya getirilmesiyle oluşturulan sistemlerdir. Yerel ağ içerisindeki bilgisayarlar birbirine bağlanarak tek bir makineymiş gibi davranırlar. Bu şekilde, tek bir makine üzerinde gerçekleştirilemeyecek işlemler çalıştırılan kodun paralelleştirilmesiyle öbek bilgisayarlar üzerinde kolaylıkla gerçekleştirilebilir.

Öbek bilgisayarlar şu şekilde sınıflandırılır (Wikipedia, the free encyclopedia, 2014):

a. **Yüksek Yararlanılabilir (High Availability Clusters):** Bunlar, öbeklerden sağlanan yararı arttırmak için oluşturulmuşlardır. Bu tür öbeklerde, bazı sistem bileşenleri çöktüğünde devreye boş düğümler (nodes) girer.
b. **Yük Dengeleyici Öbekler (Load – balancing Clusters**): Performansı arttırmak için oluşturulmuşlardır. Bu öbekler, bir veya daha fazla yük dengeleyici ön uçtan gelen iş yükünü arka uç sunucularına dağıtırlar.
c. **Yüksek Performanslı Öbekler (Compute Clusters):** Bu öbekler, düğümlerden gelen işi bölümlere ayırarak performans artışı sağlarlar. En bilinen yüksek performanslı öbek, Beowulf öbeğidir.

3.6 GRİD HESAPLAMA

Grid, dünyanın farklı coğrafyalarında yer alan kişisel bilgisayarların, küme bilgisayarların ya da süper bilgisayarların birbirlerine ağ üzerinden bağlanmaları ile yüksek performanslı hesaplama yapabilmek için oluşturulmuş yapıya verilen isimdir (TR-Grid, 2014). Öbek bilgisayarlardan temel farkı coğrafik olarak dağıtık olması ve heterojen bir yapıda olmasıdır. Bu sistem ile bilgisayarlar, yazılımlar, veritabanları paylaşılabilir. Grid kullanımıyla daha büyük ölçekli problemler çözülebilir. Kaynaklar daha verimli bir şekilde kullanılarak, araştırmacılar için daha etkin bir çalışma ortamı oluşturulur.

Grid yapılarda kullanıcı yapacağı işi sisteme gönderir. İşlemin yapılabilmesi için uygun kaynak seçilir. Daha sonra sonuç kullanıcıya gönderilir. Burada amaç tek bir sistem görünümü vermektir. Grid, verilerle dosya seviyesinde ilgilenir ve ayrıca veri yapılarıyla ilgilenmez. Depolama kaynaklarında saklanan grid dosyaları salt – okunur dosyalardır.

Grid sistemler için geliştirilmiş bazı araçlar ve uygulamalar vardır. En bilinen uygulamalardan biri SETI@home'dur (Wikipedia, the free encyclopedia, 2014). GridMiner (GridMiner, 2014) sistemi ise grid üzerinde veri madenciliği için geliştirilmiş bir yazılımdır.

Türkiye'de grid ile ilgili çalışmalar ULAKBİM tarafından 2003 yılında TR – Grid adı altında başlamıştır (TR-Grid, 2014). TR – Grid'in amaçları ulusal grid altyapısını kurmak ve kullanıcı kitlesini bilgilendirmek, çeşitli bölgesel uygulamalar geliştirmek, uluslararası grid projelerinde aktif görev almak olarak sayılabilir.

3.7 PARALEL PROGRAMLAMA MODELLERİ

Genel olarak paralel programlama modelleri şu şekildedir (Blaise Barney, Lawrence Livermore National Laboratory (LLNL)):

a. **Paylaşımlı Bellek Modeli (Shared Memory Model):** Bu modelde görevler ortak bir hafıza alanını paylaşırlar. Burada eş zamanlı olarak verilere ulaşmada karşılaşılacak sorunları engellemek için bazı yapılar kullanılır (semafor, kilit vb.)
b. **İş Parçacığı Modeli (Thread Model):** Bu modelde bir program önce seri olarak çalışmaya başlayıp daha sonra iş parçacıkları yaratılarak eş zamanlı olarak çalışmaya devam edebilir. Her iş parçacığı kendi yerel verisine sahiptir.
c. **İleti Geçme Modeli (Message Passing Model):** Burada gerçekleştirilecek iş parçalara ayrılarak düğümlere dağıtılır. Her düğüm kendi işi için gerekli olan verileri ileti geçme (send – receive) sayesinde alır. Bu iletim esnasında bir gönderici ve bir alıcı olması gerekir. Burada düşünülmesi gereken nokta, verinin doğru olarak alınıp alınmadığıdır.
d. **Veri Paralel Model (Data Parallel Model):** Bu modelde her düğüm işlenecek verinin bir kısmı üzerinde kendi görevini gerçekleştirir. Dolayısıyla bir mesaj iletimi söz konusu değildir.
e. **Melez Model (Hybrid Model):** Bu modelde herhangi iki ya da daha fazla paralel programlama modeli birlikte çalışır.

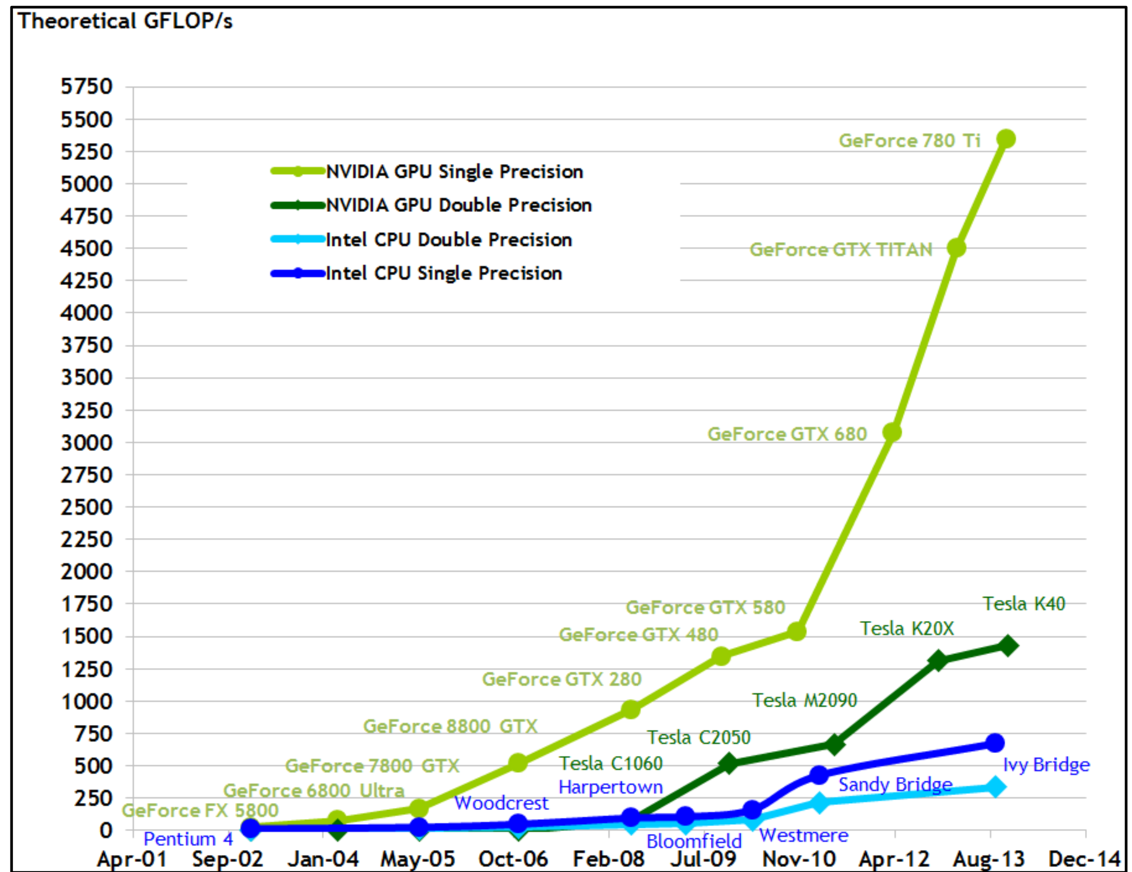
4. GPGPU

4.1 GRAFİK KARTLARI

Grafik kartları, bilgisayarlarda grafik ile ilgili işlemleri yüksek performanslı ve verimli bir şekilde gerçekleştirmek ve görüntüleme birimine çıktı üretmek için özelleşmiş kartlardır. Grafik kartları; aygıtın bilgisayar donanımıyla etkileşimini gerçekleştiren alt seviyedeki gömülü yazılımının bulunduğu bios, grafik kartı belleği, hesaplama ve komut yürütme işlemlerini gerçekleştiren grafik işlem birimi (GPU) kısımlarından oluşur. Grafik kartlarını merkezi işlem birimlerinden (CPU) farklı kılan temel özellik GPU’nun tek komut çoklu veri (Single Instruction Multiple Data – SIMD) yapısında çalışarak yüklü miktarda veriyi paralel bir şekilde işleyebilmesini sağlayan mimaridir.

4.2 GRAFİK İŞLEM BİRİMİ MİMARİSİ

Şekil 4.1: Intel CPU – NVIDIA GPU işlem hızları



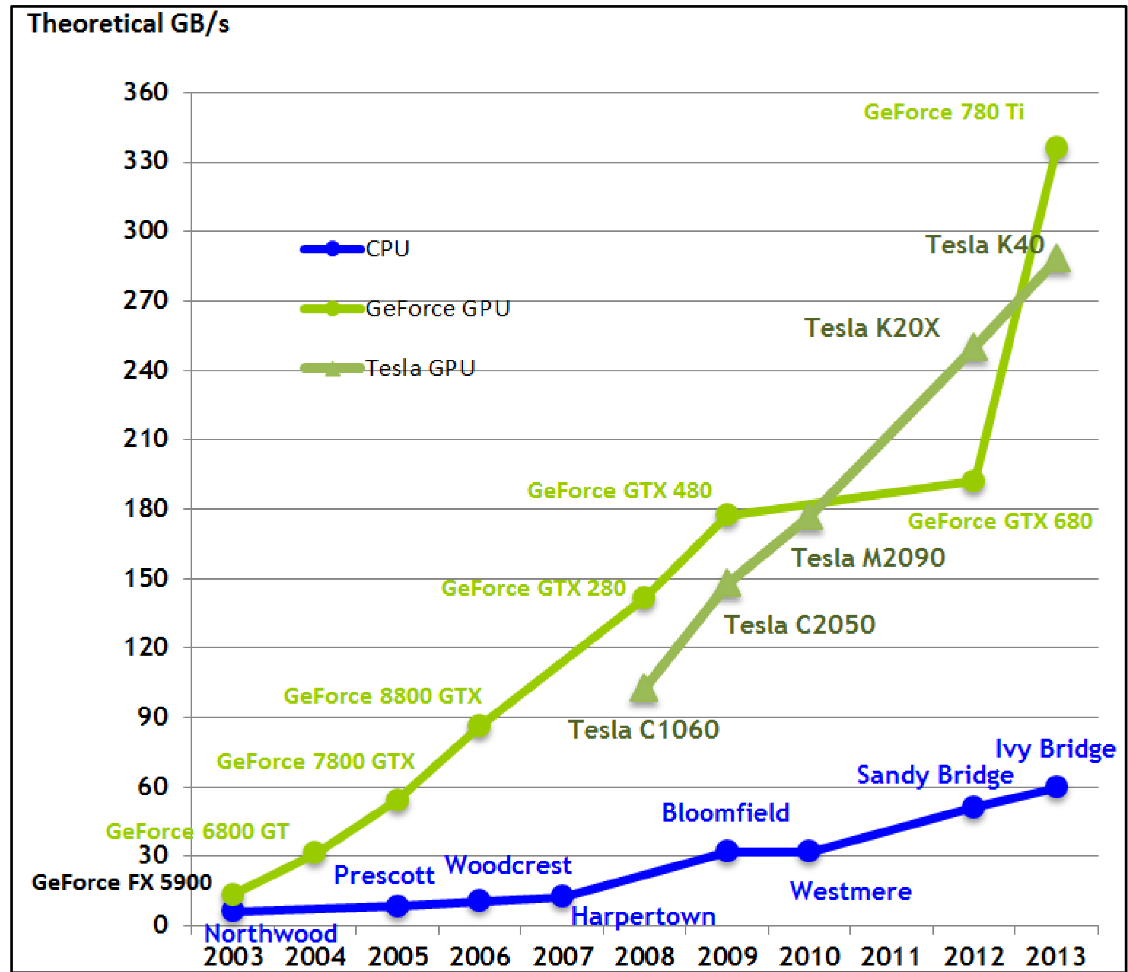
Kaynak: (NVIDIA, 2014)

Grafik işlem birimleri (GPUs), grafik uygulamalarındaki 3 boyutlu verinin gösterim için 2 boyuta dönüştürülmesi (3D to 2D transformation), düğümlerin birleştirilmesi (vertex assembly), parça oluşturma (fragmentation), parçaların piksellere

dönüştürülmesi, doku haritalama (textture mapping), gürültü giderme ve pürüzsüzleştirme (anti - aliasing) gibi çok miktarda veri üzerinde yoğun hesap gerektiren matris ve vektör işlemlerini yüksek hızda gerçekleştirebilmek üzere tasarlanmışlardır. Bu yüzden grafik işlem birimleri (GPUs), merkezi işlem birimlerine (CPUs) göre çok daha fazla sayıda çekirdek ve işlem hattına sahiptir. Günümüzde ev kullanıcıları tarafından kullanılan CPU'larda 2 ile 8 çekirdek bulunmaktayken GPU'larda çekirdek sayısı 80 ile 100 civarında olabilmektedir.

Şekil 4.1'de Intel CPU ve NVIDIA GPU'ların GFLOPS/s olarak işlem hızı bakımından son 12 yılın gelişimi gösterilmiştir (NVIDIA, 2014). Şekilde görüleceği üzere günümüzde GPU'lar işlem hızı bakımından CPU'lara göre çok daha ileridedir. Bu farkı sağlayan temel özellik, GPU'ların genellikle paralel olarak işlenebilir veriler üzerinde çalışması ve SIMD yapısında çalışabilen gelişmiş ve karmaşık işlem hattı mimarilerine sahip olmalarıdır. Şekil 4.2'de ise aynı donanımların bellek bant genişliği bakımından gelişimi gösterilmiştir (NVIDIA, 2014).

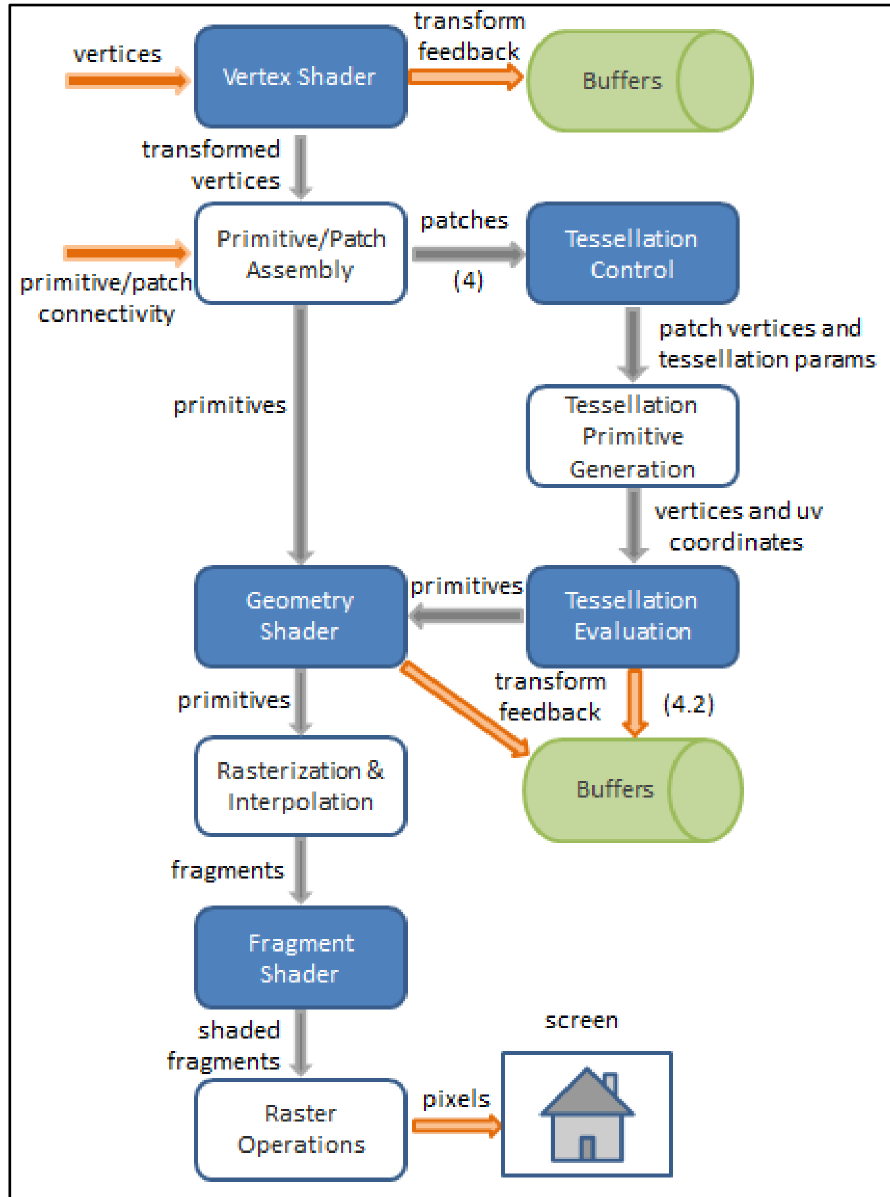
Şekil 4.2: Intel CPU – NVIDIA GPU bellek bant genişlikleri



Kaynak: (NVIDIA, 2014)

İşlem hattı birbiri ardına gerçekleşen çeşitli işlem aşamalarından oluşur. Her aşamada bir önceki aşamanın sonucu girdi olarak alınır ve işlenir. İşlem hattı sürecinde köşe kenar dönüşümü, ışıklandırma, birleştirme, piksellere dönüştürme, doku oluşturma, renk karışımları, transparanlık, derinlik katma ve gölgelendirme gibi işlemler yapıldıktan sonra oluşan veri, görüntüleme aygıtlarına aktarılır. İşlem hattında aşamaların birbiri ardına uygulanması esnasında, bir aşamada işlenen veri bloğu bir sonraki aşamaya aktarıldığı anda aynı aşamaya, işlenmesi için yeni bir veri bloğu girer. Bu şekilde aşamaların paralel işlemeye uygun olan veriler üzerinde paralel olarak çalışması sağlanmış olur. Bu işlem sonucunda grafik uygulamaları tarafından 3 boyutlu uzayda köşe, kenar ve renk bilgileri olarak aktarılan veri, görüntüleme aygıtı tarafından 2 boyutlu olarak görüntülenebilecek resimler haline getirilmiş olur.

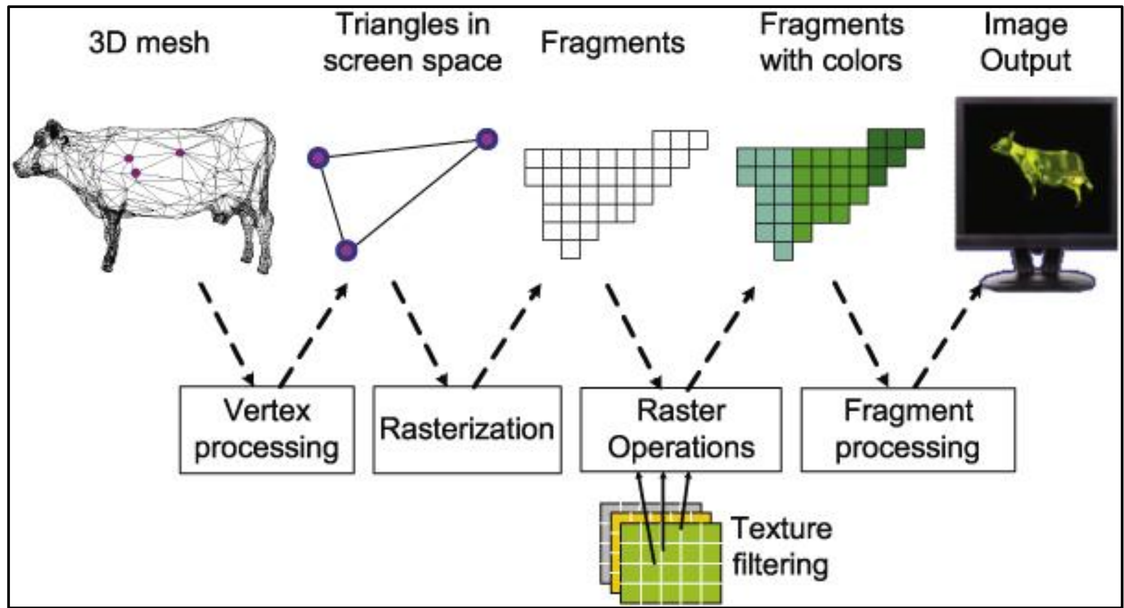
Şekil 4.3: GPU işlem hattı mimarisi



Kaynak: (Lighthouse3D, 2014)

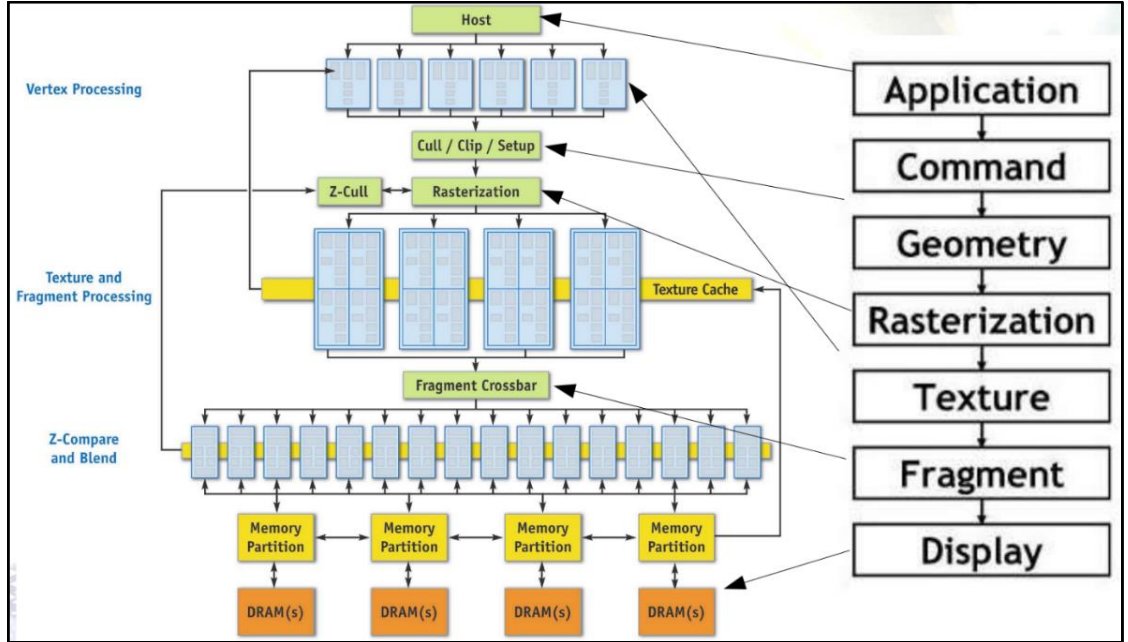
Şekil 4.3'de GPU işlem hattı mimarisi, Şekil 4.4'de işlem hattına giren veririn ne şekilde görüntüye dönüştürüldüğünü, Şekil 4.5'de ise NVIDIA GeForce 6800 ekran kartına ait blok diyagramı üzerinde işlem hattının gerçekleşmesi gösterilmiştir. Şekil 4.4 ve Şekil 4.5'de görüleceği üzere, grafik uygulaması tarafından grafik kartı API'si kullanılarak GPU'ya 3 boyutlu düzlemde düğüm koordinatları, renkleri ve düğümler arasındaki bağlılık ilişkileri iletir. İlk aşamada düğüm koordinatları 3 boyutlu düzlemden 2 boyutlu düzleme dönüştürülür. Dönüşüm işleminden sonra düğümler arası bağlılık bilgileri kullanılarak düğümlerden düzlemler veya çizgiler oluşturulur. Daha sonra oluşturulan bu nesneler görüntülenecek çözünürlüğe bağlı olarak görüntü matrisi üzerinde piksel parçalarına dönüştürülür. Piksel parçaları içerisinde kalan renksiz alanlar, düğümlerin renk bilgileri ve renk ağırlıkları kullanılarak renklendirilirler. Renklendirme işlemi sonrasında gürültü ve pürüz azaltma, ışıklandırma, gölgelendirme işlemleri de uygulanarak daha gerçekçi bir görüntü elde edilir.

Şekil 4.4: İşlem hattında veri dönüşümü



Kaynak: (Kaufman, A., Fan, Z., Petkov, K., 2009)

Şekil 4.5: NVIDIA GeForce 6800 blok diyagramı

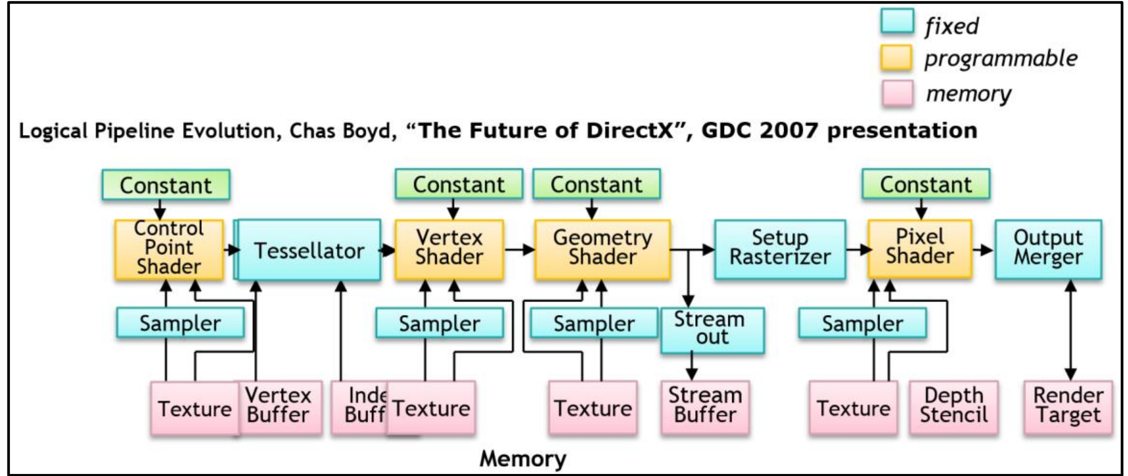


Kaynak: (Kilgariff, E., Fernando, R., 2005)

4.3 GPGPU KAVRAMI

GPU teknolojisindeki ilerlemelerle birlikte, günümüzde kullanılan modern GPU'lar programlanabilir arayüzler sunar hale gelmişlerdir. Bu programlanabilir arayüzler sayesinde GPU'nun işlem gücü ve paralel işleyebilme yeteneği sadece grafikle ilgili uygulamalarda değil aynı zamanda genel amaçlı hesaplamalar için de kullanılabilir hale gelmiştir. Bu durum ortaya grafik işlem birimi üzerinde genel amaçlı hesaplama (GPGPU – General Purpose programming on Graphic Processing Unit) kavramını çıkarmıştır.

Şekil 4.6: DirectX 10 programlanabilir işlem hattı mimarisi



Kaynak: (Tatarchuk, N., 2007)

GPU'lar yukarıdaki kısımlarda açıklanan işlem hattı mimarisi sayesinde, paralel olarak işlenebilecek nitelikte verinin yüksek performanslı bir şekilde paralel olarak işlenmesi konusunda çok elverişlidir. GPGPU uygulamaları GPU'ların grafik aygıtlarına özel olarak köşe kenar dönüşümü, dokulandırma, renklendirme, gölgelendirme vb. özelliklerinden ziyade SIMD şeklinde çalışan işlem hattı mimarisinden yararlanır. GPGPU uygulamaları genel olarak; işaret işleme, ses işleme, görüntü işleme, şifreleme, bioinformatik, yapay sinir ağları, paralelleştirilebilen bilimsel hesaplamalar, istatiksel hesaplamalar gibi yüklü miktarda verinin küçük parçaları üzerinde bağımsız ve paralel olarak işlem yapılmasına uygun olan uygulama alanlarında başarılıdırlar.

4.4 GPGPU PROGRAMLAMA MODELİ

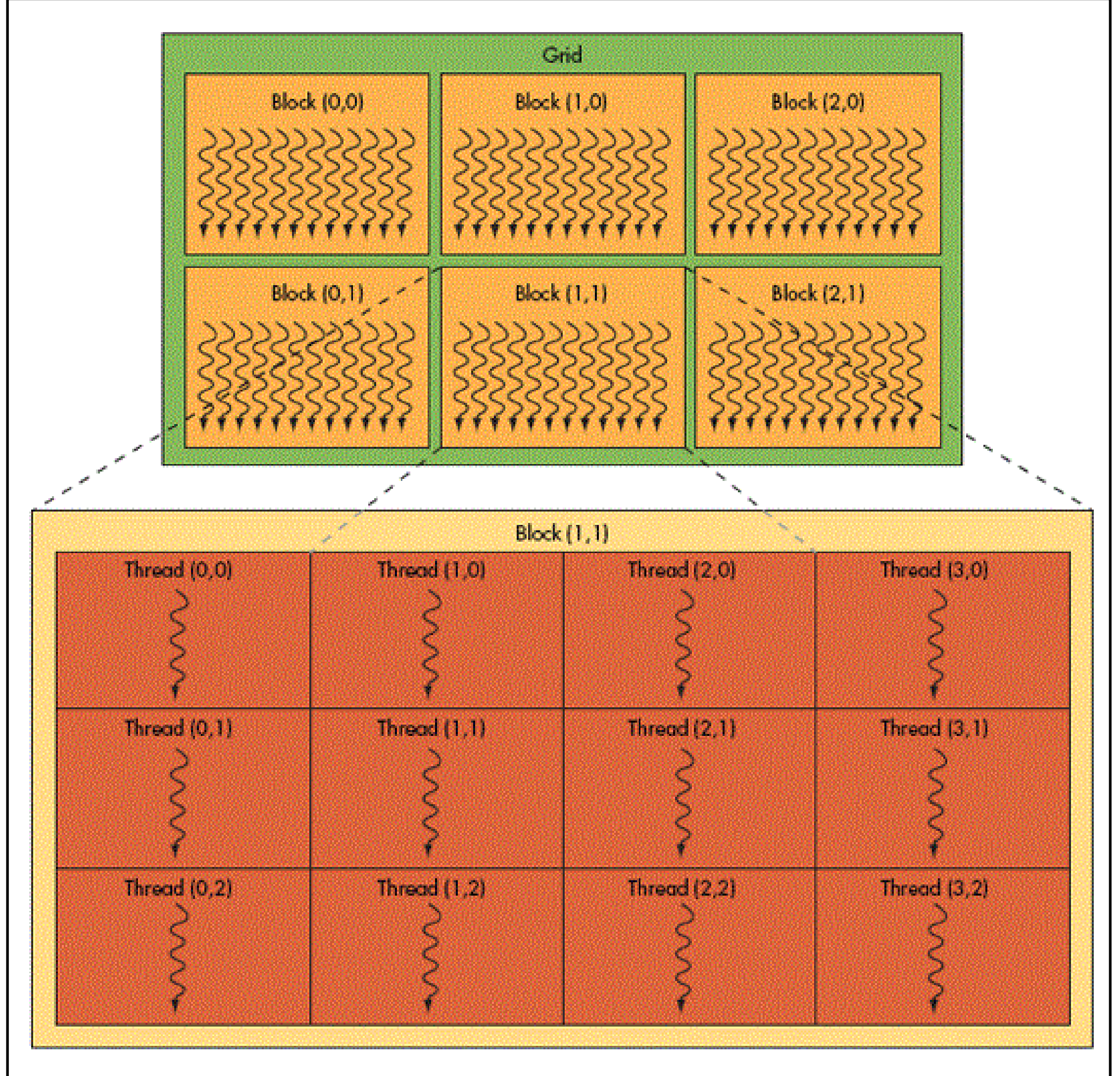
GPGPU uygulamaları genel olarak CPU üzerinde çalışan bir ev sahibi program (host program) ve GPU'daki çekirdekler üzerinde hesaplama yapacak olan çekirdek fonksiyonundan (kernel function) oluşur. Her çekirdekte çalışan çekirdek fonksiyonu, akış (stream) şekilde GPU'ya iletilen verinin kendine düşen daha düşük bir birimi üzerinde işlem yapar. Giriş verisinin GPU'ya iletilmesi, sonuç verisinin toplanması istenilen formata dönüştürülmesi gibi ardışık işlemleri CPU'da çalışan ev sahibi program yürütür.

Şekil 4.7'da modern GPGPU dillerinden CUDA programlama diline ait programlama modeli yapısı, Şekil 4.8'de ise C programlama diliyle yazılmış CPU üzerinde çalışan bir matris toplama fonksiyonunu GPU üzerinde gerçekleyen, CUDA programlama diliyle yazılmış GPU üzerinde çalışan bir kernel fonksiyonu ve C'de yazılmış bir ev sahibi program gösterilmiştir.

Şekil 4.7'da görüldüğü üzere, CUDA programlama modelinde GPU aygıtı grid olarak görülür ve çok sayıda bloktan oluşur. GPU'da bulunan çok sayıdaki çekirdekten her

biri aynı anda bir blok işleyebilir. Bir blok içerisinde paralel olarak çalıştırılabilen çok sayıda iş parçacığı (iplik – thread) bulunur. Bu iş parçacıklarından her biri kendisine düşen veri öbeği üzerinde tanımlanmış olan çekirdek fonksiyonu çalıştırır.

Şekil 4.7: CUDA dili programlama modeli yapısı



Kaynak: (NVIDIA, 2014)

Şekil 4.8: Matris toplama işleminin C'de ve CUDA'da gerçeklenmesi

CPU C program

```
void add_matrix_cpu
            (float *a, float *b, float *c, int N)
{
   int i, j, index;
   for (i=0;i<N;i++) {
    for (j=0;j<N;j++) {
      index =i+j*N;
      c[index]=a[index]+b[index];
      }
   }
}
void main()
{
  .....
     add_matrix(a,b,c,N);
}
```

CUDA C program

```
__global__ void add_matrix_gpu
            (float *a, float *b, float *c, int N)
{
   int i=blockIdx.x*blockDim.x+threadIdx.x;
   int j=blockIdx.y*blockDim.y+threadIdx.y;
   int index =i+j*N;
   if( i <N && j <N) c[index]=a[index]+b[index];
}


void main()
{
   dim3 dimBlock (blocksize,blocksize);
   dim3 dimGrid (N/dimBlock.x,N/dimBlock.y);
   add_matrix_gpu<<<dimGrid,dimBlock>>>(a,b,c,N);
}
```

Kaynak: (Ebersole, M., 2014)

Şekil 4.8'de görüldüğü gibi C programlama dilinde yazılmış olan matris toplama fonksiyonu matris elemanları üzerinde ardışıl döngüler şeklinde işlem yaparak matris toplama işlemini gerçekleştirir. CUDA ile yazılmış matris toplama programına bakıldığında, program CPU üzerinde çalışan ev sahibi programdaki main fonksiyonundan ve GPU üzerinde çalışan add_matrix_gpu çekirdek fonksiyonundan oluşur. Ev sahibi program bir bloğun boyutunu ve grid içerisindeki blok sayısını belirler. Daha sonra bloklar üzerinde paralel olarak çalışacak olan add_matrix_gpu fonksiyonunu çağırır. Çağrı sonucu add_matrix_gpu çekirdek fonksiyonu bloklar ve bloklardaki iş parçacıkları üzerinde paralel olarak yürütülür. Her bir iş parçacığı kendi iş parçacığı numarası (thread ID), blok numarası (blok ID) ve blok boyutunu (blockDim) kullanarak kendine düşen veri parçası için matris toplama işlemini gerçekleştirir. İş parçacığı numarası, blok numarası, blok boyutu gibi veriler bağlam (context) içerisinde her bir çekirdek fonksiyonuna geçer.

4.5 GPGPU PROGRAMLAMA DİLLERİ

GPGPU kavramının yaygınlaşmasıyla birlikte daha önceden assembly programlama dilleriyle programlanan GPU'lar için yüksek seviyeli GPGPU programlama dilleri geliştirilmiştir. Yüksek seviyeli programlama dillerinin geliştiriciler tarafından anlaşılması ve yazılması assembly dillerine göre daha kolaydır. Yüksek seviyeli programlama dillerinde geliştirilen kodlar aygıtın teknik detaylarını geliştiricilerden soyutlayabilme ve aygıtlar arası taşınabilme açısından da daha avantajlıdır. Zaman içerisinde geliştirilmiş olan programlama dillerinin başlıcaları şunlardır: C for Graphics, Close to Metal, BrookGPU, CUDA, DirectCompute, OpenCL.

C for Graphics

C for Graphics (Cg) (NVIDIA Developer, 2014), NVIDIA ve Microsoft tarafından geliştirilen, köşe ve piksel parçacık işlemcilerini (vertext and pixel shaders) programlamaya yarayan C tabanlı yüksek seviyeli bir GPGPU dilidir. Cg dili ile yazılmış çekirdek fonksiyonları OpenGL ve DirectX API'leri ile çağrılabilmektedir. Cg programlama dili ile yazılmış çekirdek fonksiyon kaynak kodları çalışma zamanı esnasında derlenebilir.

Close to Metal

Close to Metal (AMD's Close-to-the-Metal, 2014), ATI tarafından AMD GPU'larda kullanılmak üzere geliştirilmiş bir düşük seviyeli GPGPU dilidir. Programcılara GPU aygıtının komut setine ve belleğine doğrudan erişim sağlamaktadır. AMD kartlarda daha sonradan daha yüksek seviyeli olan Stream SDK teknolojisine geçilmiştir.

BrookGPU

Stanford Üniversitesinde geliştirilmiş olan BrookGPU (BrookGPU, 2014), Brook akış programlama (Brook stream programming) dilinin AMD ve NVIDIA GPU aygıtları üzerinde çalışmak üzere uyarlanmış halidir. BrookGPU dili ev sahibi program tarafından API olarak OpenGL, DirectX veya Close to Metal API'leri ile Windows ve Linux platformlarında çalışabilmektedir.

CUDA

CUDA (Compute Unified Device Architecture) (CUDA (Compute Unified Device Architecture, 2014), NVIDIA tarafından GPU'larda kullanılmak üzere geliştirilmiş olan bir paralel hesaplama mimarisidir. GPU etkileşimi için hem alçak seviyeli hem de yüksek seviyeli API sunmaktadır. CUDA bir GPGPU dili olarak; ardışıl bellek erişimi, paylaşımlı bellek, GPU'dan daha hzılı veri okuma, tam sayı ve bit bazında (bitwise) işlemler için destek sağladığından dolayı avantajlıdır.

Direct Compute

Direct Compute (Compute Shader Overview, 2014), Microsoft tarafında Windows işletim sistemi üzerinde GPU programlamada kullanılmak üzere geliştirilmiş olan bir API'dir. DirectX 10 ve DirectX 11 destekleyen GPU'lar üzerinde çalışabilmektedir.

OpenCL

OpenCL (Open Computing Language) (OpenCL™ (Open Computing Language) Zone, 2014), CPU, GPU ve diğer işlemcilerden oluşabilen heterojen ortamlarda taşınabilir programlar yazılabilmesi amacıyla geliştirilmiş bir çatıdır. Apple, IBM, Intel, AMD ve NVIDIA gibi büyük üreticilerin katkısı ve işbirliği ile kar amacı

gütmeyen bir proje olarak Khoronos Group tarafından geliştirilmiştir. OpenCL çatısı, GPU üzerinde çalışan çekirdek fonksiyonların yazımı için C tabanlı programlama dili ve ev sahibi program tarafında çalışan genel geçer bir uygulama programlama arayüzü (API – Application Programming Interface) barındırmaktadır. OpenCL çatısı ile programcıların çeşitli üreticilere ait, çeşitli modellerdeki CPU ve GPU aygıtlarının bulunduğu heterojen ortamlarda çalışabilen ve taşınabilir programlar yazmaları mümkün kılınmıştır. Aygıt üreticileri OpenCL standartlarına uygun OpenCL gerçeklemelerini kapalı kaynaklı olarak kendilerine özel geliştirmektedirler. Fakat standartlara uyularak ortak OpenCL veri tipleri ve API fonksiyonları imzaları (method signatures) kullanıldığı için OpenCL çatısı ile gerçeklenen programlar üretici ve aygıt bağımsız çalışabilmektedir. Bu ortak standartlar sayesinde OpenCL, geliştiricileri üreticiye özel veya aygıta özel teknik detaylardan da soyutlamaktadır. NVIDIA ve AMD'ye ait modern GPU'ların çoğu OpenCL standartlarına uygun üretilmiştir. Bu nedenlerden dolayı OpenCL birçok üretici tarafından desteklenen ve giderek daha yaygın olarak kullanılan bir GPGPU çatısı haline gelmektedir.

5. OpenCL

OpenCL (Open Computing Language), CPU, GPU ve diğer işlemcilerden oluşabilen heterojen sistemlerde genel amaçlı bilgisayar kullanımı için tasarlanmış ilk tam anlamıyla açık ve lisans ücreti gerektirmeyen programlama standardıdır. OpenCL programcılara pahalı kaynak kod yatırımlarını ellerinde tutma ve çok çekirdekli CPU'ları ve en son GPU'ları kolaylıkla hedefleyebilme fırsatı sunmaktadır. Büyük endüstri üreticilerinin temsilcilerinden oluşan bir açık standartlar komitesinde geliştirilen OpenCL, CPU ve GPU çekirdeklerindeki uygulamaları hızlandırmak için her üreticiye uygun, özel olmayan bir çözüm sunar. OpenCL standartları programcıların aygıtlar arasında taşınabilir, üretici ve aygıt bağımsız programlar yazabilmelerini sağlamak amacıyla geliştirilmiştir.

5.1 OPENCL ANATOMİSİ

OpenCL sertifikasyonu üç ana bölümden oluşur: dil spesifikasyonu, platform katmanı API ve çalışma zamanı katmanı API (ATI, 2010).

Dil spesifikasyonu, GPU'lar ve çok çekirdekli CPU'lar gibi desteklenen hızlandırıcılarda çalışan veri işleme çekirdekleri yazmak için sözdizimi ve programlama arayüzünü açıklar. Kullanılan dil ISO C99'un bir altkümesine dayanmaktadır. Geliştirici topluluğundaki yaygınlığı ve bilinirliği nedeniyle ilk OpenCL veri işleme çekirdek dilinin temeli olarak C dili seçilmiştir. Farklı platformlarda tutarlı sonuçlar sağlamak için, zengin içerikli bir yerleşik işlevler kümesiyle birlikte bütün kayan noktalı işlemler için iyi tanımlanmış bir IEEE 754 sayısal doğruluk tanımlanmıştır. Geliştirici, OpenCL veri işleme çekirdeklerini önceden derleme veya OpenCL çalışma zamanının talep üzerine çekirdeklerini derlettirme seçeneklerine sahiptir.

API platform katmanı geliştiricinin, sistemdeki cihaz sayısı ve türlerini sorgulayan rutinlere erişmesini sağlar. Bunun üzerine geliştirici, iş yüklerini doğru biçimde çalıştırmak için gerekli olan veri işleme cihazlarını seçip başlatabilir. Bu katmanda iş iletme ve veri aktarımı talepleri için veri işleme bağlamları ve iş sıraları oluşturulur.

Son olarak çalışma zamanı katmanı API, geliştiricinin uygulama için veri işleme çekirdeklerini sıraya koymasını sağlarken OpenCL sistemindeki veri işleme ve bellek kaynaklarını yönetmekten sorumludur.

5.1.1 Dil Spesifikasyonu

a. C – bazlı çapraz platform programlama arayüzü
b. Dil uzantılarıyla ISO C99 altkümesi
c. İyi tanımlanmış sayısal doğruluk – tanımlanmış maksimum hatalı IEEE 754 tamamlama davranışı

d. Veri işleme çekirdek yürütücülerinin çevrimiçi veya çevrimdışı derlenmesi ve oluşturulması
e. Zengin içerikli bir yerleşik işlevler kümesi içerir

5.1.2 Platform Katmanı API

a. Çeşitli veri işleme kaynakları boyunca bir donanım soyutlama katmanıdır
b. Veri işleme cihazlarını sorgulama, seçme ve başlatma
c. Veri işleme bağlamları ve iş sıraları oluşturma

5.1.3 Çalışma Zamanı API

a. Veri işleme çekirdekleri yürütme
b. Programlama, veri işleme ve bellek kaynaklarını yönetme

5.2 OPENCL TERİMLERİ

OpenCL standartlarında üreticiler ve aygıtlar arasında ortak standartları yakalamak amacıyla bazı terimler tanımlanmıştır. OpenCL uygulamaları; ev sahibi program (host), aygıt (device), program nesneleri (program objects), çekirdek fonksiyonları (kernel function), çekirdek nesneleri (kernel objects) ve bellek nesneleri (memory objects) gibi terimler üzerine gerçeklenir.

5.2.1 Aygıtlar

Bir bilgisayar sistemindeki hesaplama aygıtları, “aygıt” (device) olarak tanımlanır. Bir aygıt içerisinde bir veya birden fazla hesaplama birimi (compute unit) bulunabilir. Günümüzde 2 ile 8 çekirdeğe sahip olan CPU’larda veya 80 ile 100 çekirdeğe sahip olabilen GPU’larda her bir çekirdek bir hesaplama birimine karşılık düşer.

5.2.2 Çekirdek Fonksiyonları

C tabanlı OpenCL dili ile OpenCL destekleyen aygıtlar üzerindeki hesaplama birimleriyle çalıştırılmak üzere yazılan fonksiyonlar çekirdek fonksiyonlardır. Çekirdek fonksiyonları C, C++, Objective C gibi dillerde yazılabilen ev sahibi programlardan OpenCL API çağrıları aracılığı ile tetiklenir ve hesaplama birimleri üzerinde çalışır, sonuçları yine ev sahibi programa döndürür. Çekirdek fonksiyonlar, çalışma zamanı (runtime) esnasında, hesaplama birimi üzerinde çalışacak şekilde kaynak kodlarından derlenir. Çekirdek fonksiyonu derlendiğinde bir çekirdek oluşur.

5.2.3 Çekirdek Nesneleri

Çekirdek nesneleri program içerisinde tanımlanmış belli bir çekirdeği ve o çekirdek ile çalıştırılan argüman değerlerini tutar.

5.2.4 Programlar

Bir OpenCL programı, OpenCL çekirdeklerini, bu çekirdekler tarafından çağırılan dış fonksiyonları ve sabitleri barındırır.

5.2.5 Bağlamlar

Bağlam (context) OpenCL çekirdeklerinin yürütüldüğü ortamı temsil eder. Bağlam, aygıt kümesini, bu aygıtların erişebildiği bellek bilgisini ve çekirdekler üzerinde yürütülmesi için zamanlanmış komut kuyruklarının (command queue) bilgisini tutar. Bağlam, bellek nesnelerinin aygıtlar arasında paylaşılmasını sağlar.

5.2.6 Program Nesneleri

Bir program nesnesi, bir OpenCL programını temsil eden veri tipidir. Program bağlamı referansı, program kaynak kodu, programın derlenmiş ve yürütülebilir hali, programın hangi aygıtlar için derlendiği, derlenme seçenekleri ve derlenme kaydı (build log) bilgileri program nesnesinde tutulur.

5.2.7 Komut Kuyrukları

Komut kuyrukları hesaplama aygıtlarına iş atamak için kullanılır. Aygıtlardaki çekirdeklerin yürütülmesini ve bellek nesnelerini düzenlerler. OpenCL komutları komut kuyruğundaki sıraya göre yürütür.

5.2.8 Ev Sahibi Programlar

Ev sahibi programlar, çekirdek fonksiyonların hesaplama aygıtı üzerinde yürütülmesi için gerekli bağlamı hazırlayan ve yürütme işlemini düzenleyen programdır. Ev sahibi program CPU üzerinde çalışır fakat CPU aynı zamanda bir hesaplama aygıtı olarak da kullanılabilir. Çekirdek fonksiyonlar aygıtlar üzerinde çalışması için ev sahibi program hangi hesaplama aygıtlarının kullanılabileceğini bulur, uygulama için uygun olan hesaplama aygıtını seçer, seçilen aygıtlar için komut kuyruklarını oluşturur ve çekirdek fonksiyonlarda kullanılacak bellek nesnelerini yaratır.

5.2.9 Bellek Nesneleri

Bellek nesneleri aygıtın genel belleğinin ilgili bölgelerine referanstır. Ev sahibi program bellek nesnelerini kullanarak aygıt belleğine yazma ve aygıt belleğinden okuma işlemlerini gerçekleştirebilir.

5.3 OPENCL ÇALIŞMA MODELİ

OpenCL çalışma modeli aşağıdaki modellerin birleşimi şeklinde tanımlanır:

a. OpenCL Platform Modeli
b. OpenCL Yürütme Modeli
c. OpenCL Bellek Modeli
d. OpenCL Programlama Modeli

5.3.1 OpenCL Platform Modeli

OpenCL platform modeli bir ev sahibi program ve onun bağlandığı bir veya daha fazla OpenCL aygıtından oluşur. Bir aygıt bir veya birden fazla hesaplama birimi, her bir hesaplama birimi de bir veya birden fazla işleme elemanı (processing element) barındırır. Aygıt üzerindeki hesaplamalar, işleme elemanlarında gerçekleştirilir.

Ev sahibi program komut kuyruklarını kullanarak işleme birimlerine çekirdek fonksiyonların yürütülmesi için komutlar gönderir. Hesaplamalar, ev sahibi programın yarattığı bellek nesneleri üzerinde, işleme elemanlarından SIMD yapısında gerçekleştirilir. Ev sahibi program yine komut kuyruğu aracılıyla hesaplama sonuçlarının bulunduğu aygıt belleğini okuyabilir. Şekil 5.1'de OpenCL Platform Modeli gösterilmektedir.

5.3.2 OpenCL Yürütme Modeli

OpenCL, sadece GPU'ları değil aynı zamanda çok çekirdekli CPU'lar gibi diğer hızlandırıcıları da hedeflediğinden, belirlenen veri işleme çekirdeği türünde esneklik verilir. Veri işleme çekirdekleri, GPU'ların mimarisine iyi uyan veri – koşutlu veya CPU'ların mimarisine iyi uyan görev – koşutlu olarak düşünülebilir.

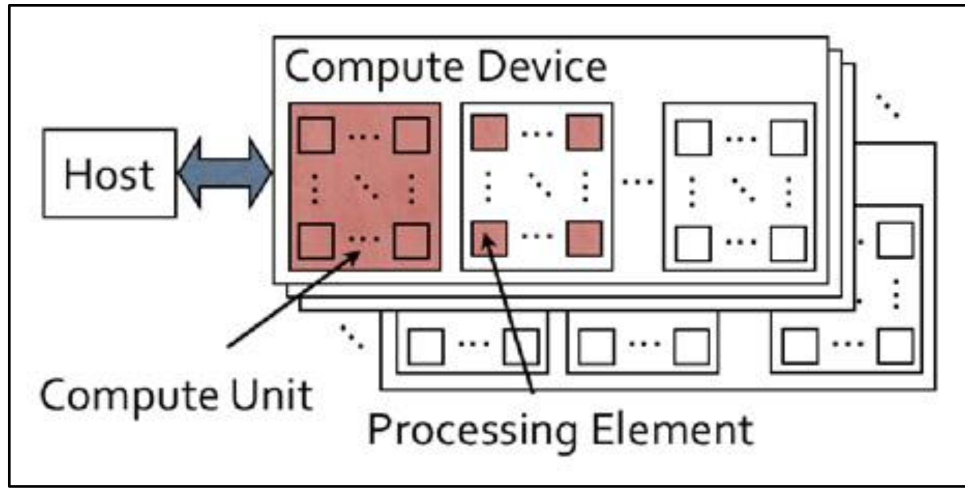
Bir veri işleme çekirdeği, makine kodunun temel birimidir ve bir C işlevine benzer olarak düşünülebilir. Bu tür çekirdeklerin yürütülmesi, yürütme için çekirdek sıraya sokulduğunda sisteme iletilen parametrelere bağlı olarak sıralı veya sıra – dışı olarak ilerleyebilir. Olaylar sağlanır, böylece geliştirici bekleyen çekirdek yürütme talepleri ve diğer çalışma zamanı taleplerinin durumunu kontrol edebilir.

Düzenleme açısından bir çekirdeğin yürütme alanı bir N – boyutlu veri işleme alanınca tanımlanır. Böylece sisteme, kullanıcının bir çekirdeğinin ne kadar büyük bir soruna uygulanmasını istediği bildirilir. Yürütme alanındaki her unsur bir iş kalemidir ve OpenCL, iş kalemlerini senkronizasyon ve iletişim amaçları için çalışma gruplarında gruplama özelliğini sağlar.

OpenCL yürütme modeli bir veya birden fazla hesaplama aygıtı üzerindeki çekirdek örneklerinin (kernel instances) ev sahibi program tarafından eş zamanlı olarak işletilmesini kapsar. Çekirdeğin her bir örneği bir iş öğesi (work – item) olarak

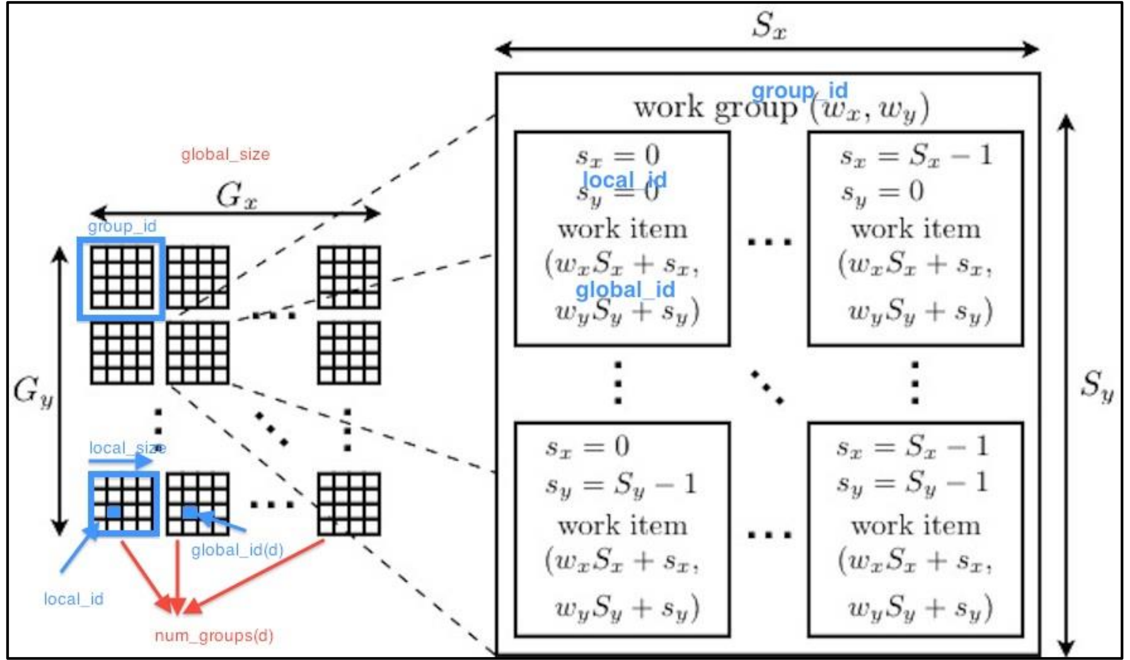
tanımlanmaktadır. İş öğeleri GPU çekirdekleri tarafından eş zamanlı olarak, her bir çekirdek bir iş öğesini çalıştıracak şekilde yürütülür. Her bir iş öğesi aynı çekirdek fonksiyonunu kendisine düşen veri parçacığı üzerinde yürütür. İş öğeleri bir araya gelerek iş gruplarını (work – group) oluşturur. Aynı zamanda, tüm veriyi kapsayan iş öğeleri bir araya gelerek bir indeks alanı (index space) tanımlar. OpenCL 1, 2 ve 3 boyutlu indeks alanlarını destekler. Bu indeks alanı OpenCL standartlarında NDRange indeks alanı olarak adlandırılır. İndeks alanı içerisinde her iş öğesi, genel ID'sini (global ID) veya yerel ID (local ID) ve iş grubu ID (work – group ID) kullanarak verinin hangi kısmını işleyeceğini belirler. Şekil 5.2 NDRange indeks alanını, iş gruplarını ve iş öğelerini göstermektedir.

Şekil 5.1: OpenCL Platform Modeli



Kaynak: (AMD Staff, 2011)

Şekil 5.2: NDRange indeks alanı, iş grupları ve iş öğeleri



Kaynak: (Rudolph, J., 2012)

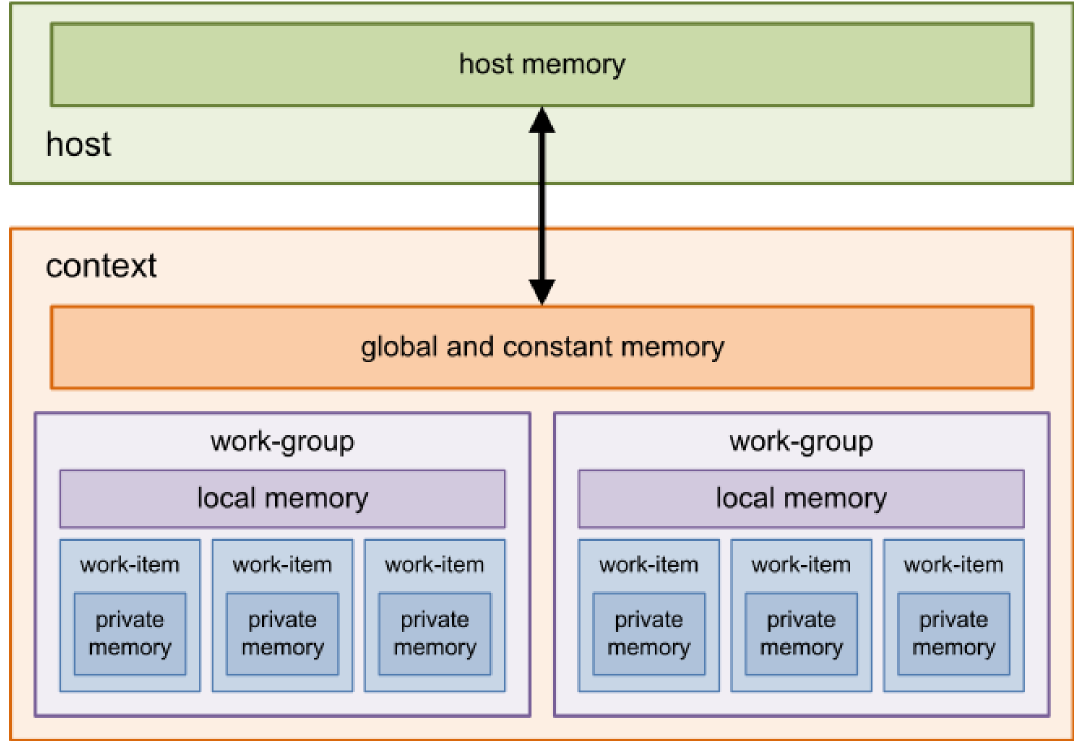
5.3.3 OpenCL Bellek Modeli

Ev sahibi program bellek nesnelerini OpenCL API çağrılarıyla genel bellek (global memory) üzerinde yaratır. Ev sahibi programın ve hesaplama aygıtlarının bellekleri genellikle birbirinden bağımsız çalışır. İki bellek arası etkileşim gerektiğinden bellek blokları iki bellek arasında transfer edilen veya ev sahibi uygulama aygıt belleğine erişim için haritalama (mapping / unmapping) yöntemini kullanır. OpenCL bellek modelinde, iş öğelerinin erişebileceği dört çekirdek bellek alanı vardır:

a. **Genel Bellek (Global Memory):** Tüm iş öğelerinin okuma ve yazma için erişimine açık olan bellek bölgesidir.
b. **Sabit Bellek (Constant Memory):** Genel belleğin, çekirdek fonksiyonlarının yürütülmesi esnasında sabit kalan alanıdır. Ev sahibi programın, çekirdek fonksiyonunun yürütülmesinden önce bu bölgeye yazdığı veri tüm iş öğeleri tarafından okunabilir.
c. **Yerel Bellek (Local Memory):** Bir iş grubu içerisinde paylaşılan, tüm iş öğelerinin okuma ve yazma iznine sahip oldukları bellek bölgesidir.
d. **Özel Bellek (Private Memory):** Bir iş öğesinin kendine özel bellek bölgesidir. Diğer iş öğeleri bu kısma erişemezler.

Şekil 5.3’de OpenCL bellek modeline göre hesaplama aygıtı içerisindeki bellek bölgesi çeşitleri gösterilmiştir.

Şekil 5.3: OpenCL Bellek Modeli



Kaynak: (AMD Staff, 2011)

5.3.4 OpenCL Programlama Modeli

OpenCL, veri paralel (data parallel) ve görev paralel (task parallel) programlama modellerini destekler. Veri paralel programlama modelinde aynı çekirdek fonksiyonu verinin küçük parçaları üzerinde paralel olarak yürütülür. Her veri kümesi 1, 2 veya 3 boyutlu uzayda belirli noktalar kümesine karşılık düşer. Görev paralel programlama modelinde ise farklı çekirdek fonksiyonları yaratılır ve bu fonksiyonlar farklı verilerle aynı anda paralel çalışan iş parçacıkları şeklinde yürütülür.

5.4 ÖRNEK – VEKTÖR EKLEME ÇEKİRDEĞİ

Aşağıda OpenCL'de yazılan basit bir vektör ekleme çekirdeği bulunmaktadır (Ahmed, M., F., 2010). Çekirdeğin, ikisi giriş (a ve b) ve biri de tekli çıkış (c) olmak üzere üç bellek nesnesi belirlendiğini görebilirsiniz. Bunlar küresel bellek alanı içinde bulunan veri dizileridir. Bu örnekte, bu çekirdeği yürüten veri işlem birimi, eşsiz iş kalemi kimliğini alır ve bunu a ve b'den uygun değeri okuyup toplamı c'de saklayarak vektör eklemedeki rolünü tamamlamak için kullanır.

```
__kernel void vec_add (__global const float *a,
                        __global const float *b,
                        __global float *c)
{
        int gid = get_global_id(0);
        c[gid] = a[gid] + b[gid];
}
```

Bu örnekte çevrim içi derleme kullanılacağı için yukarıdaki kod program_source adındaki bir karakter dizisinde kaydedilecektir.

Veri işleme çekirdeği kodunu tamamlamak için ana bilgisayar işlemcisinde aşağıdaki işlemleri çalıştırmak için kullanılan koddur:

a. Bir OpenCL bağlamı açmak,
b. Üzerinde yürütmek için cihazları alıp seçmek,
c. Yürütme ve bellek taleplerini kabul etmek için bir komut sırası oluşturmak,
d. Veri işleme çekirdeği için girişleri ve çıkışları tutacak OpenCL bellek nesneleri tahsis etmek,
e. Veri işleme çekirdek kodunu çevrim içi derlemek ve oluşturmak,
f. İfadeleri ve yürütme alanına hazırlamak,
g. Veri işleme çekirdek yürütmesini başlatmak,
h. Sonuçları derlemek.

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
                                     NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
device = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICE, cb, devices, NULL);

//create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                                sizeof(cl_float)*n, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                                sizeof(cl_float)*n, srcB, NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                                sizeof(cl_float)*n, NULL, NULL);

// create the program
program = clCreateProgramWithSource(context, 1, &program_source, NULL, NULL);

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *)&memobjs[0], sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1], sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2], sizeof(cl_mem));

// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, global_work_size,
                                NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(context, memobjs[2], CL_TRUE, 0,
                                n*sizeof(cl_float), dst, 0, NULL, NULL);
```

5.5 OPENCL’E GİRİŞ

CPU ve GPU’da işletilecek kodlu ilk uygulama yazma ve çalıştırma

OpenCL, yüksek – performanslı hesaplama alanında birçok fayda sağlamaktadır. En önemli özelliklerinden biri taşınabilirliktir. Çekirdek(Kernel) olarak isimlendirilen, OpenCL – kod rutinleri, Intel, AMD, Nvidia ve IBM gibi popüler imalatçıların GPU ve CPU’larında işletilebilir.

OpenCL çekirdekleri sadece farklı aygıt tiplerinde değil, tek bir uygulamada bir zamanda çoklu aygıtlara gönderilebilir. Örneğin, eğer bilgisayarımız, AMD fusion işlemci ve AMD grafik kartına sahipse iki aygıt arasında çekirdeklerin işletilmesini senkronize edebilir ve aralarında veriyi paylaştırabilirsiniz. Hatta OpenCL, OpenGL veya Direct3D işlemlerini hızlandırmak için kullanılabilir.

Bu avantajlarına rağmen, OpenCL'in önemli bir sakıncası: öğrenmesinin kolay olmamasıdır. OpenCL, MPI veya PVM veya benzeri mimarilerden türetilmemiştir. NVIDIA'nın CUDA'sına benzer, ama OpenCL'in veri yapıları ve fonksiyonları farklıdır. En basit anlatımlarla bile yenilerin anlaması zordur.

Bu çalışmanın amacı, basit bir şekilde OpenCL'nin arkasındaki konsepti açıklamak ve bu konseptin kodlamada nasıl uygulandığını göstermektir. Host uygulamalarının nasıl çalıştığı açıklanacak ve daha sonra bir aygıtta çekirdeklerin nasıl işletildiği gösterilecektir. 64 float değeri toplayan çekirdekli bir örnek uygulama geliştirilecektir.

5.5.1 Host Uygulamasını Geliştirmek

Bir OpenCL projesi geliştirmedeki ilk adım, host (ev sahibi) uygulamasını kodlamaktır. Bu, kullanıcının bilgisayarında (host) çalışır ve bağlı aygıtlara gönderir. Host uygulaması, C veya C++'de kodlanabilir ve her host uygulaması beş veri yapısına gereksinim duyar:

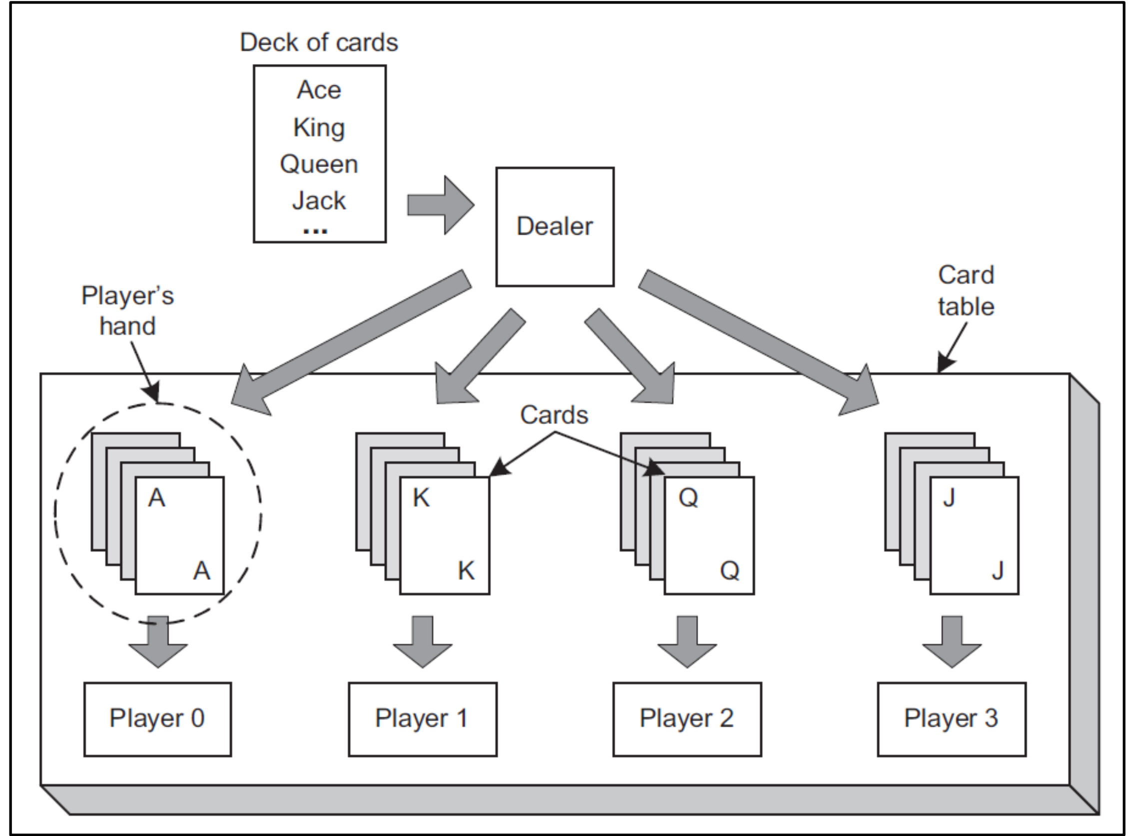
a. cl_device_id
b. cl_kernel
c. cl_program
d. cl_command_queue
e. cl_context

OpenCL'in yapısının daha kolay anlaşılabilmesi için benzetmeler kullanılarak devam edilecektir. OpenCL host uygulaması, bir kart oyunu gibidir.

5.5.1.1 Poker kart oyunu

Kart oyununda, bir dağıtıcı, bir veya daha çok oyuncuyla bir masada oturur ve bir desteden kartları dağıtır. Her oyuncu, bir el olarak kartları alır ve sonra en iyi oyun için analiz yapar. Oyuncular, birbirleri ile etkileşemez veya diğer oyuncuların kartını göremez, ama ekstra kartlar için dağıtıcıya istek yapar veya ortaya konan paralarda değişiklik yapabilir. Dağıtıcı, bu istekleri yönetir ve oyun bittiği andan itibaren kontrolü alır. Şekil 5.4'te bu olay resmedilmektedir.

Şekil 5.4: Oyun masası



Kaynak: (Scarpino, M., 2011)

Dağıtıcı ve oyuncular ek olarak, Şekil 5.4 ile aynı şekilde oyun masası vardır. Masada oturanlar ele katılmak zorunda değiller, ama sadece masada oturanlar oyuna katılabilir.

5.5.1.2 Beş veri yapısı

Benzetmede, kart dağıtıcısı host'u temsil eder. Benzetmenin diğer aktörleri, bir host uygulamasında yaratılması ve ayarlanması gereken beş OpenCL veri yapısına karşılık gelir (Scarpino, M., 2011).

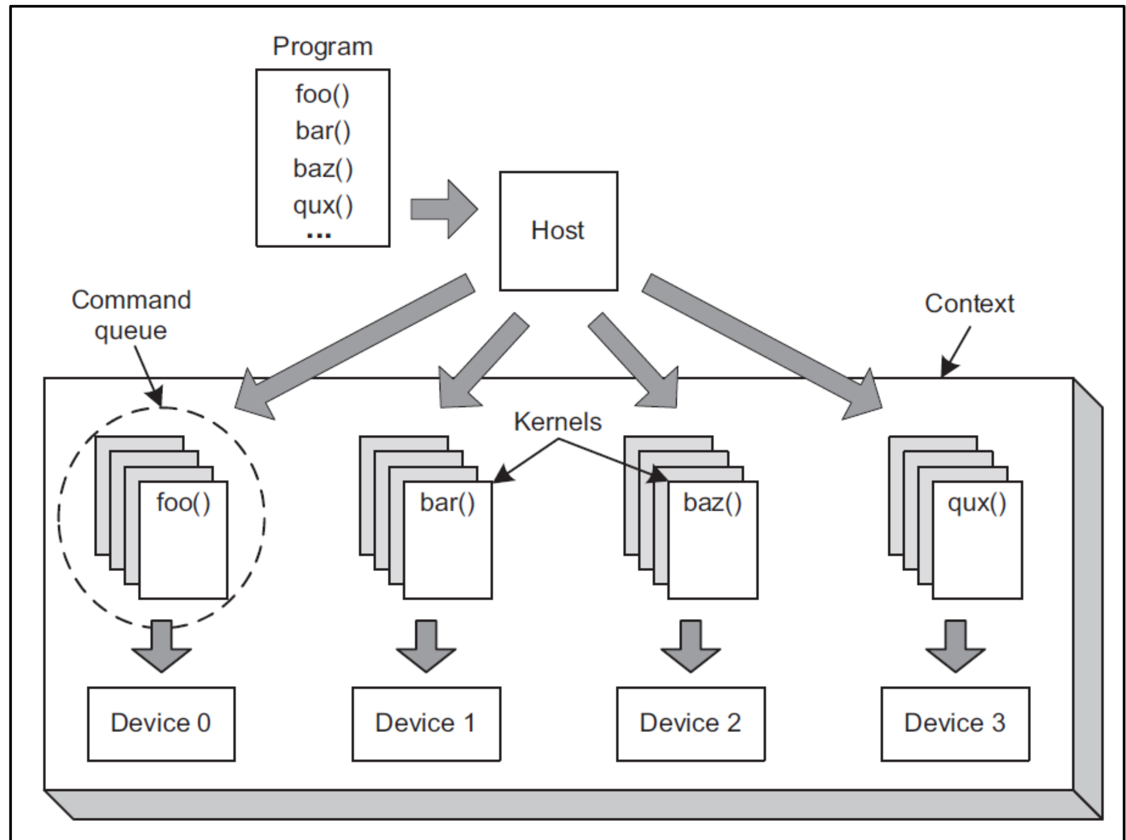
a. **Aygıt (device):** OpenCL aygıtları, oyunculara karşılık gelir. Bir oyuncunun dağıtıcıdan kartları alması gibi, bir aygıt hosttan çekirdekleri alır. Kodlamada, aygıt, cl_device_id ile temsil edilir.
b. **Kernel:** :OpenCL kernelleri, kartlara karşılık gelir. Dağıtıcının oyunculara kartları dağıttığı gibi hemen hemen aynı şekilde, bir host uygulaması aygıtlara çekirdekleri dağıtır. Kodlamada, bir kernel, cl_kernel ile temsil edilir.
c. **Program:** Bir OpenCL programı, bir kart destesi gibidir. Bir dağıtıcının bir desteden kartları seçtiği gibi aynı şekilde, host bir programdan çekirdekleri seçer. Kodlamada, bir program, cl_program ile temsil edilir.

d. **Komut Kuyruğu (Command Queue):** Bir OpenCL komut kuyruğu, bir oyuncunun eli gibidir. Her oyuncu, bir el olarak kartlar alır ve her aygıt bir komut kuyruğu boyunca çekirdekleri alır. Kodlamada, bir komut kuyruğu, cl_command_queue ile temsil edilir.

e. **Context:** OpenCL contextleri, masaya karşılık gelir. Bir masanın, kartları dağıtmak için oynadığı rol gibi, bir OpenCL contexti, aygıtlara, çekirdeklerin alınması ve veri transferi için benzer imkanı verir. Kodlamada, bir context, cl_context ile temsil edilir.

Bu benzetmeye açıklık getirmek için, Şekil 5.5'te bir host uygulamasında bu beşli veri yapısının birbiriyle nasıl çalıştığı gösterilmektedir. Görüleceği üzere, bir program çoklu fonksiyonlar içerir ve her çekirdek programdan alınan bir fonksiyon içerir.

Şekil 5.5: Host uygulamasının çalışma şekli



Kaynak: (Scarpino, M., 2011)

5.5.1.3 Benzetmedeki noksanlar

Benzetme kusurlara sahiptir. Altı önemli kusur aşağıda verilmiştir (Scarpino, M., 2011):

a. Platformlardan bahsedilmemektedir. Bir platform, OpenCL’in imalatçı uyarlamasını tanımlayan bir veri yapısıdır. Platformlar, aygıtlara erişimi mümkün kılar. Örneğin, Nvidia platformu ile Nvidia aygıtına erişebilirsiniz.
b. Bir kart dağıtıcısı masada oturacak oyuncuları seçemez. Bununla birlikte, bir OpenCL host, bir context’de oturacak aygıtları seçer.
c. Bir kart dağıtıcısı çoklu oyunculara aynı kartı dağıtamaz, ama bir OpenCL host, komut kuyrukları boyunca çoklu aygıtlara aynı çekirdekleri gönderebilir.
d. Benzetmede, aygıtların çekirdekleri nasıl işleteceğinden söz edilemez. Birçok OpenCL aygıtı, çoklu işleme öğeleri içerip, her öğe, girdi verisinin bir alt kümesini işleme tabi tutabilir. Host, çekirdeği işlemek için üretilen çalışma öğelerinin sayısını tanımlar.
e. Bir kart oyununda, dağıtıcı oyunculara kartları dağıtır ve her oyuncu elindeki kartları düzenler. OpenCL’de, host, her aygıt için bir komut kuyruğu yaratır ve komutları kuyruklar. Bir komutun yazımı, bir çekirdeğin işletilmesini aygıta söyler.
f. Bir kart oyununda, dağıtıcı, değişmez zaman paylaşımında kartları verir. OpenCL, host uygulamalarının aygıtlara çekirdekleri nasıl dağıtacağı hakkında hiçbir kısıt koymaz.

Bu noktada, bir host uygulamasının işinin büyük bir parçasının, çekirdekleri yaratmak ve GPU, CPU veya melez işlemciler gibi OpenCL destekli aygıtlara onları konuşlandırmayı içerir. Şimdi bu çekirdeklerin aygıtlarda nasıl işletildiğini göreceğiz.

5.5.2 OpenCL Çekirdekleri

OpenCL’in büyük avantajlarından biri, çekirdeklerin, GPU’lar gibi yüksek performanslı hesaplama aygıtlarında işletilebilmesidir. Kodlamada bu paralel işleme avantajından faydalanmak için, bir OpenCL geliştiricisi, iki noktayı açıkça anlamaya ihtiyaç duyar (Scarpino, M., 2011).

a. **OpenCL işletim modeli:** Çekirdekler, bir veya daha çok çalışma öğesi ile işletilir. Çalışma öğeleri, çalışma grupları içinde toplanıktır ve her çalışma grubu hesaplama ünitesinde işletilir.
b. **OpenCL bellek modeli:** Çekirdek verisi, dört adres boşluğundan birine yerleştirilmelidir. Bunlar; global bellek, sabit bellek, yerel bellek veya özel bellektir. Verinin yeri, işletimin çabukluğunu belirler.

Bir çekirdeğin işletimi bir öğrencinin okulda geçirdiği bir güne benzetilebilir. Bununla ilgili bir örnek aşağıda verilmektedir.

5.5.2.1 Matematik öğrencileri okulda

Okulda, öğretmen 30 öğrencisine 30’ar problem verir. Ama 900 cevabı kontrol etmez. Onun yerine, öğretmen her bir öğrenciye farklı bir numara tayin edip, öğrenci sınıfın

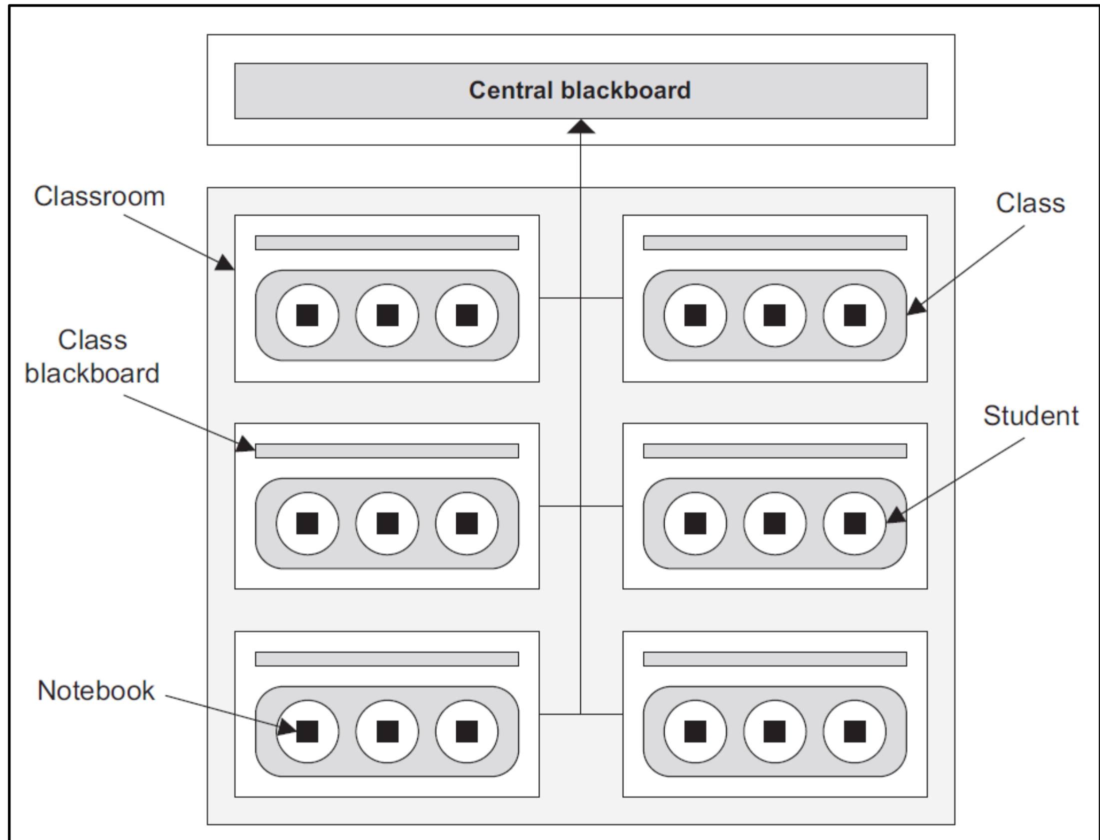
önüne geçip o numaralı problemi çözer. Öğrenci defterden tahtaya çalışmasını geçirir ve öğretmen tahtadakine göre not verir.

Bir OpenCL aygıtı, bir okul dersliği gibidir. Her derslik, matematik problemleri yapan öğrenciler içerir. Bir sınıftaki öğrenciler aynı tahtayı paylaşır, ama her öğrenci ayrı bir deftere sahiptir. Aynı sınıftaki öğrenciler tahtada beraber çalışabilir, ama farklı sınıfların öğrencileri beraber çalışamaz.

Zorluk şuradadır: Bu dersliklerin hiçbiri bir öğretmene sahip değildir. Aynı şekilde, okulda her öğrenci, aynı matematik problemini farklı değerlerle çalışır. Örneğin, eğer problem iki sayıyı toplamayı içeriyorsa, öğrencinin biri 1 + 2 toplamını, başka biri 3 + 4 toplamını ve başka bir öğrenci 5 + 6 toplamını yapar. Bir derslikteki bütün öğrenciler hesaplamalarını tamamladığı zaman ayrılabilirler. Sonra tahta silinecek ve yeni bir sınıf öğrencileri gelecek ve aynı probleme farklı değerler çalışacaktır.

Her öğrenci hangi problemi çözeceğini bilir, ama hangi değerler olacağını bilmez. Her derslikte tahta önce boştur, bundan dolayı öğrenciler tüm okul için değerleri içeren merkezi bir tahtaya gider. Bu merkezi tahta dersliklerdeki tahtalara kıyasla çok daha büyüktür ve değerleri okumak öğrenciler için büyük zaman alır. Şekil 5.6'da, derslikler, sınıflar, öğrenciler, defterler ve tahtalar arasındaki ilişki yansıtılmaktadır.

Şekil 5.6: Okul yapısı



Kaynak: (Scarpino, M., 2011)

Matematik problemi için, her öğrenci sadece iki kez merkezi tahtaya gidecek – bir kez problemin değerini okumak için ve bir kez de nihai cevaplarını yazmak için. Merkezi tahta çok uzakta olduğu için, öğrenciler, defterleri ve derslik tahtalarını kullanarak asıl çözümlerini yapar. Nihai cevapların hepsi merkezi tahtada olduğu andan itibaren okulda gün biter.

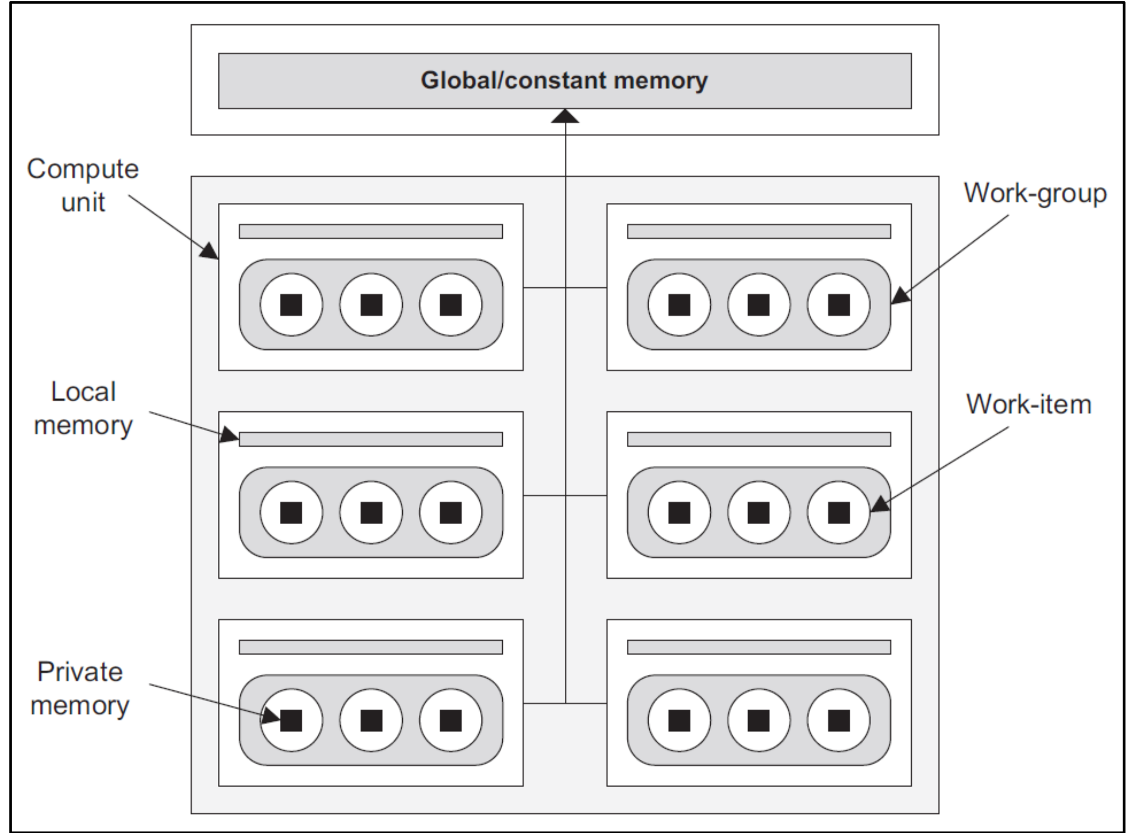
Farklı sınıfların öğrencileri birbirleriyle konuşamaz. Bundan dolayı sınıf 1'deki öğrenciler, sınıf 2'deki öğrencilerin bitirdiği zamanı bilemez. Bir sınıfın bitirdiğinden emin olmanın tek yolu, okul gününün bittiği zamandır. Sınıf ve derslik arasında ayrım yapmak önemlidir. Derslik, tahtalı fiziksel bir alandır. Sınıf, dersliği işgal eden öğrenciler grubudur. Bir sınıf derslikten ayrıldığında diğeri girebilir.

Bu organizasyonu sağlamak için, her sınıfı diğer sınıftan ayıran bir tanımlayıcı vardır. Her öğrenci, iki tanımlayıcıya sahiptir: Biri sınıftaki diğer öğrencilerden ayırt etmek için ve diğer okuldaki diğer öğrencilerden ayırt etmek içindir. Örneğin bir öğrenci, 22 sınıf id'sine ve 732 okul id'sine sahip olabilir.

5.5.2.2 Bir aygıtta çekirdek işlemi

Benzetmede, okul, bir OpenCL aygıtına karşılık gelir ve matematik problemi çekirdeği temsil eder. Her öğrenci, bir çalışma öğesine karşılık gelir ve her sınıf bir çalışma grubuna karşılık gelir. Bir derslik hesaplama ünitesine (işlemci çekirdeği) karşılık gelir; Ve tıpkı her dersliğin bir sınıf tarafından işgal edilebildiği gibi, her bir hesaplama ünitesi bir çalışma grubu tarafından işgal edilebilir. Şekil 5.7 bunu yansıtmaktadır.

Şekil 5.7: OpenCL Aygıt Yapısı



Kaynak: (Scarpino, M., 2011)

Kimlik numaraları, OpenCL'de büyük bir rol oynar ve her çalışma öğesi iki id'e sahiptir: Global id ve yerel id. Global id, çekirdeği işleten tüm çalışma öğelerini birbirinden ayırmada çalışma öğesini tanımlar. Yerel id, çalışma grubunda kalan diğer çalışma öğelerini birbirinden ayırmada çalışma öğesini tanımlar. Farklı çalışma gruplarındaki çalışma öğeleri aynı yerel id'e sahip olabilir, ama asla aynı global id'e sahip olamaz. Okul benzetmesinde, bir çalışma öğesinin yerel id'si bir öğrencinin sınıf id'sine karşılık gelir ve bir çalışma öğesinin global id'si öğrencinin okul id'sine karşılık gelir.

OpenCL aygıt modeli, dört adres boşluğu tanımlar:

a. Global Bellek: Bütün aygıt için veri depolar.
b. Sabit Bellek: Global belleğe benzer, ama salt okunurdur.
c. Yerel Bellek: Bir çalışma grubundaki çalışma öğeleri için veri depolar.
d. Özel Bellek: Bir çalışma öğesi için veri depolar.

Benzetmede, merkezi tahta, kendinden okunabilen ve host ve aygıt ikilisi ile yazılabilen global belleğe karşılık gelir. Host uygulamasında, aygıta veri transfer ettiği zaman, veri global bellekte depolanır. Benzer şekilde, host bir aygıttan veri okuduğu zaman, veri aygıtın global belleğinden gelir. Bu bellek, bir OpenCL aygıtındaki en

büyük bellek bölgesidir, ama erişim aynı şekilde çalışma öğeleri için en yavaş olanıdır.

Çalışma öğeleri için, global / sabit bellek erişim kıyaslanırsa yerel bellek çok daha hızlı erişilebilir (100x); ve yerel bellek blokları, her derslikteki tahtalara karşılık gelir. Yerel bellek, global / sabit bellek gibi büyük değil, ama erişim hızından dolayı, ara sonuçları depolamada çalışma öğeleri için iyi bir yerdir. Aynı sınıftaki öğrencilerin derslik tahtasında birlikte çalışabildiği gibi, aynı çalışma grubundaki çalışma öğeleri yerel belleğin aynı bloğuna erişebilir.

Bir OpenCL aygıtındaki özel bellek, her öğrencinin bir matematik problemini çözmede kullandığı deftere karşılık gelir. Her çalışma öğesi, özel belleğine özel erişime sahiptir ve yerel / global / sabit belleğe erişime kıyasla bu belleğe daha hızlı erişilebilir. Ama bu adres boşluğu, diğer adres boşluklarına kıyasla çok daha küçüktür, bundan dolayı onun çok fazla kullanılmaması önemlidir.

OpenCL kullanıldığı zaman, kaç çalışma öğesinin bir çekirdek için oluşturulabildiğini öğrenmek için, benzetmede bu açık, istenildiği kadar çalışma öğesi ve çalışma grupları üretilebilinir. Bununla birlikte, eğer aygıt sadece çalışma grubu başına N tane çalışma öğesi ve M tane hesaplama ünitesi içeriyorsa, M x N çalışma öğesi herhangi bir verilmiş zamanda çekirdeği işletecektir.

5.5.3 Örnek Host Uygulaması

Genelde, bir host uygulamasının ilk adımı, bir çekirdeği işletecek olan her aygıt için cl_device_id'i elde etmektir. Add_numbers uygulaması, birincil platformla ilişkilendirilmiş birincil GPU aygıtına erişir. Bu, takip eden kodda gösterilmiştir.

```
clGetPlatformIDs(1, &platform, NULL);
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
```

Buradaki, platform yapısı, OpenCL runtime ile tanımlanan birincil platformu tanımlar. Bir platform, bir imalatçı kurulumunu tanımlar. Bir sistem, NVIDIA platformu ve AMD platformuna sahip olabilir. Device yapısı, platformla ilişkilendirilen birincil erişilebilir aygıta karşılık gelir. İkinci parametre, CL_DEVICE_TYPE_GPU olduğu için, bu aygıt bir GPU olmalıdır.

Sonra, uygulama sadece bir aygıt içeren bir context yaratır.

```
context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
```

Context'i yarattıktan sonra, uygulama, add_numbers.cl dosyasındaki kaynak kodundan bir program oluşturur. Özelde, kod, program_buffer isimli char dizisinde dosyanın içeriğini okur ve sonra clCreateProgramWithSource'i çağırır.

```
program = clCreateProgramWithSource(context, 1,
                (const char**)&program_buffer, &program_size, &err);
```

Program oluşturulduğu andan itibaren, kaynak kodu contextdeki aygıtlar için derlenmelidir. Bunu başaran fonksiyon, clBuildProgram'dır ve takip eden kod, add_numbers uygulamasında nasıl kullanıldığını gösterir.

```
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

Dördüncü parametre, derlemeyi düzenleyen seçenekleri kabul eder. Bunlar, gcc ile kullanılan flaglara benzer. Örneğin, -DMACRO=VALUE seçeneği bir makro tanımlayabilir ve –cl-opt-disable ile optimizasyon kapatılabilir.

cl_program derlendikten sonra, çekirdekler fonksiyonlarından yaratılabilir. Takip eden kod, add_numbers isimli fonksiyondan cl_kernel'i yaratır.

```
kernel = clCreateKernel(program, "add_numbers", &err);
```

Uygulama, bu çekirdeği göndermeden önce, bir hedef aygıtında bir komut kuyruğu oluşturmaya ihtiyaç duymaktadır. Doğru konfigürasyonla, komut kuyruğu, düzensiz çekirdek işletimi ve / veya profillemeyi destekler. Profilleme, çekirdeğin işletimi için zaman ayarlarına imkan verir. Takip eden kod, bununla birlikte, profilleme ve düzensiz işletimi desteklemeyen cl_command_queue'i yaratır.

```
queue = clCreateCommandQueue(context, device, 0, &err)
```

Bu noktada, uygulama, bir OpenCL host uygulaması tarafından ihtiyaç duyulan bütün veri yapılarını (aygıt, kernel, program, komut kuyruğu ve context) oluşturur. Şimdi, o, takip eden kodla bir aygıta çekirdeği konuşlandırır.

```
global_size = 8;
local_size = 4;
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size,
                        &local_size, 0, NULL, NULL);
```

Hostta çalışan OpenCL fonksiyonlarından en önemlisi, clEnqueueNDRangeKernel, fonksiyonudur. O, sadece aygıtlara çekirdekleri konuşlandırmaz, aynı zamanda kaç çalışma öğesinin çekirdeği işletmek için oluşturulacağını (global_size) ve her çalışma grubunda çalışma öğelerinin sayısını tanımlar (local_size).

Bu örnekte, çekirdek, her biri dört çalışma öğeli iki çalışma grubuna bölünmüş sekiz çalışma öğesi ile işletilir. Benzetmeye bakarsak, bu, sekiz öğrencinin dörtlü olarak iki sınıfa bölündüğü bir okula karşılık gelir.

5.5.4 Örnek Çekirdek Uygulaması

Örnek uygulama, binlerce hatta milyonlarca çalışma öğesi oluşturabilir, ama 64 tane sayının toplama işi için sekiz çalışma öğesi yeter. Add_numbers.cl program dosyası bu işlemi yapan add_numbers adında bir fonksiyon içerir. Bütün çekirdek fonksiyonları gibi, void döndürür ve isminin önüne __kernel tanımlayıcısı gelir.

Çekirdek, 64 değer ve toplama yapacak sekiz çalışma öğesine sahiptir. Her çalışma öğesi, sekiz değerinin toplamını hesapladıktan sonra, tüm grup için yekün toplam hesaplanır. Sonda çekirdek, çekirdeğin işletildiği her çalışma grubu için birer tane olmak üzere iki toplam döndürür.

Çalışma öğesi, add_numbers işletimine başladığı zaman, erişmeye ihtiyaç duyduğu değerleri içeren global bellek bölgesinin hangisi olduğuna karara girişir. Bunu, takip eden kodla başarır.

```
global_addr = get_global_id(0) * 2;
```

İlk kod satırı, bütün diğer işletilen çalışma öğelerinden onu ayıran global ID ile çalışma öğesini elde eder. Bu ID ile, verinin yükleneceği global belleğin adresi olan global_addr'i hesaplar.

Sonra, çalışma öğesi, özel belleğe global verisini yükler. Benzetmeye bakarsak, bu bir öğrencinin okuldaki uzak tahtaya gidip deftere değerleri kopyalaması gibidir. Takip eden kod, bunun nasıl işlediğini gösterir.

```
input1 = data[global_addr];
input2 = data[global_addr + 1];
sum_vector = input + input2;
```

Çekirdek kodunda, input1, input2 ve sum_vector float4 veri tipindedir. Bu, bir vektör tipidir ve dört floatlı diziye benzer, ama önemli farkı: float4 ile işlem yapıldığında, tüm dört float'un saat yönünde işletilmesidir. Eğer hedef aygıt, vektör işlemlerini destekliyorsa, bu kodun son satırı, eşzamanlı 4 float toplamını gerçekleştirecektir.

Her çalışma öğesi, benzetmedeki bir derslik tahtasına karşılık gelen yerel belleğe onun nihai toplamını depolar. Bu, aşağıdaki gibi oluşturulur.

```
local_addr = get_local_id(0);
local_result[local_addr] = sum_vector.s0 + sum_vector.s1 +
                           sum_vector.s2 + sum_vector.s3;
```

İlk satır, sekiz değerin toplamının depolanacağı yeri çalışma öğesine iletir. İkinci satır, toplamı hesaplar ve yerel belleğe onu yerleştirir.

Her çalışma grubu dört çalışma öğesi içerir. Bundan dolayı yerel belleğin her bloğu, dört kısmi toplam içerir. Bu kısmi toplamları ötekilerle toplamak için, bir çalışma öğesi, grup için değerleri okumak üzere ve ötekilere onları ekleyerek bir sonucuna ulaşmak için atanmıştır. Bu, takip eden kodla başarılır.

```
if(get_local_id(0) == 0) {
    sum = 0.0f;
    for(int i = 0; i < get_local_size(0); i++) {
        sum += local_result[i];
    }
    group_result[get_group_id(0)] = sum;
}
```

grup_result verisi, global bellekte depolanır. Bu önemlidir, çünkü host uygulaması sadece bu adres boşluğunda depolu değerleri okuyabilir. Add_numbers olayında, uygulama, grup_result'taki iki toplamı okur, birbirine ekler ve doğru cevap için çekirdeğin çıktısını kontrol eder. Host uygulaması, sonucun yazımı ve OpenCL operasyonlarının gerçeklemede kullanılan yapıları serbest bırakma ile tamamlanır.

6. ÖRNEK UYGULAMALAR

OpenCL çatısının hesap yoğun işlemlerdeki performansını değerlendirmek amacıyla OpenCL avantajlarını ve işlevselliğini koruyarak Tablo 6.1'de özellikleri gösterilen hesaplama aygıtları örnek uygulamalar başlığı altındaki 13 farklı işleme sokularak yürütme süreleri ölçülmüştür. Örnek uygulamaların yürütüldüğü 3 farklı sistemin özellikleri Tablo 6.2'de gösterilmektedir.

Tablo 6.1: Hesaplama aygıtlarının özellikleri

	Intel Core 2 Duo T9550	**ATI Radeon HD 5970 X2**	**2 x Nvidia Tesla M2050**	**Kıyaslama**
Genel bellek boyutu	-	2 GB	**6 GB**	3x
Genel bellek bant genişliği	1.06 GB/s	256 GB/s	**296.84 GB/s**	280x
Hesaplama birimi sayısı	2	**3200**	896	1600x
İşlemci yonga transistörlerinin sayısı	410 milyon	4.3 milyar	**6.2 milyar**	15x
Saat frekansı	**2.66 GHz**	725 MHz	575 MHz (çekirdek) 1.15 GHz (iş hattı)	3.7x
Tepe performans değeri	21.3 GFlops	**4.64 TFlops** (Tek duyarlı)	2.06 TFlops (Tek duyarlı)	218x
Fiyat	**384 $**	743 $	5670 $	15x
Maksimum güç	**35 W**	294 W	450 W	13x
Güç/Performans (1 GFlops için)	1.64 W	**0.06 W**	0.22 W	27x
Fiyat/Performans (1 GFlops için)	18.03 $	**0.16 $**	2.75 $	113x

Kaynak: Bu tablo Ersin Kuzu tarafından hazırlanmıştır.

Tablo 6.2: Test sistemlerinin özellikleri

	CPU	ATI GPU	NVIDIA GPU
Bellek	4 GB	4 GB	22 GB
İşlemci	Intel Core 2 Duo T9550	AMD Phenom II X4 965	2 x Intel Xeon X5570
Ekran Kartı	ATI Mobility Radeon HD 3650	ATI Radeon HD 5970 X2	2 x Nvidia Tesla M2050 GPU
Sabit Disk	Kingston HyperX 3K SSD – 240 GB	OCZ RevoDrive 3 PCI Express SSD – 480 GB	1690 GB
İşletim Sistemi	Ubuntu 14.04 – 64 bit	Ubuntu 14.04 – 64 bit	Linux – 64 bit
OpenCL	v1.2	v1.2	v1.2

Kaynak: Bu tablo Ersin Kuzu tarafından hazırlanmıştır.

Günümüzün en popüler programlama dillerinden olan Java nesne yönelimli programlama dili ile bu tez çalışmasında kullanılan OpenCL paralel programlama dilinin en popüler 3 kaynaktaki tarama sonuçları Tablo 6.3'de gösterilmektedir.

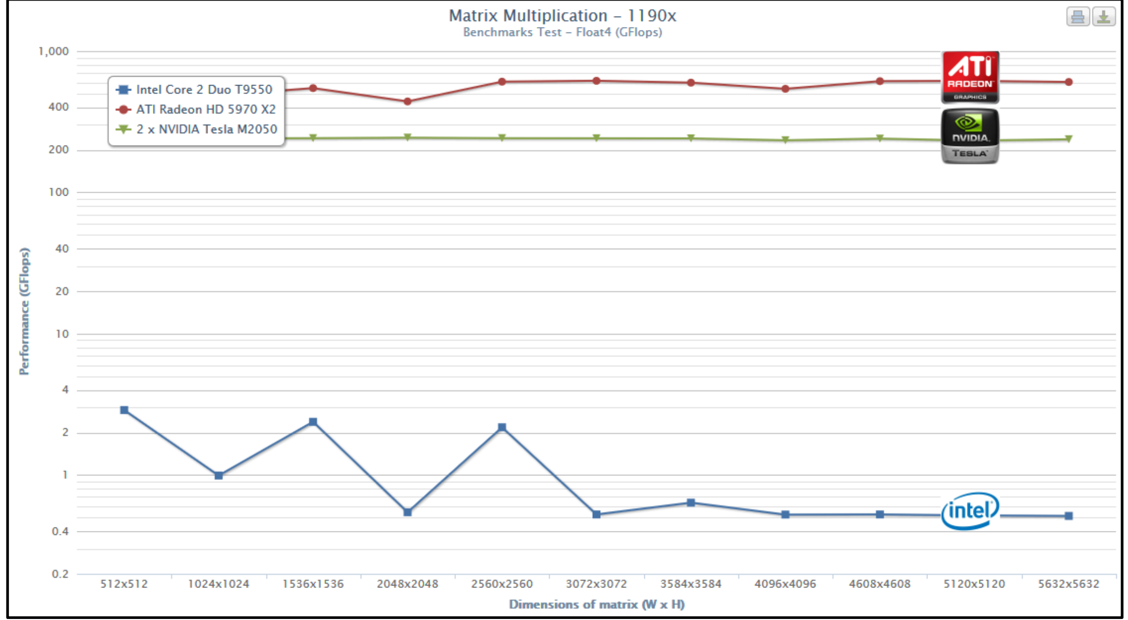
Tablo 6.3: Programcı sayıları

	OpenCL	Java
https://www.linkedin.com	7427 kişi	3863535 kişi
https://www.odesk.com	43 kişi	35535 kişi
https://www.elance.com	21 kişi	9893 kişi

Kaynak: Bu tablo Ersin Kuzu tarafından hazırlanmıştır.

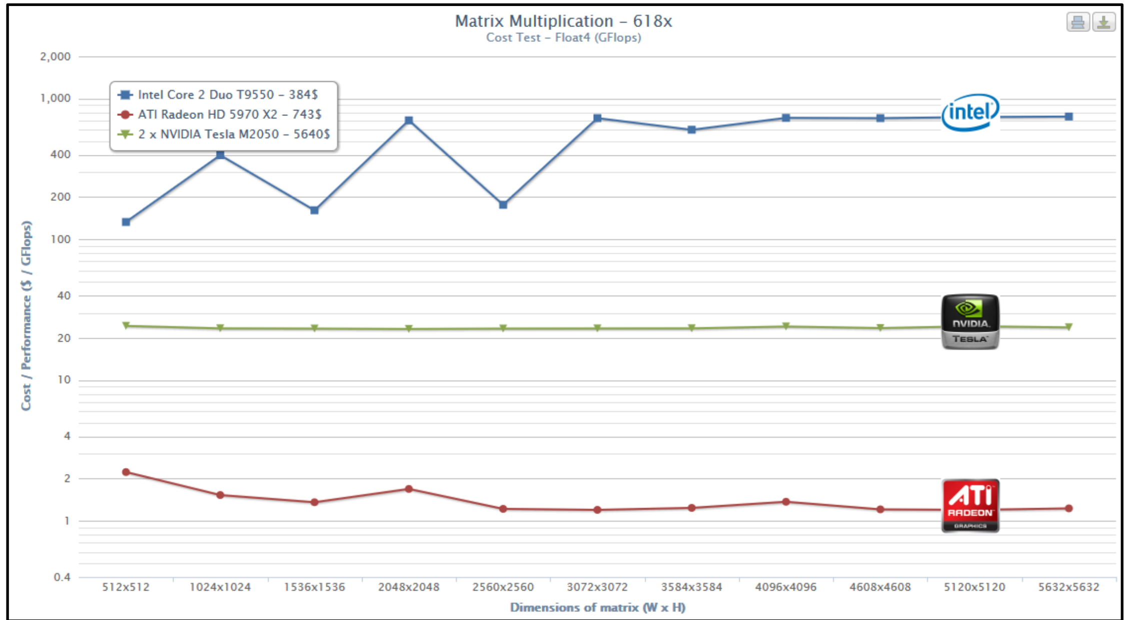
Hesaplama ağırlıklı algoritmaların performansını GPU ile arttıran çalışmalar bu başlık altında önce algoritmaların kısa bir açıklaması ardından hesaplama aygıtlarının problem aralıklarına göre hesaplama süreleri grafik halinde gösterilmektedir. Hesaplama aygıtlarının matris çarpma algoritması verileri referans alınarak hazırlanan Şekil 6.1'de Performans, Şekil 6.2'de Maliyet, Şekil 6.3'de Enerji tüketimi grafikleri verilmiştir.

Şekil 6.1: Performans (GFlops) test sonuçları



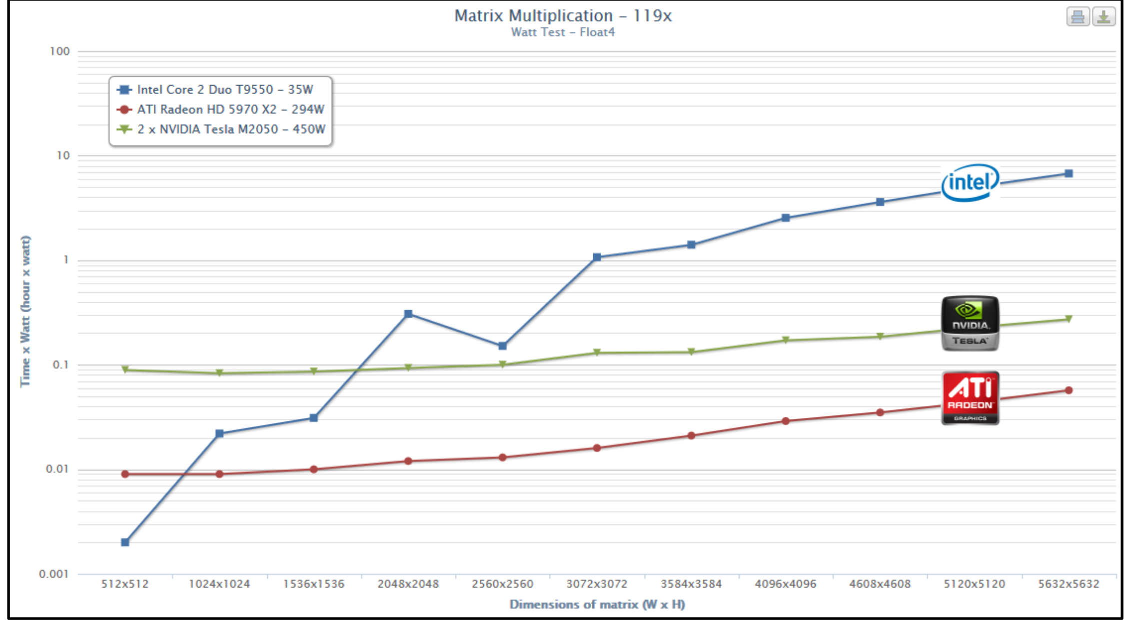
Kaynak: Bu şekil Ersin Kuzu tarafından hazırlanmıştır.

Şekil 6.2: Maliyet ($) test sonuçları



Kaynak: Bu şekil Ersin Kuzu tarafından hazırlanmıştır.

Şekil 6.3: Enerji tüketim (Watt) test sonuçları



Kaynak: Bu şekil Ersin Kuzu tarafından hazırlanmıştır.

6.1 BİNOM OPSİYON (BINOMIAL OPTION) FİYATLAMA MODELİ

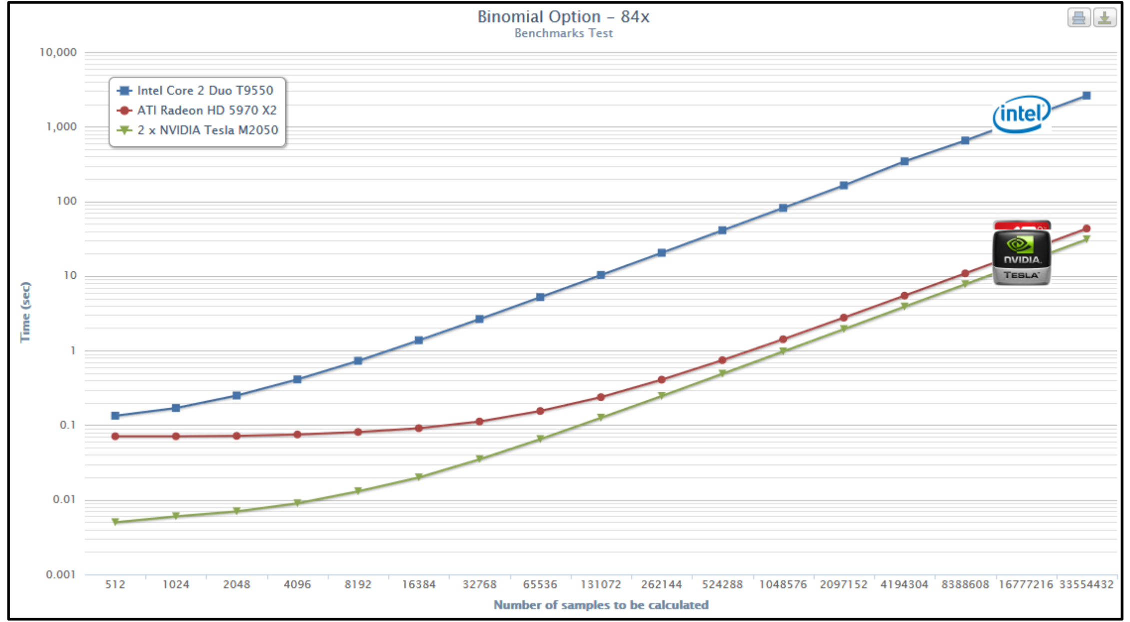
Opsiyon fiyatlandırma modeli opsiyon fiyatlarını etkileyen faktörleri girdi olarak alıp hesaplama işlemi ya da bir matematik formülüdür. Modelde çıktı bir opsiyonun teorikteki gerçek değerini verir. Model gerektiği gibi çalıştığı sürece opsiyon piyasa fiyatı teorik gerçek değere eşit olacaktır. Bu süreç opsiyon fiyatlandırma olarak adlandırılır. Bu hesaplamada kullanılan Binom Opsiyon Fiyatlama ve Black Scholes Opsiyon Fiyatlama iki önemli modeldir. Binom opsiyon formülden ziyade bir hesaplama sürecine dayanır. Black Scholes matematiksel formüle dayanır. Ancak her iki modeldeki amaç da opsiyonun işlem görmesi gereken değeri hesaplamaktadır.

Binom opsiyon fiyatlama modeli, belli bir zamanda, bir sonraki dönemde hisse senedi fiyatlarının belirli bir oranda artacağı veya azalacağını varsayarak, yatırımcının kararlarına yön veren bir modeldir. Her modelde olduğu gibi binom opsiyon fiyatlama modelinde de varsayımlar vardır. Bunlar (Simplilearn, 2013);

a. Piyasalar mükemmeldir. Vergi ve komisyonlar ihmal edilir. Açığa satışlar üzerinde bir sınır yoktur. Varlıklar sonsuz kere bölünebilir.
b. Borçlar belli bir faiz oranı üzerinden alınıp verilir.
c. Hisse senetlerinin fiyat düşüş ve artış oranları ile dönem faiz oranları bilinmektedir.

Bu model Avrupa veya Amerika tipi hisse senedi alımı veya satımında opsiyonların fiyatlarının belirlenmesinde çoğunlukla tercih edilmektedir.

Şekil 6.4: Binom Opsiyon Fiyatlama test sonuçları

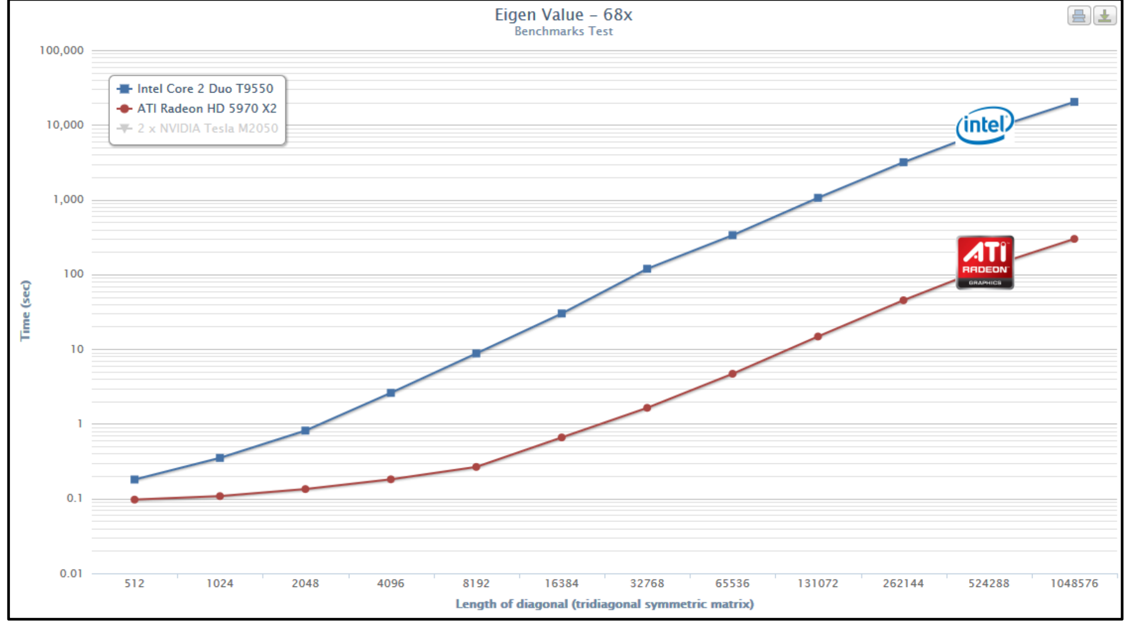


Kaynak: Bu şekil Ersin Kuzu tarafından hazırlanmıştır.

6.2 ÖZDEĞERLER (EIGEN VALUE) PROBLEMİ

Özdeğerler ve özvektörleri, fiziksel bir sistemin sahip olabileceği değerlere göre davranışlarını bulmak için önemlidir. Bu değerler sisteme ait özel bir enerji, özel bir frekans değeri, dalgaların girişimi veya kuvvet dengesinin sağlandığı bir duruma ait olabilir. Özvektörler ve özdeğerler, diferansiyel denklemler içeren denklem sistemlerinin çözümlerinde, sınır – değer problemlerinde ortaya çıkabilir. Bu tür denklemlere, kuantum mekaniğinde elektriksel bir potansiyel içinde bulunan bir parçacığın enerjisini hesaplarken, elastik çarpışma problemlerinde, akışkanlar mekaniğinde, titreşim yapan cisimlerin hareketlerinde sıkça karşılaşılır (Ankara Üniversitesi, 2013).

Şekil 6.5: Özdeğerler test sonuçları



Kaynak: Bu şekil Ersin Kuzu tarafından hazırlanmıştır.

6.3 FLOYD – WARSHALL ALGORİTMASI

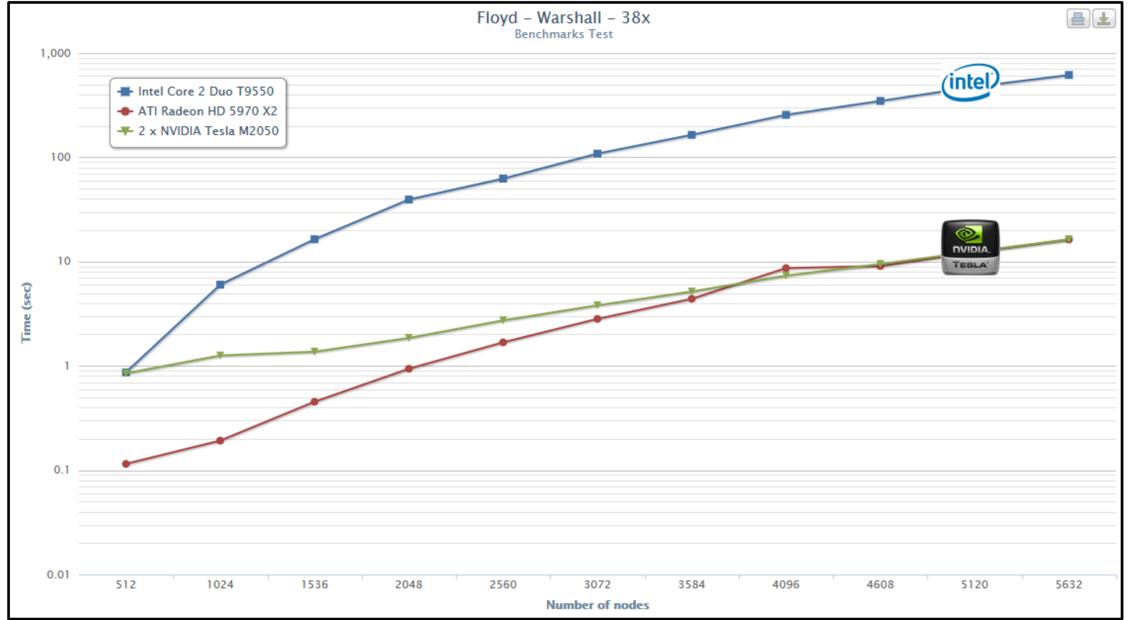
Bilgisayar bilimlerinde algoritma analizi konusunda sıkça kullanılan bir algoritmadır. Algoritma belirli bir çizge (graf - graph) üzerinde başlangıçtan bitiş düğümüne en kısa yoldan ulaşım yolunu bulmayı amaçlamaktadır. Bu nedenle maksimum akış problemlerinin çözümünde kullanılan bir algoritmadır. Algoritma, algoritmayı bulan iki kişinin adı ile anılmaktadır.

Algoritma bir çizge üzerinde gidilebilecek düğümlerin komşuluk listesini çıkaran ve düğümlere olan uzaklıkların bulunduğu bir matris üzerinde çalışır. Algoritma matris üzerindeki işlemleriyle en kısa yolu bulmayı hedefler. Algoritma düğümler arası uzaklıkları her işlemde güncellediği ve matris üzerinde çalıştığından özyinelemeli bir algoritmadır.

Floyd – Warshall algoritması aşağıdaki amaçlar için kullanılabilir (Şeker, Ş.E., 2009):

a. Yönlü çizgelerde en kısa yolun bulunması için.
b. Bir düğümden gidilebilecek diğer düğümlerin bulunmasında.
c. Düzenli ifadelerde tekrarlı olmayan yapı tespiti için kullanılabilir. Kleen yıldızı şeklinde tekrar eden ifadelerin bulunmasına yarar.
d. Ağ programlamasında özellikle yönlendirici algoritmalarında en kısa yolun tespiti için kullanılır.
e. Yönsüz bir çizgenin, iki parçalı çizge olup olmadığının bulunmasında kullanılabilir.

Şekil 6.6: Floyd – Warshall test sonuçları



Kaynak: Bu şekil Ersin Kuzu tarafından hazırlanmıştır.

6.4 K – MEANS KÜMELEME (CLUSTERING) ALGORTİMASI

K – Means kümeleme algoritması n tane verinin k tane gruba ayrılması olayıdır. Bu gruplama için elemanların grup merkezine mesafelerini gösteren bir metrik oluşturulur. Gruplar verilerin birbirine yakınlık ve uzaklıklarına göre oluşturulur.

Diğer kümeleme algoritmaları bazı tür verilerde daha iyi sonuçlar vermesine karşın K – Means kümeleme algoritmaları her çeşit veride kabul edilebilir sonuçlar verir. Algoritmanın en büyük dezavantajı yerel optimumlarda kalarak genel optimumlara ulaşamamasıdır (Yünel, Y., 2010).

Tüm veriler sayısal ifadelere dönüştürülür. Algoritma başlangıçta rastgele ya da belirli yöntemlerde seçilen k tane küme merkezi belirler. Sonra her eleman yakın oldukları sınıflara yerleştirir. Sınıflardaki elemanların ortalama değerlerini alınıp, bu değerler sınıfların merkezlerini belirler. Sonra her elemanın yakın olduğu sınıflara göre tekrar gruplanır. Eski sınıf merkezleri yenilerine eşit olana ya da belirtilen çevrim sayısınca algoritma bu işlemleri yeniler.

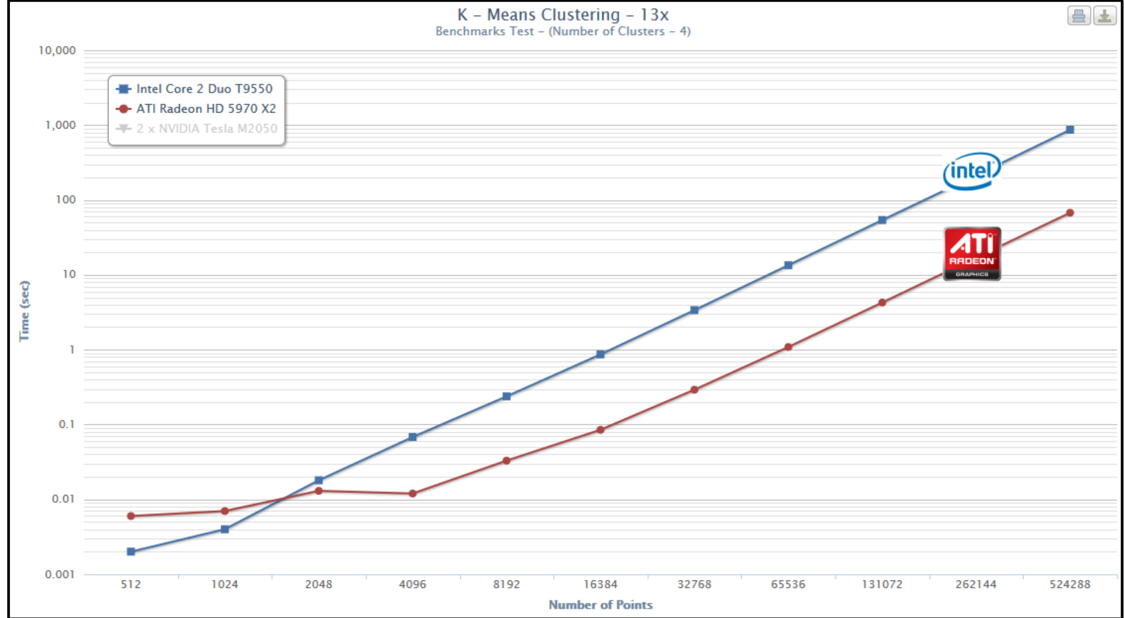
K – Means algoritmasının işlem basamakları şu şekilde özetlenebilir (Özkan, H., 2013):

a. K adet başlangıç sınıfının belirlenmesi gerekir.
b. Her pikselin seçilen merkez noktalara uzaklığı belirlenir. Bu uzaklığa göre pikseller en yakın oldukları sınıfa yerleştirilirler.

c. Aynı kümeye ait piksellerin ortalaması alınır. Bu ortalamalar ilgili kümenin yeni piksel değerlerini belirler.

d. 2. Adımda bulunan merkez değerleri ile 3. Adımdaki merkez değerleri değişmeyene kadar bu işlem tekrarlanır.

Şekil 6.7: K – Means Kümeleme test sonuçları



Kaynak: Bu şekil Ersin Kuzu tarafından hazırlanmıştır.

6.5 MANDELBROT KÜMESİ VE FRAKTAL SİMÜLASYONU

Bilgisayar teknolojilerindeki gelişimeler rakamlarla dolu tabloların yerine görüntüleri getirmiştir. İlk başlarda kabaca olan görüntüler teknolojik gelişmelere paralel olarak büyüleyici şekillere dönüşmüştür. Bu şekiller doğadaki fraktal objelerin aynı özelliklerini göstermektedir. İnsan gözünün bu yapıları saptayabilmesi mümkün olmuştur. Fraktal görüntüleri karmaşık sayılardan oluşan polinomların yinelenmesiyle gerçekleştirilir. Başlangıçtaki veriye bitişte başka verinin karşılık gelmesiyle özyineli metot her seferinde bitiş verisini yeni başlangıç verisi olarak kabul eder ve bu işlem tekrar edilir (Onurlu, S., 2003). Bunu ifade edersek:

(Giriş / Başlangıç verisi / Transformasyon / Çıkış)

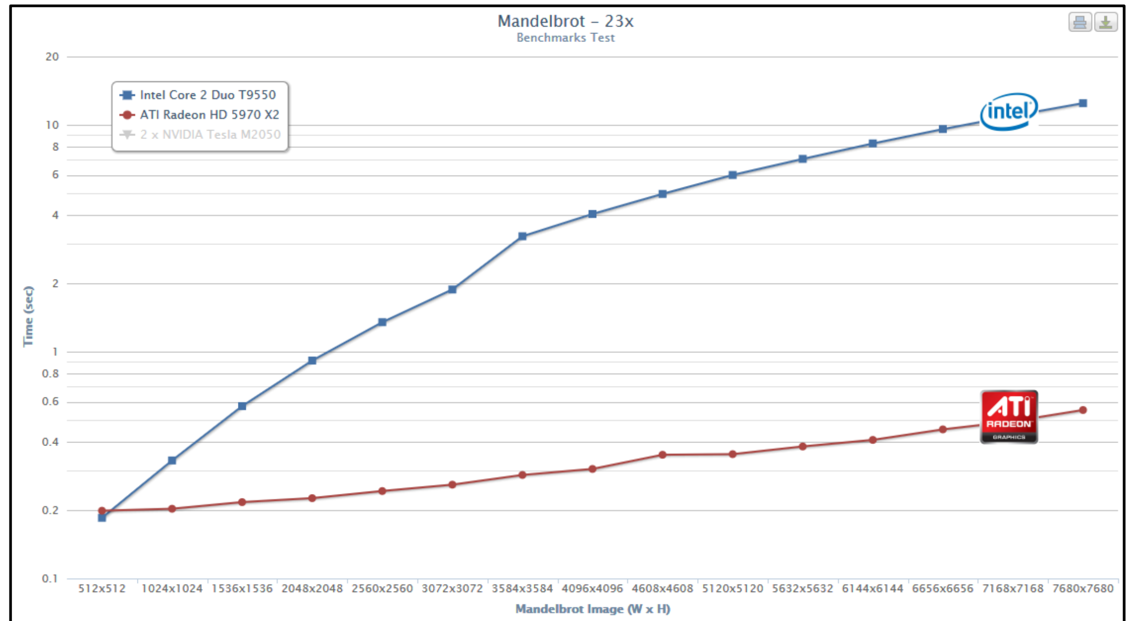
Karmaşık sayılarda dinamik sistemler teorisindeki en basit gelişim kanunu kullanılır. Bu da farklı sonuçlar doğurabilir. x ve y birer karmaşık sayı olmak üzere, lineer olmayan olayları inceleme olanağı sunan en basit form x->x2+y polinomudur. Yineleme metodu x başlangıç noktasını x2+y noktasına dönüştürme işlemiyle başlar. Sonra x2+y'nin değeri başlangıç olarak alınıp yeni (x2+y)2+y noktası elde edilir. Bitiş noktası başlangıç noktası olarak alınır ve işlem devam eder. Böylelikle x noktasının yörüngesi olarak adlandırılan dizi oluşturulur. Bilgisayar bu işlemi birçok kere

kolaylıkla tekrarlayabilir. Noktalar hareketlerine göre renklendirilerek olağanüstü güzelliklerde fraktal görüntüler elde edilebilir. Bunlara örnek olarak mandelbrot kümesi parametreler düzlemde, julia kümeleri değişkenler düzleminde oluşur.

Mandelbrot kümesi kompleks sayılar dinamiği açısından geometrideki çember kadar doğal bir nesnedir. x değerinin değişik y değerleriyle değiştirilmesiyle parametreler düzlemi üzerinde oluşur. M kümesi yakından incelendiğinde harika bir zenginliğe sahiptir. Ayrıntılar son derece sofistike bir biçimde düzenlenmiş filamentlerden oluştuğu görülür. Mandelbrot kümesi incelendiğinde içinde her bölgenin aynı kümenin bütününün küçük birer kopyaları olduğu görülür. Bu kopyalar hafif farklılık gösterse bile farklı parçalar aynı düzen içerisinde yeniden oluşturulur. Her kopyanın farklılık gösterdiği farklı boylarda bir çevreye gömülü haldedirler.

Faraktalların astronomi, geometri, petrol araştırmaları, tıp, finans, biyoloji, mekanik, ekonomi gibi düzensizlikler ve değişkenlerle uğraşan ilgili bütün alanlarda uygulamaları bulunmaktadır. Bu konularla olduğu kadar türbülans, DNA yapısındaki uygulamalar, borsanın seyri, kırıklardaki yayılma kavramlarıyla da yakından ilgilidir.

Şekil 6.8: Mandelbrot Fraktal Simülasyon test sonuçları

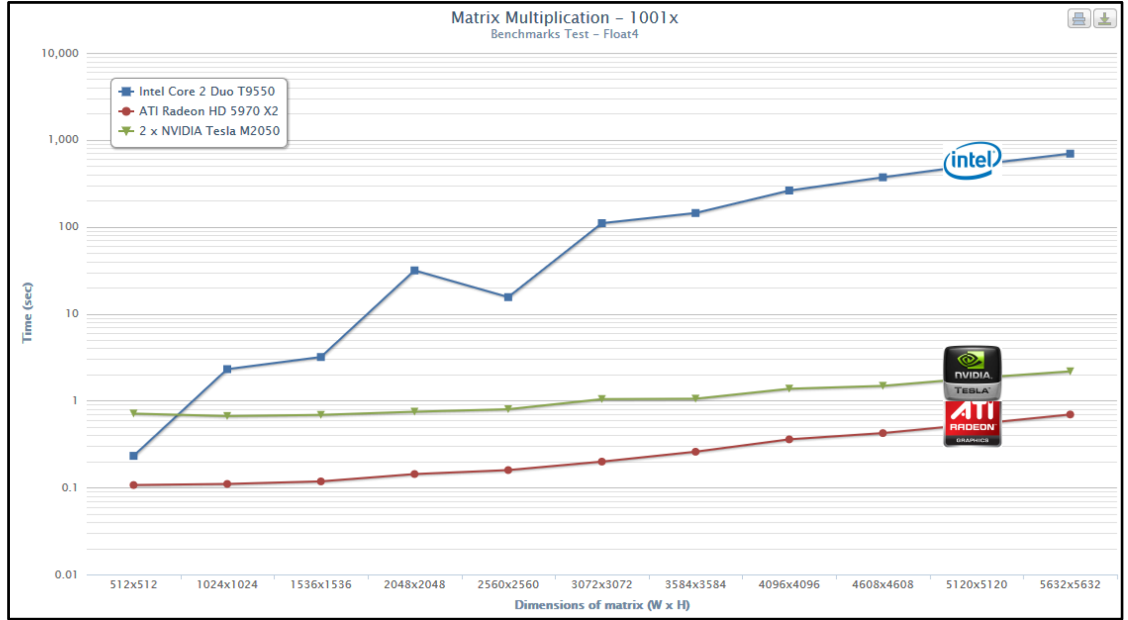


Kaynak: Bu şekil Ersin Kuzu tarafından hazırlanmıştır.

6.6 MATRİS ÇARPMA

Matematikte matris çarpımı iki matris ile yapılan ve başka bir matris oluşturan ikili işlemdir. Temel aritmetiğe uygun olarak karmaşık sayılar veya reel sayılarda çarpma yapılabilir. Matrisler sayı dizilerinden oluşur. Uygulamalı matematik, fizik ve mühendislik matris çarpımının kullanım alanlarıdır.

Şekil 6.9: Matris Çarpımı test sonuçları



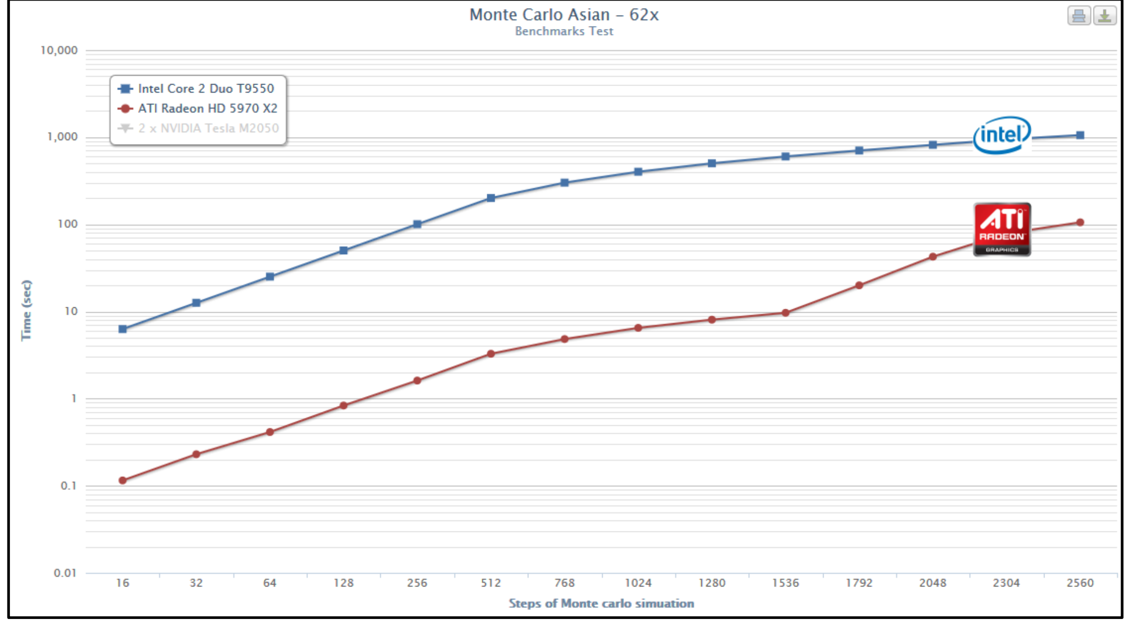
Kaynak: Bu şekil Ersin Kuzu tarafından hazırlanmıştır.

6.7 MONTE CARLO SİMÜLASYONU İLE OPSİYON FİYATLAMA

Rastgele sayılar kullanılarak üretilen istatistiksel sonuçların simülasyonlarında Monte Carlo metodundan faydalanılmaktadır. Atom bombası geliştirilirken bombanın patlamasıyla dağılan nötronlara karşı modellenen kalkanlarda günümüzde kullanılmaktadır.

Girdilerin belirli olmayan bir şekilde gelmesi bekleniyor ise ve dağılım fonksiyon ile hesaplanabiliyorsa kullanılır. Bu model rastgele sayıları baz alarak sistemleri tahmini bir şekilde modeller. Borsa modelleri, deney aletlerinin simülasyonu, hücre simülasyonu, doğal olayların simülasyonu, dağılım fonksiyonları, atom ve fizik molekülleri, sayısal analiz, nükleer fizik ve yüksek enerji fiziği modellerinin simülasyonu kullanım alanlarıdır.

Şekil 6.10: Monte Carlo Simülasyon test sonuçları



Kaynak: Bu şekil Ersin Kuzu tarafından hazırlanmıştır.

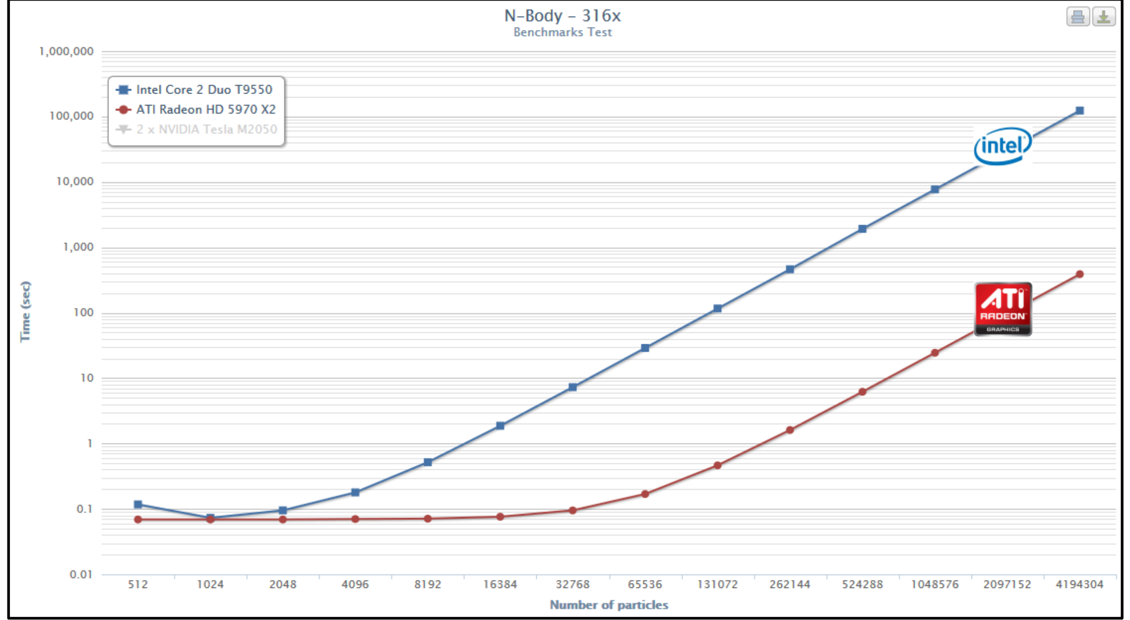
6.8 CİSİM (N – BODY) PROBLEMİ

Fiziksel bir sistemin davranışlarının incelenmesi birçok bilimsel çalışmaya konu olmuştur. Klasik N – cisim problemi genellikle birbirini etkileyen cisimlerden oluşan fiziksel sistemlerde olduğu gibi, n tane oluşan bir sistemde, her bir cisme diğer cisimlerini ve bu etkilerini ve bunların sonucunda oluşan hareketleri hesaplamaktadır. Bu etki elektrostatik kuvvet veya kütle çekim kuvveti gibi kurallardır. Plazma fiziği, bilgisayar çizgeleri (grafları), akışkanlar mekaniği, astrofizik gibi alanda n – cisim problemi kullanılmaktadır. Fiziksel sistemlerden başka diferansiyel denklem ve sayısal analizde çözüm yöntemi olarak uygulanmaktadır.

Dört veya daha fazla cisim bulunan ve ters kare kuralına uyan n – cisim problemi için bilinen başka bir analitik çözüm yoktur. Bu da yaklaşımı popüler kılmaktadır. Benzetim genelde küçük zaman aralıklarında çalıştırılır ki istenilen zamanda cisimlerin konumları bulunabilsin. Her adımda cisimlerin birbirlerine olan etkileri hesaplanır. Toplam n tane cisimden oluşan bir sistemde n(n-1) tane hesaplama yapılmalıdır. Genellikle fiziksel sistem simülasyonlarında çok sayıda cisim bulunur. Bu nedenle n sayısı büyüdükçe gereken işlem sayısı hızla büyüyüp algoritma çok zaman alacaktır (İnner, B., Sevilgen, F.E., 2013).

Sistem simülasyonu her bir zaman aralığı için kuvvetleri hesapladıktan sonra kuvvetlerin değerleriyle hız ve konum hesaplanır. Cisimler yeni konumlarına yerleştirilir ve sonraki zaman dilimi için aynı işlemler tekrarlanır. N – cisim algoritmalarında kuvvetlerin hesaplanması haricindeki bütün adımlar aynıdır.

Şekil 6.11: Cisim test sonuçları

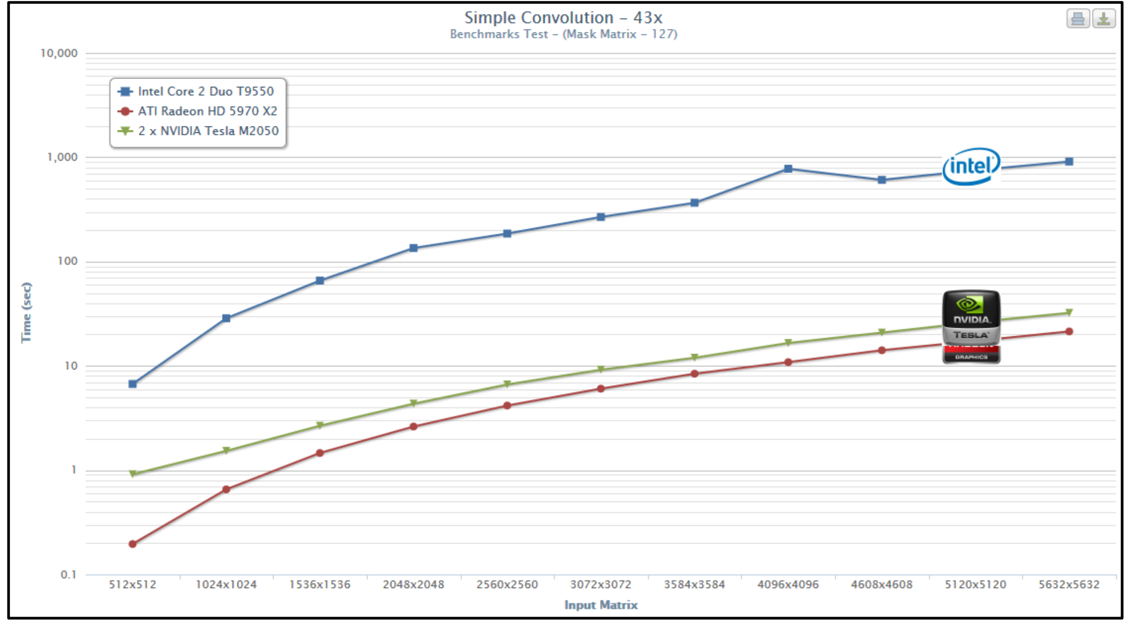


Kaynak: Bu şekil Ersin Kuzu tarafından hazırlanmıştır.

6.9 KONVOLÜSYON İŞLEMİ (SIMPLE CONVOLUTINO)

Konvolüsyon bir sinyal işleme kavramıdır. Sinyaller üzerinde gerçekleştirilen işlemlerle anlamlı veriler elde edilmesine sinyal işleme denir. Konvolüsyon iki işlevli bir matematik işlemini temsil eder ve üçüncü bir fonksiyon ile sonuçlanır. Fonksiyonel analiz, elektrik mühendisliği, olasılık, görüntü ve sinyal işleme uygulamaları iki boyutlu konvolüsyon teknikleri kullanan alanlardır (Electronic, 2014).

Şekil 6.12: Konvolüsyon İşlemi test sonuçları



Kaynak: Bu şekil Ersin Kuzu tarafından hazırlanmıştır.

6.10 AES ŞİFRELEME ALGORTİMASI

Modern kriptoloji genel olarak simetrik ve asimetrik kriptoloji olarak ikiye ayrılmaktadır (Özkat, A.S., 2014). Asimetrik şifreleme açık anahtar ilkesine dayanmaktadır. Gizlenmek istenen metin herkesin bildiği bir anahtarla şifrelenip sadece gizli anahtar ile çözülebilir. Simetrik algoritmalarda ise tek bir gizli anahtar bulunur; şifreleme ve şifre çözme için bu anahtar kullanılır. Simetrik şifreleme blok ve dizi şifreleme olarak iki ana başlıkta toplanır. Gelişmiş şifreleme standardı (AES – Advanced Encryption Standard) bir blok şifreleme türüdür.

Algoritmanın her turunda 4 farklı alt işlem gerçekleştirilir. Bunlar sırası ile bayt değiştirme, satır kaydırma, sütun karıştırma ve tur anahtarı ile toplamıdır. 10 tur sonrasında giren veri şifrelenmiş olarak dışarı çıkmaktadır. İlk tura anahtar ilk haliyle katılmakta diğer turlarda yeni üretilen anahtarlar sokulmaktadır.

Algoritmanın Çalışma Yapısı (Özgenç, B., 2014)

a. Aes orijinal veriyi 128 bitlik bloklara bölüyor ve her bloğu 4x4'lük bir matris halinde düzleniyor. Böylece 16 matris hücresi, iki basamaklı onaltılık bir sayıya denk düşen 8 bit ya da 1 byte içeriyor.

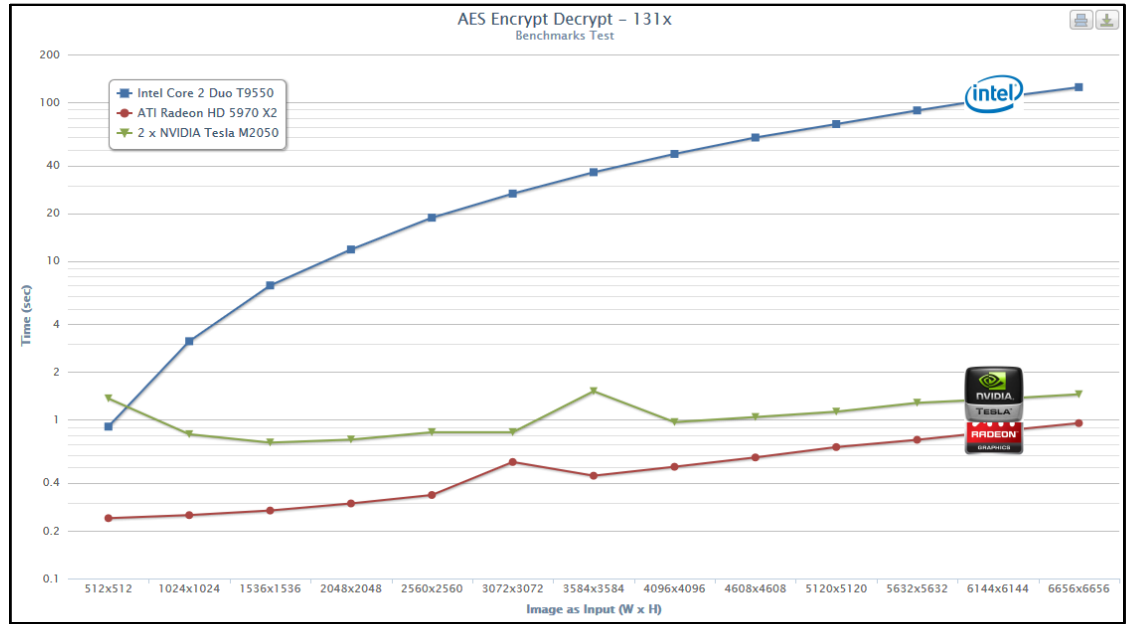
b. Anahtar geliştirime; bir algoritma 128 bitlik aes anahtarından her biri 128 bit uzunluğunda 11 anahtar oluşturuyor.

c. Başlangıç; aes veri bloğu bit bit (XOR işlemi) ilk anahtar ilişkilendiriliyor.

d. Alt byte'lar; bloğun her bir byte'ı referans tablosundan bir byte'la değiştirilir. Eski byte'ın iki onaltılık rakamı, yeni byte'ın satır ve sütun sayısı oluyor.

e. Sütun kaydırma; bloğun sütunları 0, 1, 2 ya da 3 konum sola kaydırılıyor. Taşan sütunlar sağa ekleniyor.
f. Sütun karıştırma; aes bloğun sütunlarından ve sabit bir matrisin satırlarından hareketle yeni hücre değerleri hesaplanıyor. Bu işlem sadece son geçişte yapılıyor.
g. Anahtar ekleme; uygun anahtar kullanılarak bloktaki veriler bit bit değiştiriliyor.
h. Son olarak aes şifreli blokları bir araya getirerek tek bir metin oluşturuluyor. Bu kodun çözümü için aynı işlemleri tersine sırasıyla yapmak gerekiyor.

Şekil 6.13: AES Şifreleme test sonuçları



Kaynak: Bu şekil Ersin Kuzu tarafından hazırlanmıştır.

6.11 MD5 ALGORİTMASI

Message Digest 5 (MD5) şifreleme algoritmaları arasında en çok karşılaşılan algoritmalardandır. Md5 geliştiricisi Ron Rivest tarafından 1992'de Md5'in önceki sürümü Md4'deki zayıflıklar giderilerek oluşturulmuştur. Bir yazının Md5 ile şifrelenmiş hali her zaman 32 karakter ve 128 bittir. Yani 5 harflik bir veri şifrelense de 20 harflik bir veri şifrelense de çıktı her zaman 32 karakterden oluşacaktır.

Md5 tek yönlü bir algoritmadır. Yani md5'de çift yönlü algoritmalar gibi hem şifreleme hem şifre çözme olanağı yoktur. Sadece şifreleme yapılabilir. Bu md5'in daha güçlü ve güvenli olmasını sağlayan bir özelliktir. Forumlarda şifreyi unuttum seçeneği seçildiğinde kullanıcıya şifreyi göndermek yerine şifreyi sıfırlayıp kullanıcıdan yeni şifre girmesini istemelerinin nedeni budur. Forumlarda ilk kayıt olunduğunda girilen şifre md5 ile şifrelenip veritabanına kaydedilir. Daha sonraki girişlerde her defasında girilen şifrenin md5 hali ile veritabanındaki asıl şifrenin md5^li hali karşılaştırılır. Veritabanındaki veriler veritabanına yapılacak herhangi bir saldırıda gerçek değerleri bulunamaması için md5'le şifrelenmiş halde tutulur. Bu da

kullanıcıya şifreleri gönderememelerine sebep olur. Çünkü sistem md5'li veriyi deşifre edemez. Md5 her ne kadar güçlü bir algoritma olsa da çözülemez değildir. Çeşitli siteler ya da programlarla çok uzun ve karışık değilse şifreler çözülebilir. Md5 ile şifrelenmiş yazıları kırmanın birkaç yolu vardır.

Brute Force yöntemi sistemlerin veri tabanlarında MD5 ile şifrelenmiş olarak saklanan şifreleri belirli tahminler yürüterek önceden hazırlanmış hazır karakter setli algoritmalarla bulmaya çalışan bir saldırıdır. Bu yöntemle 10 karakterli bir şifrede büyük – küçük harf rakam ve özel karakterlerden oluşması birleşen sayısını çok fazla arttıracağından Brute Force'nin şifreyi çözme süresi uzun sürecektir.
Yeni geliştirilmiş sistemlerde, olası sistem açıkları kullanılarak şifrelerin ele geçirilmesinde şifrelerin çözülebilmesini en aza indirgeyebilmek için şifreleri şifreledikten sonra şifrelenmiş hallerini tekrar şifrelemektedirler.

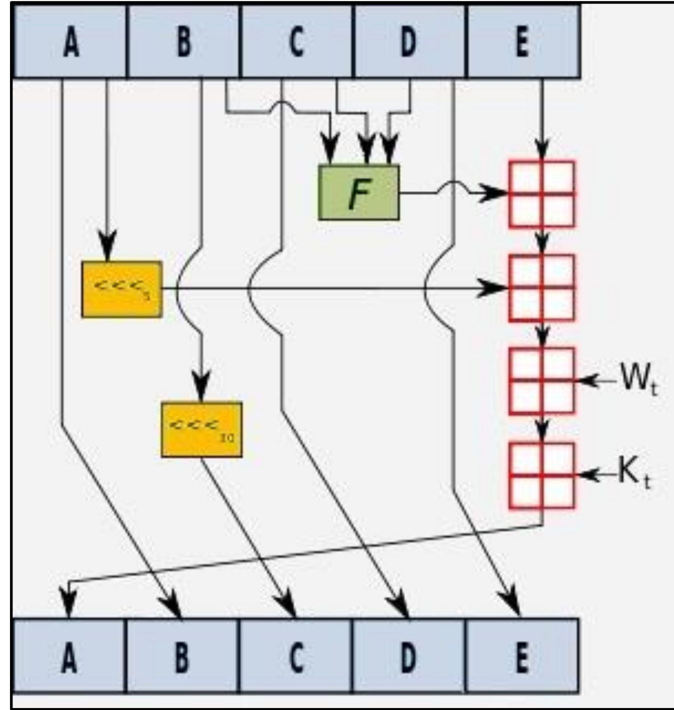
Rainbow büyük – küçük harf, rakam, ve özel karakterlerin kendi içlerinde oluşturabilecekleri olasılıklar düşünülerek 1 karakterden başlayıp sonsun karaktere kadar tüm şifrelerin MD5 ile şifrelenmiş hallerinin bir tabloda biriktirilmesidir. MD5 ile şifrelenmiş hali bilinen bir şifrelenmiş şifrenin tespiti saniyeler sürmektedir. Çok fazla olasılık olduğundan bu yöntem de uzun sürecektir.

6.12 SHA1 ALGORİTMASI

Secure Hashing Algorithm olarak adlandırılan, şifreleme algoritmaları içerisinde en yaygın olarak kullanılan algoritma olduğu kabul gören SHA1, United States National Security Agency tarafından tasarlanmıştır. "Hash" fonksiyonlarına dayalı veritabanı yönetimine (database management) imkan sağlar (İTÜBİDB, 2013). Özellikleri;

a. SHA1 algoritması ile sadece şifreleme işlemi yapılır, şifre çözümleme işlemi yapılamaz.
b. Diğer SHA algoritmaları içerisinde en yaygın olarak kullanılan SHA1 algoritmasıdır.
c. SHA1 algoritması ile 160 bitlik özetler oluşturulur. MD5 ve SHA1 arasındaki temel fark oluşturdukları özetlerdeki boyut farkıdır.
d. SHA1 algoritması, e-posta şifreleme uygulamaları, güvenli uzaktan erişim uygulamaları, özel bilgisayar ağları ve daha birçok alanda kullanılabilir.
e. Günümüzde güvenliği arttırmak amacıyla SHA1 ve MD5 algoritmaları birbiri ardına kullanılarak veriler şifrelenir.

Şekil 6.14: SHA1 şifreleme yapısı



Kaynak: (İTÜBİDB, 2013)

Çakışmalar (Collisions)

Şifreleme algoritmalarında karşı karşıya gelinen problemlerden biri Çakışma (Collision) problemidir. Çakışma, iki farklı verinin şifreleme işlemi sonrasında aynı özete sahip olması olarak tanımlanabilir. SHA1 ile birlikte en yaygın kullanılan şifreleme algoritması olarak tanımlayabileceğimiz MD5 algoritması ile 128 bitlik özetler oluşturulur. Bu durumda, 264 deneme yapmak çakışma bulmak için yeterli olacaktır. SHA1 algoritmasında bu durum 160 bitlik özetler oluşturularak ve çakışmaya karşı güvenlik derecesi 280 denemeye çıkarılarak daha güvenli hale getirilmiştir.

6.13 NTLM ALGORİTMASI

NTLM (New Technology LAN Manager) Microsoft Windows ortamlarında en sık kullanılan yöntemlerden biridir. Bu yöntemde kullanıcı sisteme girmek istediğinde kullanıcıya kimlik bilgileri sorulur. Diğer yöntemlerden farklı olarak kullanıcı parolası bu yöntemde ağ (network) ortamında dolaşmaz. Parola bilgisi one – way hash algoritması ile hash'lenerek sunucuya yollanır. İstenci ve sunucu Challenge/Response protokolü ile ortamda haberleşirler (Yüksektepeli, O., 2011).

a. Kullanıcı istemci bilgisayarda domain, kullanıcı adı ve şifre bilgilerini sağlar. İstemci bilgisayar kullanıcısı parolasının hash'ini çıkarır ve orijinal şifreyi yok eder.

b. İstenci kullanıcı adını Palin – Text olarak kullanıcıya gönderir.
c. Sunucu 16 byte'lık rastgele bir sayı üretir (Challenge) ve istemci bilgisayara gönderir.
d. İstemci bilgisayar kullanıcının şifresinin hash'i ile challenge'yi encrypt eder. Buna Response denir.

Kimlik doğrulaması yapılacak olan sunucuda (Domain Controller) aşağıdaki üç bilgi olur;

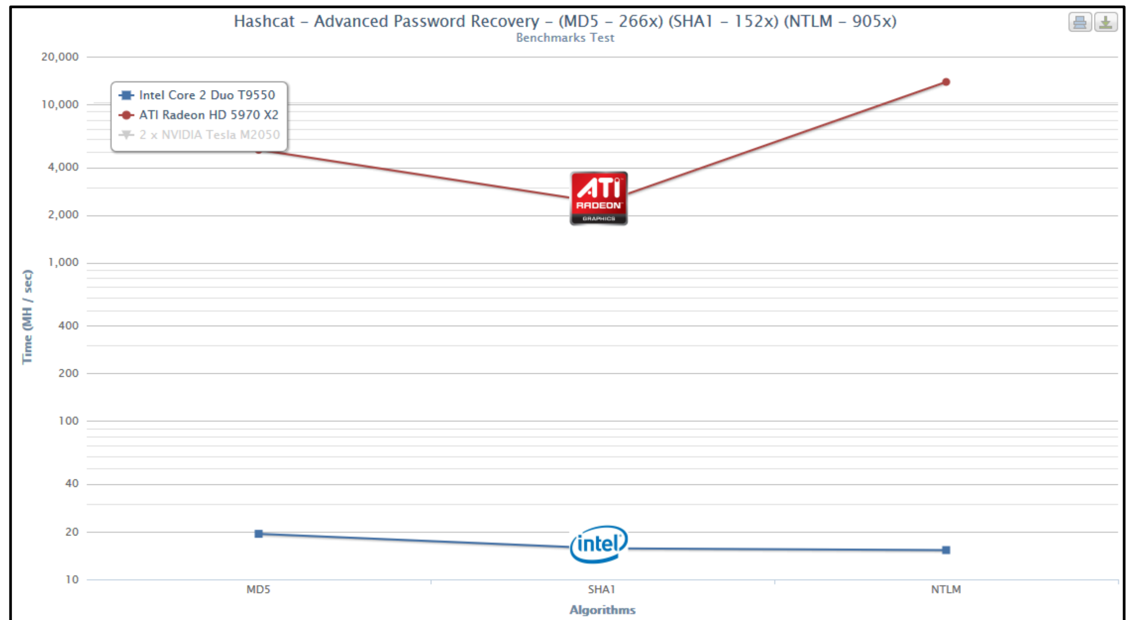
a. Kullanıcı adı
b. İstemci makinaya gönderilen Challenge
c. İstemci makinadan alınan Response

Bu bilgiler kullanıcının Authentication yapmak istediği sunucu tarafından Domain controller'a gönderilir.

a. Domain controller kullanıcı adını Security Account Manager (SAM) veritabanında bulunan şifrenin hash'ini almak için kullanılır. Bu şifre hash'i istemci tarafından challenge'i encrypt etmek için kullanılır.
b. Bu encrypted challenge istemci tarafında oluşturulan response ile aynı ise, kullanıcı Active Directory tarafından Authenticate edilir.

NTLM konfigürasyon kolaylığı ve kolay kullanım avantajlarından dolayı en fazla tercih edilen kimlik doğrulama yöntemlerindendir.

Şekil 6.15: MD5, SHA1, NTLM test sonuçları



Kaynak: Bu şekil Ersin Kuzu tarafından hazırlanmıştır.

7. SONUÇ

Bu araştırmada OpenCL platformunun sunduğu GPU, CPU desteği ve paralel programlama ile yüksek başarımlı hesaplamalar gerçekleştirme imkanı sağlayan OpenCL dili incelenmiştir. Bu dil ile bazı hesaplama ağırlıklı algoritmaların uygulaması gerçekleştirilmiştir.

OpenCL; GPU, CPU, FPGA ve diğer işlem birimlerinden oluşan heterojen ortamlarda GPGPU programlarını çalıştırmaya yarayan bir çatıdır. Sunmuş olduğu API ile platform, işletim sistemi ve üretici bağımsızlığı sağlamaktadır. OpenCL standartlarının kabulünün gün geçtikçe artması ve önde gelen üreticilerin katkısıyla OpenCL'in kullanımı giderek artmaktadır.

GPU'nun genel amaçlı hesaplamalar için kullanılması fikrinin üzerinden 10 yıl geçmesine rağmen, araştırmacı ve yazılım geliştiricilerin uygulamalarını GPU'lar üzerinde kolaylıkla geliştirebilecekleri ortam ve araçların geç sunulmuş olması bu teknolojinin (GPGPU) kavranmasını, yaygınlaşmasını yavaşlatan bir neden olmuştur. Dünyanın en büyük profesyonel sosyal iletişim ağı olan Linkedln'de Java için yapılan kişi aramaları OpenCL'e göre 520 kat daha fazla sonuç çıkmıştır.

OpenCL ile geliştirilebilecek uygulamaların OpenCL platformuna bütünleşik kullanıcı dostu bir geliştirme ortamının olmaması geliştirme sürelerini uzatmakta ve hata ayıklama işlemlerini zorlaştırmaktadır. Bu nedenle tasarımdan kaynaklanan hataların yakalanması, kod üzerindeki hatalı bölümün bulunması çok zaman ve emek kaybına neden olmaktadır.

Kernel fonksiyonlarının en etkili şekilde çalışmasını sağlayacak indeks alanı, iş grupları ve iş öğeleri sayılarını belirleme yazılımcı tarafından gerçekleştirilecek olması performans açısından sorun teşkil etmektedir. Kernellerin verimli çalışması için bu sayıların dikkatli belirlenmesi gereklidir.

Yapılan denemelerde CPU ortamında kötü performans sergileyen algoritmaların veri paralelliği ilkesine uyumlu olmaları halinde GPU üzerinde CPU eşleniklerine göre daha verimli oldukları saptanmıştır.

Yapılan testler, uygulamalar ve araştırmalar sonucu GPU'ların hesaplama yeteneklerinin özellikle veri paralelliği gerektiren durumlarda çok başarılı sonuçlar göstermiştir. Mimarilerine uygun algoritmalarda yüksek başarımlı hesaplama alanındaki bilgisayar kümelerinin yerini almaktadır.

Algoritmalar hesaplanırken 2 çekirdekli 1 GPU ve 1 çekirdekli 2 GPU kullanılmıştır. Çoklu GPU teknolojisi ile aynı sistem üzerindeki birden fazla grafik kartında aynı uygulamanın çalıştırılması mümkündür. GPU kümeleme teknolojisi ile farklı sistemler

üzerindeki bir ya da birden fazla grafik kartında aynı uygulamanın çalıştırılması mümkündür.

Kullanılan algoritmalar ATI GPU (Ati Radeon HD 5970 X2) üzerinde farklı problem aralıklarında minimum 13 maksimum 1001 kata kadar hızlanma elde edilmiştir. CPU'ya göre 1190 kat performans artışı, 119 kat enerji tasarrufu, 618 kat daha düşük maliyet farkı elde edilmiştir.

GPU mimarisinin paralel işlemeyi sağlaması nedeniyle iş parçacıkları ile hesaplama süreleri arasında yaklaşık doğrusal bir ilişki bulunmamaktadır.

OpenCL ile yapılan testlerde ATI GPU (Ati Radeon 5970 X2) ile eş değeri NVIDIA GPU (2 x Nvidia Tesla M2050) arasında çok fazla bir performans farkı görülmemektedir. Teorik performans değerlerine göre ATI GPU 2.25 kat daha hızlı olarak gösterilmektedir. Testlerde bu değer 2.55 kat olarak elde edilmiştir. Hesaplama sürelerine göre ise ATI GPU 3.14 kat hızlı, 4.77 kat enerji tasarruflu, 19.30 kat düşük maliyetli değerler elde edilmiştir.

KAYNAKÇA

Kitaplar

Kilgariff, E., Fernando, R. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison-Wesley, 2005. ftp://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch30.pdf

Scarpino, M. *OpenCL in action how to accelerate graphics and computation.* Shelter Island. Manning, November 2011. http://it-ebooks.info/book/1620/

Wilkinson, B., Allen, M. (2004, Mart). *Parallel Programming: Techniques and Application Using Networked Workstations and Parallel Computers.* 2. Baskı. Charlotte: Pearson Prentice Hall

Süreli Yayınlar

Kaufman, A., Fan, Z., Petkov, K. *Implementing the lattice Boltzmann model on commodity graphics hardware.* J. Stat. Mech.: Theory Exp. 2009 (2009), no. 06, P06016. http://iopscience.iop.org/1742-5468/2009/06/P06016/pdf/1742-5468_2009_06_P06016.pdf

Diğer Yayınlar

Ahmed, M., F. (2010, Mayıs). *OpenCL.* http://mohamedfahmed.wordpress.com/2010/05/20/opencl-2/

AMD Staff. (2011, Haziran). *OpenCL™ and AMD APP SDK v2.4 | AMD.* http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-and-the-amd-app-sdk-v2-4/

AMD's Close-to-the-Metal. (2014, Şubat). *AMD's Close-to-the-Metal.* http://amdctm.sourceforge.net/

Ankara Üniversitesi. (2013, Kasım). *FZM421 Fizikte Bilgisayar Uygulamaları I (5Kasım2013) – Özdeğerler ve Özvektörler.* http://phys.eng.ankara.edu.tr/files/2013/11/Ozdeger.pdf

ATI. (2010, Mayıs). *Introduction to OpenCL™ Programming Training Guide.* http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/01/Introduction_to_OpenCL_Programming-Training_Guide-201005.pdf

Blaise Barney, Lawrence Livermore National Laboratory (LLNL). (2014, Şubat). *Introduction to Parallel Computing.* https://computing.llnl.gov/tutorials/parallel_comp/

BrookGPU. (2014, Şubat). *BrookGPU.* http://graphics.stanford.edu/projects/brookgpu/

C++ AMP (C++ Accelerated Massive Parallelism). (2014, Şubat). *C++ AMP (C++ Accelerated Massive Parallelism).* http://msdn.microsoft.com/tr-tr/library/hh265137.aspx

Compute Shader Overview. (2014, Şubat). *Compute Shader Overview.* http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331%28v=vs.85%29.aspx

CUDA (Compute Unified Device Architecture). (2014, Şubat). *CUDA Zone.* https://developer.nvidia.com/cuda-zone

Ebersole, M. (2012, Eylül). *What is CUDA? | NVIDIA Blog.* http://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/

Electronic. (2014, Nisan). *İki Boyutlu Konvolusyon Nedir.* http://elektronik.365electric.com/satellite-radio/satellite-radio-receivers/1017039260.html

GridMiner. (2014, Şubat). *Intelligent Grid Solutions.* http://www.gridminer.org/

IBM developerWorks. (2014, Şubat). *Cloud-based education, Part 2: High – performance computing for education.* http://www.ibm.com/developerworks/cloud/library/cl-ind-cloud-e-learning2/

İnner, B., Sevilgen, F.E. (2013, Şubat). *N-cisim algoritmalarının performans karşılaştırması.* Kocaeli Üniversitesi. http://akademikpersonel.kocaeli.edu.tr/binner/bildiri/binner13.02.2013_18.34.42bildiri.pdf

İTÜBİDB. (2013, Eylül). *SHA1 Şifreleme Metodu.* http://bidb.itu.edu.tr/seyirdefteri/blog/2013/09/08/sha1-%C5%9Fifreleme-metodu

Lib Sh – Embedded Metaprogramming Language. (2014, Şubat). *Lib Sh – Embedded Metaprogramming Language*. http://libsh.org/

Lighthouse3D. (2014, Şubat). *GLSL Core Tutuorial – Pipeline (OpenGL 3.2 – OpenGL 4.2).* http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/pipeline33/

NVIDIA Developer. (2014, Şubat). *Cg Toolkit.* https://developer.nvidia.com/cg-toolkit

NVIDIA. (2014, Şubat). *Programming Guide :: CUDA Toolkit Documentation.* http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Nvidia. (2014, Şubat). *GPU Hesaplama Nedir? | Yüksek Performanslı Hesaplama | NVIDIA.* http://www.nvidia.com.tr/object/gpu-computing-tr.html

Onurlu, S. (2003, Mayıs). *Fraktal Dünya | ODTÜ'DEN.* http://www.odtumd.org.tr/bulten/119/fraktal.htm

OpenCL. (2014, Şubat). *The open standard for parallel programming of heterogeneous systems.* https://www.khronos.org/opencl/

OpenCL™ (Open Computing Language) Zone. (2014, Şubat). *OpenCL™ Zone | AMD.* http://developer.amd.com/tools-and-sdks/opencl-zone/

Özgenç, B. (2014, Nisan). *btriskblog Bilgi Güvenliği Metodlerı ve Paylaşımları: Kriptografi – 1.Bölüm.* http://blog.btrisk.com/2014/04/kriptografi-1bolum.html

Özkan, H., (2013, Ocak). *K-means kümeleme ve k-nn sınıflandırma algoritmalarının öğrenci notları ve hastalık verilerine uygulanması.* Bitirme Ödevi. İstanbul Teknik Üniversitesi. http://akademi.itu.edu.tr/kiris/DosyaGetir/92593/09%20Ozkan%202013.pdf

Özkat, A.S. (2014, Nisan). *Kriptografi Algoritmaları ve Kullanımları.* http://e-bergi.com/y/Kriptografi-Algoritmalari-ve-Kullanimlari-

Rudolph, J. (2012, Şubat). *OpenCL Work Item Ids: Global/Group/Local | Johannes Rudolph's Blog.* http://jorudolph.wordpress.com/2012/02/03/opencl-work-item-ids-globalgrouplocal/

Simplilearn. (2013, Şubat). *Binomial Option Pricing Model | AnalystForum.* http://www.analystforum.com/article/frm/binomial-option-pricing-model

Şeker, Ş.E. (2009, Mayıs). *Floyd-Warshall Algoritması – Bilgisayar Kavramları.* http://bilgisayarkavramlari.sadievrenseker.com/2009/05/29/floyd-warshall-algoritmasi/

Tatarchuk, N. 2007. *Topics in Tessellation for High Quality Real-Time Rendering in Computer Graphics. Eurographics Game Industry Presentations Track: New Technologies and Solutions session.* Eurographics, Prague, Czech Republic, September 2007. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Tatarchuk-Tessellation%28EG2007%29.pdf

TR-Grid. (2014, Şubat). *TR-Grid Wiki Duyuruları.* http://wiki.grid.org.tr/index.php/Ana_sayfa

Wikipedia, the free encyclopedia. (2014, Şubat). *Computer cluster.* http://en.wikipedia.org/wiki/Cluster_%28computing%29

Wikipedia, the free encyclopedia. (2014, Şubat). *Distributed computing.* http://en.wikipedia.org/wiki/Distributed_computing

Wikipedia, the free encyclopedia. (2014, Şubat). *Flynn's taxonomy.* http://en.wikipedia.org/wiki/Flynn%27s_taxonomy

Wikipedia, the free encyclopedia. (2014, Şubat). *SETI@home.* http://en.wikipedia.org/wiki/SETI@home
Yan, K. (2014, Şubat). *Introduction to parallel computing*. http://kuangshi-yan.net/introduction-to-parallel-computing/
Yüksektepeli, O. (2011, Ağustos). *Sharepoint 2010 Kimlik Doğrulama Yöntemleri.* http://onuryuksektepeli.com/sharepoint-2010-kimlik-dogrulama-yontemleri/
Yünel, Y. (2010, Mayıs). *K-means kümeleme algoritmasının genetik algoritma kullanılarak geliştirilmesi.* Bitirme Ödevi. İstanbul Teknik Üniversitesi. http://akademi.itu.edu.tr/kiris/DosyaGetir/92587/03%20Yunel%202010.pdf

ÖZGEÇMİŞ

Adı Soyadı:	Ersin Kuzu	
Doğum Yeri ve Yılı:	Daday, 1982	
Yabancı Dil:	İngilizce	
Lisans:	Süleyman Demirel Üniversitesi, 2006	
Yüksek Lisans:	Bahçeşehir Üniversitesi, 2014	
Enstitü Adı:	Fen Bilimleri Enstitüsü	
Program Adı:	Bilgi Teknolojileri	
Çalışma Hayatı:	Galata Mesleki ve Teknik Anadolu Lisesi	2014
	Sepetçioğlu Mesleki ve Teknik Anadolu Lisesi	2011
	Melahat Öztoprak İlköğretim Okulu	2007
	Kastamonu Ticaret Meslek Lisesi	2006