

**ÇİFT YÖNLÜ KARAYOLU TRAFİĞİNİN HÜCRESEL OTOMAT TABANLI
2B BENZETİMİ VE CUDA MİMARİSİ İLE HIZLANDIRILMASI**

2015

**LİSANS BİTİRME PROJESİ
BİLGİSAYAR MÜHENDİSLİĞİ**

NEZİHE SÖZEN

TC

KARABÜK ÜNİVERSİTESİ

MÜHENDİSLİK FAKÜLTESİ

Lisans Bitirme Projesi

**ÇİFT YÖNLÜ KARAYOLU TRAFİĞİNİN HÜCRESEL OTOMAT TABANLI
2B BENZETİMİ VE CUDA MİMARİSİ İLE HIZLANDIRILMASI**

NEZİHE SÖZEN

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

LİSANS BİTİRME PROJESİ

KARABÜK

2015

Nezihe SÖZEN tarafından hazırlanan "ÇİFT YÖNLÜ KARAYOLU TRAFİĞİNİN HÜCRESEL OTOMAT TABANLI 2B BENZETİMİ VE CUDA MİMARİSİ İLE HIZLANDIRILMASI" başlıklı bu projenin Lisans bitirme projesi olarak uygun olduğunu onaylıyorum.

Yrd.Doç.Dr. İsmail KURNAZ

.....

Tez Danışmanı, Bilgisayar Mühendisliği Anabilim Dalı

"Bu projedeki tüm bilgilerin akademik kurallara ve etik ilkelere uygun olarak elde edildiğini ve sunulduğunu; ayrıca bu kuralların ve ilkelerin gerektirdiği şekilde, bu çalışmadan kaynaklanmayan bütün atıfları yaptığımı beyan ederim."

Nezihe SÖZEN

ÖZET

ÇİFT YÖNLÜ KARAYOLU TRAFİĞİNİN HÜCRESEL OTOMAT TABANLI 2B BENZETİMİ VE CUDA MİMARİSİ İLE HIZLANDIRILMASI

Lisans Bitirme Projesi

NEZİHE SÖZEN

Karabük Üniversitesi

Mühendislik Fakültesi

Bilgisayar Mühendisliği Anabilim Dalı

Tez Danışmanı

Yrd. Doç. Dr. İsmail KURNAZ

Bu çalışmada; konum bilgileri ve kategorileri bilinen araçların yer aldığı bir trafik akışının bilgisayar grafikleri kullanılarak Merkezi İşlem Birimi(MİB) ve Grafik İşlem Birimi(GİB) üzerinde benzetimi gerçekleştirilerek performans karşılaştırılması yapılmıştır. GİB üzerinde hesaplamalar yapılırken CUDA teknolojisi kullanılmış, Fermi mimarisi üzerinde uygulama geliştirilmiştir. Hücresel otomatlardan faydalanan ve NaSch Modeli temel alınarak kodlamalar yapılmıştır. Her bir hücre ya dolu ya da boş olacak şekilde oluşturulmuştur. Şeritler kapalı bir sistem modelinde tasarlanmıştır, bir döngüye sahiptir. Bu modele göre aracın haritadan çıkış yapması durumunda tekrar ilk başladığı noktaya geri dönmesi sağlanmaktadır. Bunun sebebi hücresel otomatın paralel bir güncellemeye ihtiyaç duyuyor olmasıdır.

Uygulama MS Visual Studio 2012 editörü üzerinde geliştirilmiş, bilgisayar grafikleriyle görselleştirilmesi amacıyla OpenGL kütüphanesinden yararlanılmıştır.

Anahtar Sözcükler: karayolu trafik benzetimi, hücresel otomatlar, CUDA, GPGPU, OpenGL

ABSTRACT

2D SIMULATION OF TWO WAY ROAD TRAFFIC BASED ON CELLULAR AUTOMATA AND ACCELERATING WITH CUDA ARCHITECTURE

BSc. Senior Project

Nezihe SÖZEN

Karabük University

Faculty of Engineering

Computer Engineering Department

Senior Project Advisor:

Assist. Prof. İsmail KURNAZ

In this study; using a computer graphics of a traffic flow which includes location infos and vehicles that their categories are known, a performance benchmark has been done by a Central Processor Unit (CPU) and Graphics Processing Unit simulation. CUDA technology has been used for the GPU calculations and Fermi architecture has been used for application development. Cellular automaton model has been used and based on NaSch Model, implementations have been done. Every cell is either empty or full. Lanes have been designed in a closed system model and include loops. According to this model if a vehicle leaves the map, vehicle will be returned to its starting point because cellular automaton needs a parallel update.

Application has been developed on the MS Visual Studio 2012 editor. To visualize with computer graphics has been benefited from the OpenGL library.

Key Words : highway traffic simulation, cellular automaton, CUDA, GPGPU, OpenGL

TEŞEKKÜR

İÇİNDEKİLER

BÖLÜM 1.....	1
GİRİŞ.....	1
1.1 PROJENİN AMACI.....	1
1.2 PROJENİN YAPISI.....	2
BÖLÜM 2.....	3
TRAFİK BENZETİM SİSTEMLERİ.....	3
2.1 TRAFİK AKIŞI MODELLEME YÖNTEMLERİ.....	3
2.1.1 Uzman Sistemler.....	3
2.1.2 Bulanık Mantık.....	5
2.1.3 Sonlu Durum Makineleri (Finite State Machines).....	7
2.1.4 Etmen Tabanlı Modelleme.....	9
2.1.5 Hücresel Otomatlar(Cellular Automaton).....	13
BÖLÜM 3.....	23
GPGPU (GENERAL PURPOSE COMPUTING ON GRAPHICAL PROCESSING UNIT) MODELİ.....	23
3.1 GPGPU PLATFORMLARI.....	23
3.1.1 DirectCompute.....	23
3.1.2 CUDA.....	23
3.1.3 OpenCL.....	24

3.2	GRAFİK KARTLARI VE GRAFİK İŞLEM BİRİMİ (GİB).....	25
3.3	SERİ HESAPLAMA.....	27
3.4	PARALEL HESAPLAMA.....	28
3.4.1	Paralel İşlem Örneği.....	28
3.5	HETEROJEN HESAPLAMA.....	29

BÖLÜM 4.....	30
CUDA.....	30
4.1 İŞ AKIŞI.....	30
4.2 ÖLÇEKLENİLEBİLİR PROGRAMLAMA.....	32
4.3 CUDA PROGRAMLAMA MİMARİSİ.....	32
4.3.1 Kernel.....	32
4.3.2 Thread.....	33
4.3.3 Block.....	33
4.3.4 Grid.....	34
4.4 CUDA HAFIZA ORGANİZASYONU.....	35
4.4.1 Yerel Bellek (Local Memory).....	35
4.4.2 Paylaşımılı Bellek (Shared Memory).....	36
4.4.3 Genel Bellek (Global Memory).....	36
4.4.4 Sabit Bellek (Constant Memory).....	37
4.4.5 Doku Bellek (Texture Memory).....	39
4.5 CUDA PROGRAM YÜRÜTME İŞLEMLERİ.....	40
4.6 BELLEK ERİŞİM FONKSİYON NİTELEYİCİLERİ.....	42
4.6.1 __global__.....	42
4.6.2 __device__.....	43
4.6.3 __host__.....	43
4.7 DEĞİŞKEN NİTELEYİCİLERİ.....	43
4.7.1 __device__.....	43
4.7.2 __constant__.....	43
4.7.3 __shared__.....	43
4.8 BELLEK YÖNETİMİ.....	44
4.8.1 Hafıza Ayırma Fonksiyonları.....	44
4.8.2 Veri Kopyalama Fonksiyonu.....	44
4.9 HESAPLAMA KAPASİTESİ.....	45

4.9.1	Fermi Mimarisi[50].....	47
4.9.2	Kepler Mimarisi[51].....	48
4.9.3	Fermi ile Kepler Mimarilerinin Teknik Karşılaştırması.....	50
4.9.4	Warp Yapısı.....	50
4.10	PERFORMANS ÖLÇÜMÜ.....	51
4.10.1	Teorik Bantgenişliği.....	51
4.10.2	Etkin Bantgenişliği.....	51
4.10.3	CUDA Event API.....	52
4.11	CUDA-OPENGL İŞBİRLİĞİ.....	52
4.11.1	OpenGL.....	52
4.11.2	CUDA ile OpenGL Etkileşiminin Sağlanması İçin Gerekli İşlemler	53
BÖLÜM 5.....		56
ÇİFT YÖNLÜ KARAYOLU TRAFİĞİNİN HÜCRESEL OTOMAT TABANLI 2B BENZETİMİ VE CUDA MİMARİSİ İLE HIZLANDIRILMASI.....		56
5.1	MATERIALLER.....	56
5.1.1	CUDA Destekli Ekran Kartı(NVIDIA GeForce GT 520M).....	56
5.1.2	Microsoft Visual Studio 2012 Ultimate.....	57
5.1.3	CUDA Toolkit V7.0.....	57
5.2	YÖNTEM.....	57
5.2.1	Özet.....	57
5.2.2	Benzetim Verileri.....	58
5.2.3	Benzetimin MİB Tarafı.....	61
5.2.4	Benzetimin GİB Tarafı.....	69
BÖLÜM 6.....		72
SONUÇLAR VE ÖNERİLER.....		72

KAYNAKLAR.....	74
EKLER.....	81
CUDA İLE İLGİLİ ÖRNEKLER.....	81
ÖZGEÇMİŞ.....	108

ŞEKİLLER DİZİNİ

Şekil 2-1: Bir Uzman Sistemin Yapısı.....	4
Şekil 2-2:Bulanık Mantık Tabanlı Sistemin Yapısı.....	5
Şekil 2-3: Tek girdili bir sinyalizasyon bulanık sistem örneği.....	6
Şekil 2-4 :Bir Sonlu Durum Makinesinin Geçiş Diyagramı.....	8
Şekil 2-5: Trafik Işıklarının SDM ile Modellenmesi.....	8
Şekil 2-6:Bir SDM'nin Moore Makinesi (Sol) ve Mealy Makinesi(Sağ) Modelinin Durum Geçiş Diyagramı.....	9
Şekil 2-7: FIPA Standardına Göre Etmen Yaşam Döngüsü.....	13
Şekil 2-8: 1B, 2B ve 3B'lu Hücresel Otomat Örnekleri.....	14
Şekil 2-9: Kural-184'e göre hücrelerin durumu.....	14
Şekil 2-10: Kural-184'e göre bir hü cresel otomatın 25 iterasyon işlenmiş hâli	15
Şekil 2-11 :Von Neumann komşuluğunda merkez hücre(P) ve komşuları(D)..	16
Şekil 2-12: Dörtlü komşuluğa göre oluşan 2B bir hü cresel otomat.....	16
Şekil 2-13: Moore Komşuluğunda merkez hücre(C) ve komşuları.....	17
Şekil 2-14 : 5 iterasyon boyunca hü cresel otomatın bulunduğu durumlar.....	17
Şekil 2-15: NaSch Modeline Göre Araçların İlerlemesi Gösteren Örnek (Deterministik).....	18
Şekil 2-16: TCA'da yer alan tek şeritli hü cresel otomat yapısı.....	20
Şekil 3-1: MİB(CPU) ve GİB(GPU) Donanımsal Olarak Karşılaştırılması.....	26
Şekil 3-2: Grafik İşlem Biriminin İş Hattında Veri Dönüşümünü Öرنkleyen Görsel.....	26
Şekil 3-3: Grafik İşlem Birimi İş Hattı Mimarisi.....	27
Şekil 3-4: Seri Hesaplama ile Komut Yürütlmesi.....	27
Şekil 3-5: Paralel Hesaplama Komutlarının Yürütlmesi.....	28
Şekil 3-6: Paralel Hesaplama Yöntemi ile Toplama İşlemi.....	29
Şekil 4-1: CUDA iş akış modeli.....	30
Şekil 4-2: CUDA ile geliştirilen bir programın MİB(Host) ve GİB(Device) üzerinde yürütülmesi.....	31
Şekil 4-3: Çift çekirdekli GİB ile 4 çekirdekli GİB'in Gelen Programa Göre Kendiliğinden Ölçeklenmesi.....	32
Şekil 4-4: Thread Block'ları.....	34
Şekil 4-5: Thread hiyerarşisi.....	34
Şekil 4-6: Thread-Block-Grid İlişkisi.....	35
Şekil 4-7:Her bir thread'in kendine özel Register ve Local Memory alanları mevcuttur	36
Şekil 4-8: Her bir Thread Block'una özel Shared Memory alanı mevcuttur....	36

Şekil 4-9:Tüm block'ların kullanabildiği memory alanı Global Memory'dir...	37
Şekil 4-10:CUDA bellek modeli.....	37
Şekil 4-11:Spatial Locality mantığına göre dizilmiş Texture Memory yapısı...	39
Şekil 4-12: CUDA Bellek Modelleri.....	40
Şekil 4-13: Windows ortamında CUDA kodlarının derlenme ve yürütülmeye aşamaları.....	41
Şekil 4-14:Linux ortamında CUDA kodlarının derlenme ve yürütülmeye aşamaları.....	42
Şekil 4-15: Host ile Device Arasında Veri Aktarımı.....	44
Şekil 4-16: Thread'lerin yazılımsal ve donanımsal yapıları.....	46
Şekil 4-17: Thread'lerin üzerinde çalıştığı SP ve SM yapıları.....	46
Şekil 4-18:Fermi mimarisi genel yapısı.....	48
Şekil 4-19: Bir Fermi SM yapısı.....	48
Şekil 4-20: Kepler Mimarisi Genel Yapısı.....	49
Şekil 4-21: Kepler Mimarısında bir SMX'nin iç yapısı.....	49
Şekil 4-22:Teorik Bantgenişliği Örneği.....	51
Şekil 4-23: OpenGL ile etkileşimli olarak çalışan bir CUDA uygulamasının şematik gösterimi.....	55
Şekil 5-1:NVIDIA GeForce GT 520M ekran kartı aygıt sorgulama ekran çıktısı	56
Şekil 5-2:Hücresel otomat modelinde aracın haritaya giriş-çıkış tipleri.....	57
Şekil 5-3:Modellenen Karayolunun GoogleMaps'ten Alınan Harita Bilgisi.	58
Şekil 5-4:Modellenen Karayolunun GoogleMaps'ten Alınan Yeryüzü Bilgisi	59
Şekil 5-5: Türkiye'de araçların uyması gereken hız sınırları.....	60
Şekil 5-6: Benzetimde modellenen harita.....	62
Şekil 5-7: Hücrelerin görsel olarak ifade edilmesi.....	62
Şekil 5-8: Değişik konumlarda bulunan araçların harita ve hücreler üzerinde yerleşimi.....	65
Şekil 5-9: Kullanıcı Etkileşimi İçin Oluşturulan Menü.....	69
Şekil 6-1: GİB ile MİB Üzerinde Çalıştırılan Fonksiyonların Zaman Verileri.	72

KISALTMALAR DİZİNİ

KISALTMALAR

	<u>AÇIKLAMA</u>
CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
GPU	Graphics Processing Unit
MİB	Merkezi İşlem Birimi
GİB	Grafik İşlem Birimi
SDM	Sonlu Durum Makinesi
JADE	Java Agent DEvelopment Framework
NYP	Neseneye Yönelik Programlama
EYP	Etmene Yönelik Programlama
FIPA	The Foundation for Intelligent Physical Agents
1B	Bir Boyutlu
2B	İki Boyutlu
3B	Üç Boyutlu
CA	Cellular Automata (Hücresel Otomat)
NaSch Modeli	Nagel-Schreckenberg Modeli
GPGPU	General Purpose Computing on Graphical Processing Unit
API	Application Programming Interface
OpenCL	Open Computing Language
BIOS	Basic Input-Output System
SIMD	Single Instruction, Multiple Data
OpenGL	Open Graphics Library
DRAM	Dynamic Random Access Memory

SM	Streaming Multiprocessor
SP	Streaming Processor
SMX	Next Generation Streaming Multiprocessor
ARB	Architectural Review Board
PBO	Pixel Buffer Object
VBO	Vertex Buffer Object
GLUT	OpenGL Utility Toolkit

BÖLÜM 1

GİRİŞ

Karayolu trafik yoğunluğu, en çok büyük şehirlerde etkisini göstermesinin yanısıra gelişimini sürdürden şehirlerde de artık bir sorun hâline gelmiştir. Bilgisayar ortamında hazırlanmış benzetimler ile trafik kurallarına uygun bir akış hem zamandan hem de maddi açıdan kazanç sağlamak üzere test edilebilmektedir.

Bu benzetimler bilgisayar grafiği kullanılarak görselleştirilirler ve gerçekliğe yakınlık elde edilmeye çalışılır.

Trafik sorunlarında en uygun çözüme ulaşılabilmesi için sistemlerin analizi gereklidir. Gerçek dünyada bu sistem analizini gerçekleştirmek neredeyse imkânsızdır. İşte bu noktada trafik akışlarının bilgisayar üzerinde benzetim yoluyla ifade edilmesi hem maaliyeti azaltır hem de deneysel tehlikeleri ortadan kaldırır.

Trafik akışlarını modellemek ve sıkışıklıkları gidermek amacıyla birçok benzetim modeli geliştirilmiştir. TRANSIMS(TRANsportation ANalysis SIMulation System), DRACULA (Dynamic Route Assignment Combining User Learning and microsimulAtion), MATSim (Multi-Agent Transport Simulation), SUMO (Simulation of Urban MObility) bu yazılımlara örnek olarak verilebilir.[1]

Bilgisayarda çizilen bir görüntüyü ekrana iletten birim grafik kartıdır. Grafik kartının işlem birimi olan GPU(Grafik İşlem Birimi) yapısal olarak paraleldir ve bu paralellik kullanılarak işlemlerde hızlanma elde edilmektedir.

1.1 PROJENİN AMACI

Öncelikli amaç, otonom bir şekilde ve kurallara uygun işleyen karayolu trafik akışının modellenmesi ve bu modelin grafik kartının mimarisinden kaynaklı paralellikten yararlanılarak hızlandırılmasıdır. Proje iki aşamada gerçekleştirılmıştır.

Birinci aşamada, geliştirilen uygulama sadece MİB ile birlikte çalışmış, ikinci aşamada CUDA mimarisi kullanılarak hem MİB hem de GİB üzerinde çalıştırılarak birinci aşama ile performans karşılaştırılması yapılmıştır.

Düzen hedefler ise:

- Hücresel otomatların tanıtılması ve örneklerinin incelenmesi
- NVIDIA firmasının CUDA mimarisi ile üretilmiş grafik kartlarında program geliştirme ve incelenmesi.

1.2 PROJENİN YAPISI

Bu çalışmada NVIDIA CUDA C/C++ dilleri kullanılarak, OpenGL etkileşimli bir simülasyon(benzetim) hazırlanmıştır.

Proje çalışması 6 bölümden oluşmaktadır. Birinci bölüm “Giriş” bölümü olup amaç ve yapı ifade edilmiştir. İkinci bölüm “Trafik Benzetim Sistemleri” başlığı ile oluşturulmuş ve projede gerçekleştirilen hücresel otomatların yanı sıra diğer trafik benzetim modelleri de incelenmiştir. Üçüncü bölümde “GPGPU(Grafik İşlemcisinde Genel Amaçlı Hesaplama)” başlığı ile birlikte grafik kartları hakkında genel bilgilendirme, CUDA ve rakipleri hakkında özet bilgiler ve heterojen hesaplama gibi terimlerin açıklamaları verilmiştir. Dördüncü bölümde CUDA mimarisi projede uygulanan bilgiler için altyapı oluşturulması amacıyla gerektiği kadar detay verilerek incelenmiştir. Beşinci bölümde uygulama tanıtılmış , materyal ve yöntem bilgileri verilmiştir.Altıncı bölümde sonuçlar ve öneriler aktarılmıştır.

BÖLÜM 2

TRAFİK BENZETİM SİSTEMLERİ

Trafik benzetimleri birçok uygulama ve modeli içermektedir. Gerçekleştirme amaçlarına göre benzetimler detaylandırılabilir. Bu detay seviyeleri: Mikroskopik (çok detaylı), mezoskopik (orta seviyede detaylı) ve makroskopik (az detaylı) olarak sıralanabilir.

Mikroskopik benzetim trafik öğelerinin karakteristik özellikleri modellenir. Örneğin bir otomobilin bölünmüş ana yol üzerindeki maksimum hızı 110km/sa iken bir otobüsün 90km/sa'tır. Mezoskopik benzetimde sistem öğelerinin özellikleri ayrıntılı bir şekilde incelenirken, bu öğeler arası etkileşimler kabaca incelenir. Makroskopik benzetimde ise tüm sistem öğeleri ve aralarındaki etkileşimler en düşük ayrıntı ile incelenir.

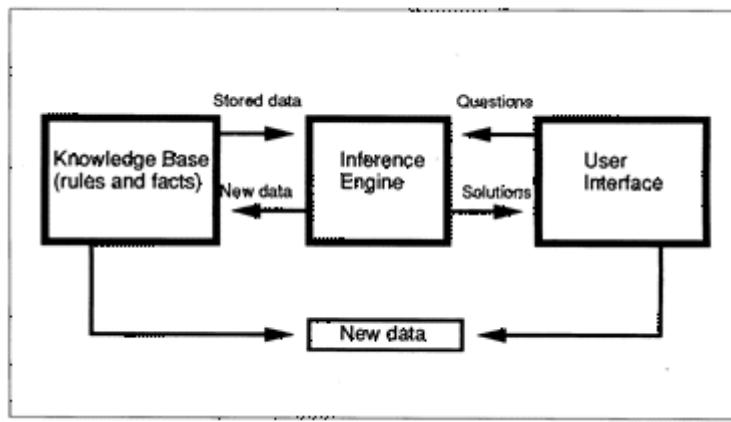
Başka bir sınıflandırma türü ise deterministik ve stokastik sınıflandırmadır. Deterministik modelde rasgele değişkenler mevcut değildir. Tüm öğeler arası etkileşim düzenli ilişkilerle ifade edilmektedir. Stokastik modelde ise olasılık sözkonusudur.[2][3]

2.1 TRAFİK AKIŞI MODELLEME YÖNTEMLERİ

2.1.1 Uzman Sistemler

Uzman sistemler, konusunda uzman olan bir veya birden fazla kişinin bilgi birikimlerine göre muhakeme yapabilme ve karar verebilme işlevlerini modelleyen

yazılım sistemidir.Bir sistemin uzman sistem olabilmesi için kullanıcı hatalarını algılayıp, kullanıcıyı yönlendirebilme yetisine de sahip olması gereklidir.[4]



Şekil 2-1: Bir Uzman Sistemin Yapısı[5].

Bilgi Tabanı(Knowledge Base), bilgilerin tutulduğu ve tutulan bilgilerden yeni bilgiler üretilmesine olanak sağlayan birimdir. Bir anlamda uzman sistemin beyni olarak nitelendirilebilir.

Uzman(expert), sisteme yeni bilgiler kazandırmakla görevlidir. Bu bilgi kazandırma işlemi genellikle sıradan birinin yapabileceği çok konusunda uzman bir kişinin yapması gereken bir iştir.

Çıkarım modülü(Inference Engine), veritabanını da kullanarak bilgi tabanı üzerinde kendisine verilen önermelerin doğruluğunu araştıran modüldür.

Kullanıcı Arayüzü(User Interface), kullanıcının sistem ile etkileşimini sağlamakla görevlidir. Örneğin kullanıcı tarafından verilen bir önermeyi çıkarım modülünün anlayacağı bir ifade şekline getirmek bu birimin görevidir.[6]

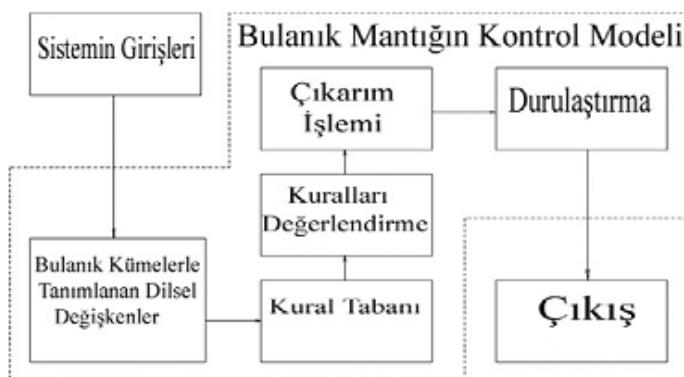
Trafik benzetimlerinde uzman sistemler, kavşak kontrolü ve sinyalizasyon planlamalarında kullanılabilirler. Ayrıca benzetim sonucu elde edilen verilerin yorumlanması gibi işlemler de uzman sistemler kullanılarak gerçekleştirilebilir.[7]

2.1.2 Bulanık Mantık

Bulanık mantık bir önermenin kesin olmayan doğruluk değerine dayanmaktadır. Klasik mantıkta bir önerme ya bir kümenin ögesidir ya da o kümenin ögesi değildir. Bulanık mantıkta ise her bir varlığın üyelik derecesi mevcuttur.[8]

Bulanık kümelerde dereceli üyelik tanımını ilk kez 1965 yılında Prof.Dr. Lütfi A. Zade(Lotfi Zadeh) yayınlanan bir makalesinde yapmıştır. Zade, bu çalışmasında insan düşüncesinin büyük bir çoğunluğunun bulanık olduğunu belirtmiştir. Bu yüzden 1 ve 0 ile ifade edilen boolean mantık insan düşüncesini ifade etmekte yetersizdir. Bulanık mantık bilinen klasik mantık gibi (0,1) olmak üzere iki seviyeli değil, [0,1] aralığında çok seviyeli doğruluk değerlerini ifade etmektedir.[9]

Bulanık mantık insan düşüncesini temel alır. Örneğin, bir kavşakta sinyalizasyonun uygun bir şekilde ayarlanabilmesi ve trafik yoğunluğuna göre güncellenebilmesini sağlamak amacıyla bulanık mantık ile geliştirilen sistem bir trafik polisinin tutumunu taklit edebilir. Trafik polisi kavşağı kontrol altında tutarken yoğun olan şeritlere daha uzun geçiş süresi verirken, az yoğun olan şeritlere daha az geçiş süreleri tanımlamaktadır. Burada olduğu gibi “yoğun”, “az yoğun”, “çok yoğun” kavramları bulanık terimlerdir. Bu terimlerin bilgisayarın anlayabileceği şekilde ifade edilebilmesi için bulanık mantıktan faydalansılır. [10]



Şekil 2-2:Bulanık Mantık Tabanlı Sistemin Yapısı[10].

Bulanık mantık tabanlı sisteme sistemin girişleri önce bulanıklaştırılır. Ardından kural tabanına göre çıkarım yapılır ve en son durulaştırılarak çıkışa aktarılır.

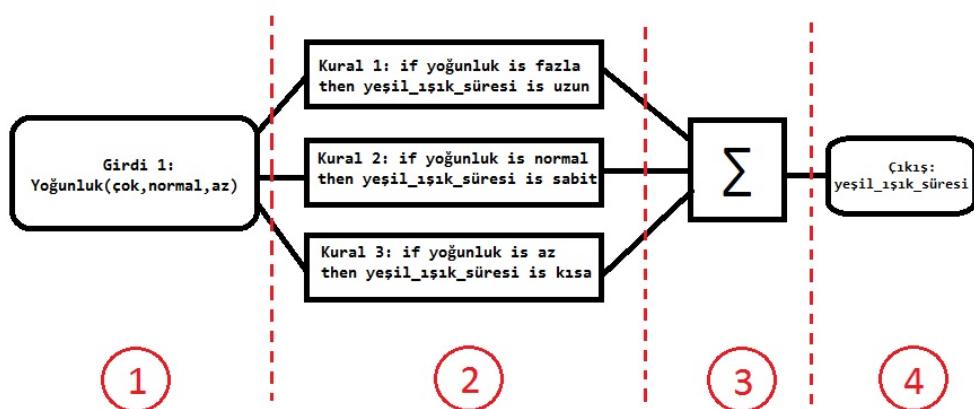
Bulanıklaştırma aşamasında üyelik fonksiyonlarından yararlanılır. Üyelik fonksiyonları, bir bulanık küme içerisindeki kısmi üyeliğin olasılığını ortaya çıkarır ve bulanık kümeye ait olmanın ölçüsünü gösteren üyelik değerini bulanık küme ile eşleştirir.[11]

2.1.2.1 Bulanık Sistemlerde Mantıksal İşlemler(IF-THEN Kuralları)[12]

Bulanık kümeler bulanık mantığın öznisi konumundadır. IF-THEN kuralları ise bulanık mantığın şartlı ifadeleridir. Aşağıdaki x ve y ifadeleri dilsel değişkenleri, A ve B ifadeleri ise x ve y'ye karşılık gelen dilsel değerleri temsil etmektedir.

$$\text{if } x \text{ is } A \text{ then } y \text{ is } B$$

Örneğin, “**if yoğunluk is fazla then yeşil_ışık is uzun**” şeklindeki bir kuralda “fazla” ifadesi aynı zamanda 0-1 aralığındaki bir sayıya karşılık gelmektedir. Her bir kuralın çıktısı aynı zamanda bir bulanık kümedir.



Şekil 2-3: Tek girdili bir sinyalizasyon bulanık sistem örneği

Şekil 2-3'te görülmekte olan 1 nolu aşamada girdiler bulanıklaştırılmıştır. 2 nolu aşamada kurallar , paralel bulanık muhakeme kullanılarak değerlendirilmiştir ve 3

nolu aşamada kuralların sonuçları birleştirilmiş , elde edilen bulanık çıktı kümesi durulaştırılmıştır. 4 nolu aşama ise kesin bir sayının elde edildiği sonuç aşamasıdır.

2.1.2.2 Bulanık Mantığın Temel Özellikleri[12]

Bulanık mantığın temel özellikleri aşağıdaki gibi özetlenebilir:

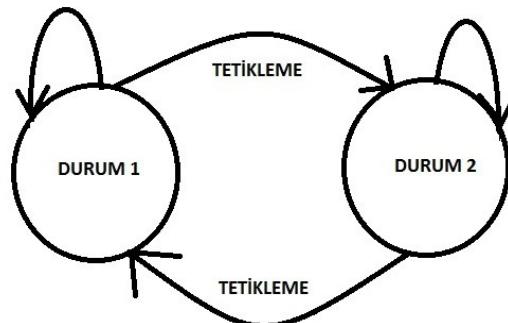
- Bulanık mantıkta her ifadenin bir önem derecesi vardır.
- Herhangi bir mantıksal sistem bulanıklaştırılabilir.
- Bulanık sistemler belirsiz modeller ve özellikle de matematiksel temelde kurulamayan sistemler için çok uygundur.
- Esnektir.
- Kesin olmayan verilere tolerans tanıyarak yaklaşır.
- Karmaşık yapıdaki doğrusal olmayan işlevleri tanımlayabilir.
- Bulanık mantık sistemleri doğal diller üzerine kurulur. Bu tip sistemlerdeki etkileşim insanlar arasındaki iletişime benzer.

2.1.2.3 Bulanık Mantığın Kullanılamayacağı Durumlar[12]

Bulanık mantık çok sayıda giriş verisi ile çok sayıda çıkış verisi arasında ilişkilendirme yapacağından, eğer daha kesin ve kolay bir model varsa bulanık mantık kullanılmaz. Ayrıca bulanık mantık insan aklına uygun kararlara bağlı olarak ortaya çıkarılan kuralları kullanacağından, kuralları düzgün bir şekilde oluşturulamamış sistemlerde bulanık mantık kullanımından kaçınılmalıdır.

2.1.3 Sonlu Durum Makineleri (Finite State Machines)

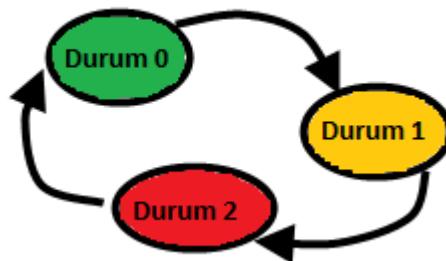
Sonlu sayıda durum içeren ve bulunduğu durumu başka bir tetikleme olmadığı sürece koruyan modellere sonlu durum makineleri ya da sonlu durum otomatları denilmektedir.



Şekil 2-4 :Bir Sonlu Durum Makinesinin Geçiş Diyagramı

Şekil 2-4'de gösterilen Sonlu Durum Makinesinin geçiş diyagramında yer alan Durum 1 başlangıç durumunu ifade etmektedir. Durum 1 tetiklendiğinde Durum 2'ye geçer. Tetikleme olmadığı sürece de mevcut durumunu korur.[13]

Trafik benzetimlerinde sinyalizasyonu sağlamak amacıyla genellikle sonlu durum makinelerini kullanılmaktadır.Şekil 2-5'te trafik sinyalizasyonunun sonlu durum makineleri ile ifadesi görülmektedir.



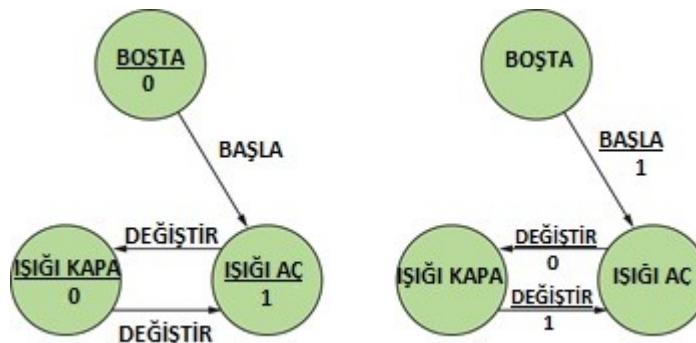
Şekil 2-5: Trafik Işıklarının SDM ile Modellenmesi

2.1.3.1 Sonlu Durum Dönüştürüçüler (Finite State Transducer)

Giriş(Input) ve Çıkış(Output) durumlarına sahip olan sonlu durum modelidir. Dönüştürüçüler, verilen girdi ve eylemleri kullanarak ortaya çıkan durumlara dayanarak çıktı üretirler. Kontrol uygulamaları için kullanılırlar[14].

Moore Makinesi ve Mealy Makinesi

Sonlu Durum Makineleri sadece giriş eylemlerini kullanır, çıkış duruma bağlıdır. Mealy Makinesinin Moore Makinesinden farkı, çıkışının sadece o anki durumuna bağlı değil, girişe de bağlı olarak değişmesidir.



Şekil 2-6:Bir SDM'nin Moore Makinesi (Sol) ve Mealy Makinesi(Sağ) Modelinin Durum Geçiş Diyagramı[15]

2.1.4 Etmen Tabanlı Modelleme

Etmenler, uyarlanabilen, öğrenebilen ve sosyal olma gibi insana özgü özelliklere sahip bilgisayar yazılımı ya da donanımıdır. Etmenler, yapay zekânın bir konusudur. Etmenlerin karakteristik özellikleri Sundsted'in 2004 yılında yayınlanmış makalesinde aşağıdaki gibi sıralanmıştır. [16]

- Otonom
- Uyarlanabilen/Öğrenebilen
- Hareket hâlinde
- Kalıcı
- Amacı olan
- Haberleşebilen, çevresiyle etkileşim hâlinde olan
- Esnek
- Reaktif
- Proaktif
- Sürekli/Kesintisiz

Verilen özellikler tek tek açıklayalım [17] :

- 1. Otonom olma(Özerklik):** Etmenlerin dışarıdan bir etki beklemeksizin kendi kendilerine hareket edebilmeleri anlamına gelir.
- 2. Uyarlanabilen/Öğrenebilen:** Etmenlerin deneyimlerine dayanarak hareketlerini güncelleyebilmelerini ifade etmektedir.
- 3. Hareket Hâlinde Olma:** Bazı etmenler bulundukları ortamda hareket hâlindedir. Bazı etmenler ise bir bilgisayarda başlayan işlemi başka bir bilgisayarda sürdürübirlirler.
- 4. Kalıcı Olma:** Bir etmen belirli bir ortamda bulunur ve o ortmanın parçasıdır.
- 5. Amaç Sahibi Olma:** Bu özellik, etmenin hedefine ulaştığında bulunacağı durumu ifade etmektedir. Örneğin, trafik benzetiminde bir aracın istediği konuma güvenli bir şekilde gitmesi etmenin amacıdır.
- 6. Çevresiyle İletişim Hâlinde Olma:** Etmenlerin diğer modellerden en belirgin farkı bulunduğu konuma göre içinde bulunduğu konum ve çevresiyle etkileşim durumunda olmasıdır. Örneğin, bir trafik benzetiminde bulunan kavşak etmeni, araç etmeni ve şerit etmeni bulunsun. Araç etmeni şerit üzerinde ilerlerken karşılıklı veri传递i içersindedirler. Benzer olarak araç etmeninin kavşak noktasına geldiğini anlayabilmesi için kavşak etmeni ile araç etmeninin etkileşim içerisinde olması gereklidir.
- 7. Esnek Olma:** Etmenlerin hedeflerine ulaşabilmesi için alternatif planlara sahip olması durumudur. Bu alternatiflere göre seçme yetisine sahiptirler.
- 8. Reaktif Olma:** Etmenlerin bulundukları ortamdaki değişikliklere doğru zamanda tepki verebilme yetisidir.
- 9. Proaktif Olma:** Etmenlerin dışarıdan gelen etkiler dışında, etmenin içsel olarak amacına yönelik hareket etmesi demektir.
- 10. Sürekli Olma:** Çalışma ortamı var olduğu sürece etmenin de var olması anlamını taşımaktadır. Etmenin çalışması herhangi bir sebepten sekteye uğrasa bile etmen kendini yenileyebilmeli ve kaldığı konumdan devam edebilmelidir.

Trafik benzetimi tasarılanırken, yoğun bir trafiğin kuralları statik değil dinamik olmalıdır. Trafik ışıklarının sinyalizasyon işleminin gün içerisinde değişken yoğunluk koşullarına göre güncellenmesi bu duruma örnek olarak verilebilir. Trafik uyarlanabilen bir şekilde incelenmelidir.

2.1.4.1 Yapılan Önceki Çalışmalar

Trafik Kontrolü için Çoklu Etmen Tabanlı Etmen Sistemi

Oliviera ve Duarte(2005)'in çalışmasında, araçlar birer nesne olarak tanımlanırken trafik ışıkları ve kavşaklar birer etmen olarak tanımlanmıştır. Bu modelleme tekniğinde araçlar etmen olarak ifade edilmediği için araç farklılıklarından kaynaklanan bazı aksamalar oluşabilmektedir. Örneğin, otomobil ile tır araç tiplerinin şerit üzerinde ilerlemesi aynı hız kurallarına göre yapılmayacaktır. Oysa bu modelde araç tipleri için bir ayırım olmadığından belirgin bir fark elde edilemeyecektir.[18]

Çoklu Etmen Sistemi ile Düzensiz Trafik Simülasyonu Gerçeklenmesi

Paruchuri vd. (1998) çalışmasında her öğeyi etmen olarak tanımlamıştır. Asıl amaç araçların kendini yöneten bir dış etki olmadan(trafik polisi, trafik ışığı vb.) akışın sağlanmasıdır. Ancak bu model yoğun trafik koşullarında etkili bir performans ortaya koyamayabilir.[18]

Trafik Simülasyonunda Etmen Teknolojileri Kullanılması

Erol vd. (1998) bu çalışmasında trafikte yer alan tüm varlıklarını, yolları, kavşakları, trafik ışıklarını, trafik işaretlerini, araçları etmen olarak modellemiştir. Kavşak etmeni trafik ışıklarını yönetmekte ve şeritler arası değişimi kontrol etmektedir[18]

Çoklu Etmen Tabanlı Trafik Yönetim Sistemi

Soysal (2008) çalışmasında kavşak, araç ve şerit varlıklarını etmen olarak tanımlamıştır. Bu modelde akıllı bir kavşak sistemi tasarlanmıştır. Akıllı kavşak sistemi sayesinde acil durum araçları daha kısa sürede olay yerine ulaşacak, trafik sıkışıklıkları azaltılacaktır. Modelleme yapılırken JADE etmen platformu kullanılmıştır. Etmenler arası iletişim mesajlaşma yoluyla olmaktadır. Her etmenin posta kutusuna benzer bir şekilde yapılandırılmış bir mesaj kuyruğu mevcuttur. JADE bu asenkron mesajlaşmayı sağlayabilecek altyapıya sahiptir. Bir etmenin mesaj kuyruğuna ileti geldiğinde etmen bu mesaja yanıt verecek şekilde hareket eder.

Etkileşimli Sürücü Eğitimi İçin Kural Tabanlı Bir Platform Geliştirilmesi

Kurnaz(2012), bu çalışmasında araçları birer etmen olarak tanımlamış ve trafik akışını sağlayan yapıları hiyerarşik eş zamanlı durum makineleri ile modellemiştir. Sürücüler karakteristik özelliklere göre tanımlanmıştır. Araç etmen ilerlerken bulunduğu konuma(segment ya da kavşak) hakimdir. Trafik Simülasyon Sistemi, sürücü eğitimi amacıyla geliştirilmiş olup arayüz ve kural veritabanları buna göre tasarlanmıştır. Sadece etmenlerden değil birçok modelleme tekniğinden faydalansılmıştır.

2.1.4.2 Etmen Tabanlı Modelleme ile Nesne Tabanlı Modellemenin Karşılaştırılması[19]

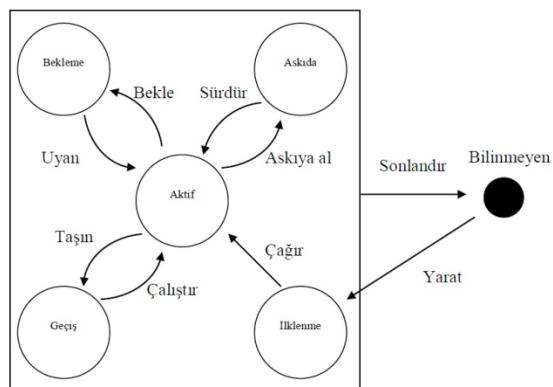
- Ortaya çıkış önceliği NYP yaklaşımındır. EYP kendinden önce gelen NYP'den doğal oalrak etkilenmiştir.
- EYP uygulamalarını , kendi kendilerine çalışan ve belli bir amacı olan NYP uygulamaları şeklinde ifade edebiliriz.
- NYP'nin temel ögesi nesne iken EYP'nin temel ögesi etmendir.
- Dışarıdan bir etki ile harekete geçen varlıklar için nesneler daha uygundur. Kendiliğinden aktif duruma geçebilen varlıklar içinse etmenler daha doğru bir seçim olacaktır.
- Nesneler gelen iletılere, önceden kendisine belirtildiği şekilde cevap verir. Nesneye bir iletinin gelmesi demek aslında o nesnenin bir işi yapmasını

istemek anlamındadır. Etmenlerde ise gelen iletiye cevap niyet ve amaç gibi faktörler doğrultusunda yanıt verilir.

- NYP'de bir işlemin yapılp yapılmayacağı ya da hangi işlemin yapılacığı nesneyi çağıran işlev tarafından belirlenirken, EYP'de etmenler çevresel değişikliklere, bilgi birikimlerine ve amaçlarına göre bir işlemi gerçekleştirdip gerçekleştirmeyeceklerine karar verirler.
- Nesneler bilgilerini diğer nesnelerin erişebileceği şekilde ya da gizlenmiş bir biçimde tutarken, etmenler çevreden aldıkları bilgileri de ekleyerek oluşturdukları ve sürekli güncelledikleri bir bilgi tabanında tutarlar.
- Nesneler değişken davranışlar sergileyebilirler ancak bu davranışların da dışarıdan tetiklenmesi gerekmektedir. Etmenler ise özerk bir şekilde hareket etmektedirler.

2.1.4.3 Etmen Yaşam Döngüsü

FIPA(Foundation for Intelligent Physical Agents) standardına göre belirlenmiş etmen yaşam döngüsü şekil 2-7'de görülmektedir.

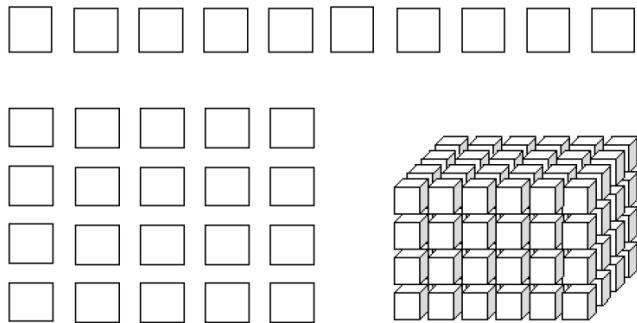


Şekil 2-7: FIPA Standardına Göre Etmen Yaşam Döngüsü[20]

2.1.5 Hücresel Otomatlar(Cellular Automaton)

2.1.5.1 Genel Bakış

Hücresel otomatlar fiziksel sistemlerin matematiksel olarak idealleştirilmesidir.[21] Otomatın hücresel olarak adlandırılmasının sebebi, sistemin hücrelerden meydana gelmiş olmasıdır. Her bir hücre sonlu durum makinası olarak çalışmaktadır. Hücesel otomatlar, hücrelerin birbirilerini etkilemesi şeklinde oluşan yapılardır. Hücrelerin düzenli dizilmesi ile ızgara(grid) oluşturulur. Hücresel otomatlar 1B, 2B ya da 3B olabilirler.

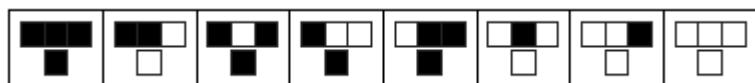


Şekil 2-8: 1B, 2B ve 3B'lu Hücresel Otomat Örnekleri[21]

2.1.5.2 CA Çalışma Mantığı ve Trafik Akışında Kullanılan Kural-184

Hücresel otomatın çalışma mantığı, hücrenin komşularını hesaba katarak adım adım iterasyonların işletilmesine dayanmaktadır[22]. 184-Kuralı üzerinden bir hücresel otomatu inceleyelim:

Her bir hücre, bir kutucuk ile ifade edilir ve bu kutucukların içi ya dolu olur ya da boş. Her adımda, üstteki 3 hücrenin durumu alttaki bir hücrenin durumunu belirlemekte kullanılacaktır.[22]



Şekil 2-9: Kural-184'e göre hücrelerin durumu

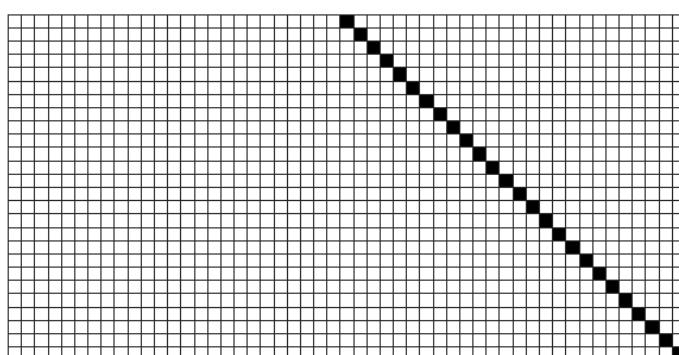
Şekil 2-9 göz önüne alındığında ve siyah kutular 1, beyaz kutular ise 0 olarak kabul edildiğinde aşağıdaki doğruluk tablosu ortaya çıkmaktadır:

Çizelge 2-1: Kural-184'ün Doğruluk Tablosu

$t-1$	t_{184}
000	0
001	0
010	0
011	1
100	1
101	1
110	0
111	1

Bu kurala “Kural-184” denilmesinin sebebi, hücrenin yeni değerinin 2’li sayı sisteminde almış olduğu toplam değerin 10’luk tabanda karşılığının 184 çıkmasıdır. Çizelge 2-1’de t_{184} sütunu incelendiğinde $(2^0*0) + (2^1*0) + (2^2*0) + (2^3*1) + (2^4*1) + (2^5*1) + (2^6*0) + (2^7*1) = 184$ şeklinde sonuçlanacaktır.

Üstteki üç kutunun $2^3=8$ farklı durumuna göre 10111000 şeklinde bir kural oluşturuldu. Dolayısıyla 8 basamaklı olan bu yeni duruma göre $2^8=256$ farklı kural kombinasyonu oluşturulabilir.[22]

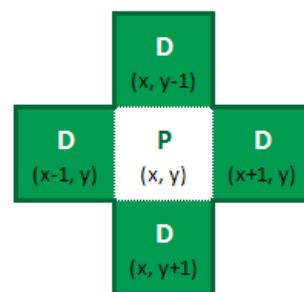


Şekil 2-10: Kural-184'e göre bir hücresel otomatın 25 iterasyon işlenmiş hâli

2.1.5.3 İki Boyutlu Hücresel Otomatlar

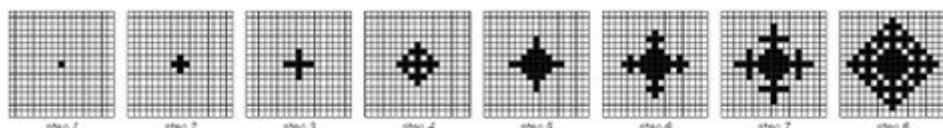
Önceki bölümde tek boyutlu hücresel otomat yapısı incelenmiştir. Projede 2B karayolu trafik akışı modelleneceği için bu bölümde de iki boyutlu hücresel otomatların yapısı incelenecaktır ve komşuluk kavramları detaylandırılacaktır.

Von Neumann komşuluğuna göre bir hücrenin yeni durumu, bu hücrenin 4 komşusunun durumuna bağlı olarak değişmektedir.



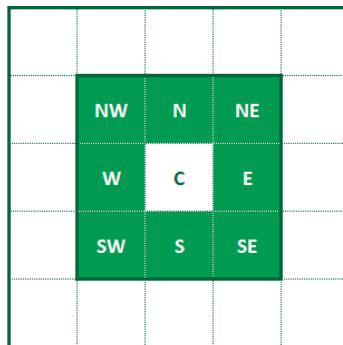
Şekil 2-11 :Von Neumann komşuluğunda merkez hücre(P) ve komşuları(D)

Komşulardan sadece biri veya dördü birden siyah olduğunda, hücrenin yeni durumu siyah olacak mantığı ile elde edilebilecek durumlar Şekil 2-12'de gösterilmiştir [22].



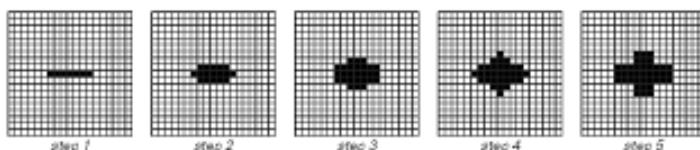
Şekil 2-12: Dörtlü komşuluğa göre oluşan 2B bir hücresel otomat

Moore komşuluğunda ise, bir hücrenin yeni durumu, bu hücrenin 8 komşusunun durumuna bağlı olarak değişmektedir.



Şekil 2-13: Moore Komşuluğunda merkez hücre(C) ve komşuları

Komşularından 3 veya 5inin siyah olması durumunda siyah aksi takdirde beyaz olan bir hücresel otomatın durumları Şekil 2-14'de gösterilmiştir [22].



Şekil 2-14 : 5 iterasyon boyunca hücresel otomatın bulunduğu durumlar

2.1.5.4 Nagel-Schreckenberg Modeli[23]

Karayolu trafik akışının 1B'lu hücresel otomatlar ile benzetiminde en temel model “CA Kural-184”tür. Nagel-Schreckenberg modeli, Kural-184’ün bir türevidir. NaSch modeli, tek şeritli bir yolu tanımlar. Bu yol birebir ardına gelen noktaların bir ızgara oluşturmasıyla meydana gelir. Izgaranın her bir karesinde yalnızca tek bir araç bulunabilmektedir. NaSch modeli, araçların hızlanmasıının ve yüksek hızdaki araçların yavaşlatılmasının etkilerini dikkate almaktadır[24].

Bu hesaplama modelinde L dizisi, konumların tek boyutlu ifadesidir. Her konum ya bir araç ile işgal edilmiş ya da boştur. Her aracın hızı tamsayı ile ifade edilen değere sahiptir. Bu hız sıfır(0) ile v_{\max} arasındaki değerlere sahiptir. Rastgele bir dizilim için, aşağıdaki ardışık 4 basamak tüm araçların paralel olarak başlaması için güncellenir.

1- **Hızlanma:** Eğer aracımızın hızı v , v_{\max} 'tan düşük ise ve eğer öndeki aracın hızı $v+1$ 'den büyükse, aracımızın hızını bir kademe ilerletiriz [$v \rightarrow v+1$].
 Örneğin; eğer hız 4 ise, 5'e yükseltilir.[25]

2- **Diğer Araçlara Bağlı Olarak Yavaşlama:** Eğer i konumundaki bir araç, önünde $j+i$ konumundaki bir aracı görürse ($j \leq v$), hızını ($j-1$) kadar azaltır.

Örneğin; birinci aracın hızı 5, aracın önünde sadece 3 boş hücre var ve dördüncü hücrede başka bir araç var ise birinci aracın hızı 3'e indirilir.[25]

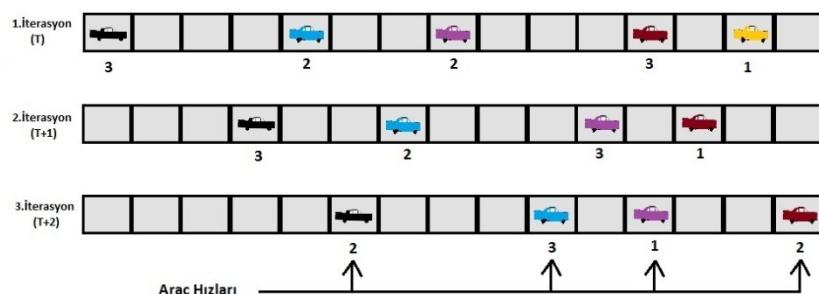
3- **Rasgele Dağılım:** p olasılığı ile her bir aracın hızı (eğer sıfırdan büyükse) 1 azaltılır [$v > v-1$].

Örneğin; eğer $p=0,5$ ve hız 4 ise, zamanın %50'si bir sürede hız 3'e indirilir.[25]

4- Araç Hareketi: Her araç, v hızını kullanarak konumlarını ilerletirler.

Örneğin; eğer hız 3 ise, araç 3 hücre ileri taşınır.[25]

Yukarıda sayılan maddelerin hücreler üzerine uygulanması örneği Şekil 2-15'te görülmektedir. Her bir iterasyon benzetimin güncellenmesi (time-step) anlamına gelmektedir.



Şekil 2-15: NaSch Modeline Göre Araçların İlerlemesi Gösteren Örnek
 (Deterministik)

NaSch modeline göre “randomisation rule”, insan davranışları ya da çeşitli harici durumlar nedeniyle oluşan doğal hız dalgalanmalarını yakalar. Bu kural, sürücülerin

fren yapamsındaki tepkisini ortaya koymaktadır ve gecikmeli hızlanmayı tanımlamaktadır.

Trafik benzetimlerinde genellikle şehir trafiğinin rasgele dağıtım olasılığı $p=0.5$ alınırken, otoban/anayol trafiğinin rasgele dağıtım olasılığı $p=0.3$ alınmaktadır.

2.1.5.5 Yapılan Önceki Çalışmalar

Nagel-Schreckenberg 'in modeli temel alınarak birçok model türetilmiştir. Tek şeritli model çok şeritli şeklinde tasarlanmış ve bazı yeni kurallar eklenerek güncellenmiştir.

Çevreyolları İçin Hücresel Otomata Modeli

Nagel ve Schreckenberg (1992) bu çalışmasında tek şeritli bir otoyol üzerinde , en yüksek araç hızının 5 hücre olduğu ve her bir hücrenin 7.5 metre olduğu bir tasarım kullanmışlardır. Her bir hücre ya doludur(yani üzerinde araç bulunmaktadır) ya da boştur(üzerinde araç bulunmamaktadır). Aracın ilerlemesi için Bölüm 2.1.5.4'te anlatılan kurallar kullanılır. Bu modelde her bir aracın uzunluğu aynıdır ve her biri bir hücre boyutundadır. Modellenen sınır dışına çıkan her araç , sisteme yeniden dahil olmaktadır. Bu durum tipki bir araba yarışı gibi düşünülebilir fakat burada yol tek şeritlidir. Her bir aracın ilerlemesi için yenilenme süresi 1 saniye olarak belirlenmiş ve bir şeritte 10000 hücre kullanılmıştır.[25]

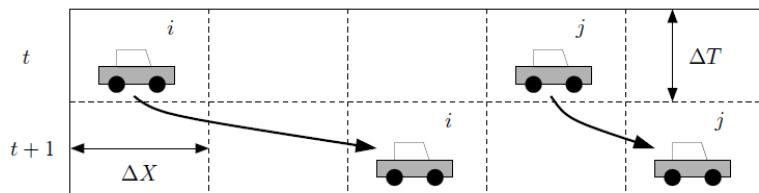
Hücresel Otomatlar Kullanılan İki Şeritli Trafik Benzetimi

Rickert, Nagel , Schreckenberg , Latour (1996) bu çalışmada tek şeritli yol modelinin yetersizliği üzerine iki şeritli modeli geliştirmiştir.Gerçek hayatı bulunan araç tiplerine göre hızları da farklılık göstermektedir. Örneğin tek şeritli bir yolda tır tipinde bir araç en önde giderken, arkasında kalan araçlar da tırın maksimum hızına göre hareket etmek zorundadır. Ancak iki şeritli yol kullanıldığında şerit değiştirme kavramları da ortaya çıkacaktır ve model daha

gerçekçi bir hâl olacaktır. Stokastik modelde yer alan randomization(rasgelelik) ifadesi yanında lane changing probability(şerit değiştirme olasılığı) ifadesi de bu modelde yer almıştır.[26]

Karayolu Trafiğinin Hücresel Otomat Modelleri[27]

Maerovet ve DeMoor (2005) bu çalışmalarında mikro trafik akış modelini ve hücresel otomat modelini temel almışlardır.TCA(Traffic Cellular Automaton) olarak adlandırdıkları uygulamalarında birçok hücresel otomat kalibini kullanmışlardır. Deterministik model olan Wolfram'ın 184 nolu kuralı, stokastik bir model olan NaSch modeli ve “Slow-to-start models” olarak adlandırılan birçok model bu uygulamada yer almıştır.



Şekil 2-16: TCA'da yer alan tek şeritli hücresel otomat yapısı

Şekil 2-16'te görülmekte olan i ve j araçlarının kapladıkları hücreyi ve hücresel otomatın boyutunu ifade eden ΔX , 7.5 olarak alınmıştır. Her bir aracın ilerlemesi için gerekli yenilenme süresi olan ΔT ise 1 sn olarak alınmıştır. Bir hücreyi geçmek için gerekli ortalama hız $\Delta V, 27 \text{ km/sa}$ olarak hesaplanmıştır. i ve j araçları farklı hızlara sahiptir bu yüzden hücreler üzerinde ilerlerken güncelleme kendi hızlarına göre olacaktır.

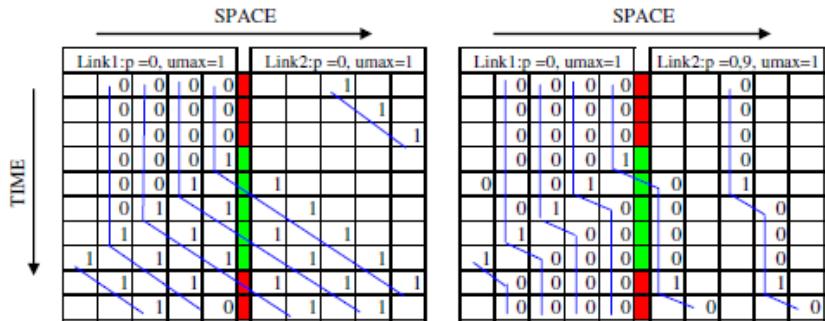
Sinyal kontrollü Trafik Akışının Hücresel Otomat ile Modellenmesi[28]

Trafik benzetimleri için kullanılan hücresel otomat temelli modellerden ilki NaSch modelidir. Diğer modeller NaSch'in modifikasyonlarıdır. Spyropoulou (2007) , bu çalışmasında modele trafik sinyalizasyonunu da katmıştır. Sinyalizasyon sayesinde kavşak tanımlanmış ve bu kavşak üzerinden araç geçişleri sağlanmıştır.

NaSch modelinde rasgelelik parametresi $p=0$ olursa model deterministik olur. p 'nin sıfır dışında bir değere sahip olması ise sistemi stokastik yapar. Şekil 2-17 ve Şekil 2-18'de , deterministik ve stokastik modellerin sinyalizasyon şemaları görülmektedir.

t/x			S	P	A	C	E
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	1
5	0	0	0	0	0	1	2
6	0	0	0	0	0	1	2
T 7	0	0	0	0	1	2	3
I 8	0	0	0	1	2	3	4
M 9	0	0	0	1	2	3	4
E 10	0	0	1	2	3	4	4
11	0	1	2	3	4	4	4
12	1	2	3	4	4	4	4
13	2	3	4	4	4	4	4
14	3	4	4	4	4	4	4
15	4	4	4	4	4	4	4
16	3	4	4	4	4	4	4
17	3	4	4	4	4	4	4
18	4	4	4	4	4	4	4
19	4	4	4	4	4	4	4
20							

Şekil 2-17: Deterministik modelde aracın sinyalizasyona göre konum-zaman şeması



Şekil 2-18: Stokastik modelde sinyalizasyona göre aracın konum-zaman şeması

Çok Şeritli Otoban Modellemesi ve Trafik Tikanıklığının Analizi[29]

İmrem(2008), bu çalışmasında diğer örneklerde olduğu gibi NaSch modelini temel almış ve çok şeritli olarak güncellemiştir. Sık şerit değiştirme, takip mesafesinin olması gerekenden az bırakılması ve araç yoğunluğunun fazla olması gibi nedenlere dayalı trafik sıkışıklığının analizi yapılmıştır. Bu analizlere göre rasgelelik parametresi (randomisation parameter) belirlenmiş ve deney amaçlı değişikliklere

uğratılarak sonuçlar gözlemlenmiştir. İmrem'in bu modelinde NaSch'in 4 temel kuralına ek olarak "Şerit Değiştirme Kuralı", "Hız Adaptasyon Kuralı" , "Çıkış Şeridinden Geçiş Kuralı", "Çevreyoluna Giriş Şeridinden Geçiş Kuralı" ve "Şeride Araç Ekleme Kuralı" gibi kurallar dahil edilerek benzetim modellenmiştir.

BÖLÜM 3

GPGPU (GENERAL PURPOSE COMPUTING ON GRAPHICAL PROCESSING UNIT) MODELİ

Grafik İşlem Birimi(GPU)'nin temel görevi bilgisayarda oluşturulan görüntülerin ekrana verilmesini sağlamaktır.Bu yüzden ilk GİB'ler sadece bu görevi yerine getirmektedir. Zaman içerisinde Merkezi İşlem Birimi(CPU)'nin karşılaşılan büyük hesaplama problemlerinde yetersiz kaması üzerine GİB'nin donanımsal paralelliğinden yararlanma fikri ortaya çıkmıştır. GİB'lerin programlanabilir bir arayüze sahip olmasını ve yüksek seviyeli dillerle programlanabilmesini sağlamak için GPGPU modeli oluşturulmuştur.

3.1 GPGPU PLATFORMLARI

3.1.1 DirectCompute

Windows işletim sistemi üzerinde GPGPU programlama gerçekleştirilmesi amacıyla Microsoft firması tarafından geliştirilmiş bir API'dir.[30]

3.1.2 CUDA

CUDA(Compute Unified Device Architecture), NVIDIA firmasının 2006 yılında GİB'nin donanımsal hesaplama gücünden faydalananmak amacıyla sunduğu paralel hesaplama mimarisidir. Linux, Windows ve Mac Osx platformları üzerinde çalışabilmektedir.FORTRAN, C/C++ ve Python gibi dilleri destekleyen bir API'dir.Rakiplerine göre avantajları paylaşımı bellek kullanımı, GİB'den daha hızlı veri okuma ve bit düzeyinde işlem yapılabilmesine olanak sağlama olarak sayılabilir.[31]

3.1.3 OpenCL

OpenCL(Open Computing Language), 2009 yılında Khronos Group tarafından piyasaya sunulan , NVIDIA, AMD/ATI,Intel, IBM ve Texas Instruments gibi firmalar tarafından gelişimi desteklenen platform ve CUDA'dan farklı olarak aygıt bağımsız olan bir hesaplama çatısıdır.

OpenCL, MİB'in seri işlemlerde ve GİB'in paralel işlemlerde sahip oldukları hız avantajlarını harmanlayarak heterojen hesaplama imkânı sunar.

Donanımsal olarak farklılıklar olmasına rağmen OpenCL'in bu farklılıklardan etkilenmeden programlama geliştirimesine olanak sağlaması için bazı standartlar ortaya koyulmuştur. Her aygıt üreticisi işlevleri kapalı kaynak kodu olarak saklamalarına rağmen, oluşturulan bu standartlara uymak zorundadırlar. Bu sayede aygıt bağımsız programlar geliştirilebilmektedir. Kullanıcılar bu standartlar sayesinde aygıtın teknik ayrıntılarından da soyutlanmaktadır. [32][33]

3.1.3.1 CUDA ile OpenCL Terminolojisinin Karşılaştırılması

Heterojen hesaplamaya olanak sağlayan iki önemli platform olan CUDA ve OpenCL'in terminolojik karşılaştırılması Çizelge 3-1'de görülmektedir. Buradaki

terimlerden CUDA'ya ait olanlar proje çalışması boyunca kullanılacak ve gerekli açıklamalar yapılacaktır. OpenCL terimleri ise karşılaştırma yapılması amacıyla verilmiştir.

Çizelge 3-1: Grafik İşlemcisinde Genel Amaçlı Hesaplama

NVIDIA CUDA	OpenCL
Multiprocessor (SM)	Compute Unit(CU)
Streaming Processor	Processing Element
Global Memory	Global Memory
Shared Memory	Local Memory
Local Memory	Private Memory
Kernel	Kernel
Block	Work-Group
Thread	Work-Item
Warp	Wave Front

3.2 GRAFİK KARTLARI VE GRAFİK İŞLEM BİRİMİ (GİB)

Grafik kartları, grafikle ilgili işlemleri yapmak ve bilgisayarın görüntü bileşenine çıktı üretmek için üretilmiş kartlardır. Grafik kartları üzerinde donanım ile haberleşmeyi sağlayan BIOS, bellek ve işlemci (GİB) bulunmaktadır.

GİB'lerin hızla gelişmesine en büyük katkı bilgisayar oyunlarına artan ilgidir. Görüntü kalitesinin artması sağlanırken grafik işlemciler, bilgisayarın merkezi işlem biriminden çok daha iyi seviyelere gelmiştir. Daha yüksek işlem kapasitesine sahip olmuşlardır. Aynı anda bir çok thread(iş parçası) işleyebilecek çok çekirdekli bir yapıdadırlar.

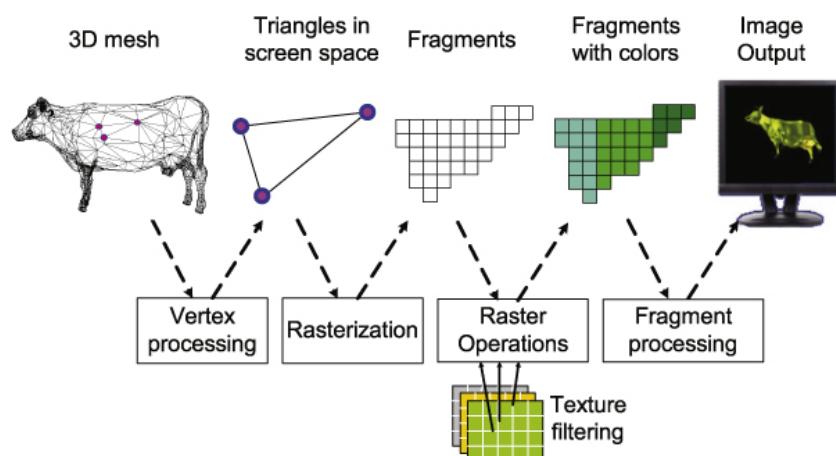
GİB'lerin MİB'den farkı, SIMD(Single Instruction Multiple Data-Tek Komut Çoklu Veri) mimarisine sahip olmasıdır.Şekil 3-1'de MİB(soldaki) ve GİB(sağdaki)'lerin donanımsal olarak karşılaştırılması gösterilmiştir. MİB hesaplamalarını seri bir şekilde gerçekleştirirken, GİB hesaplamalarını yapısal olarak paralel olması sebebiyle paralel bir şekilde gerçekleştirmektedir.

Bir GİB'de , MİB'de olduğundan daha fazla transistör ve çekirdek bulunmaktadır. Bu sayede hesaplamalarını Merkezi İşlem Birimi(CPU)'ne göre daha hızlı yapmaktadır.

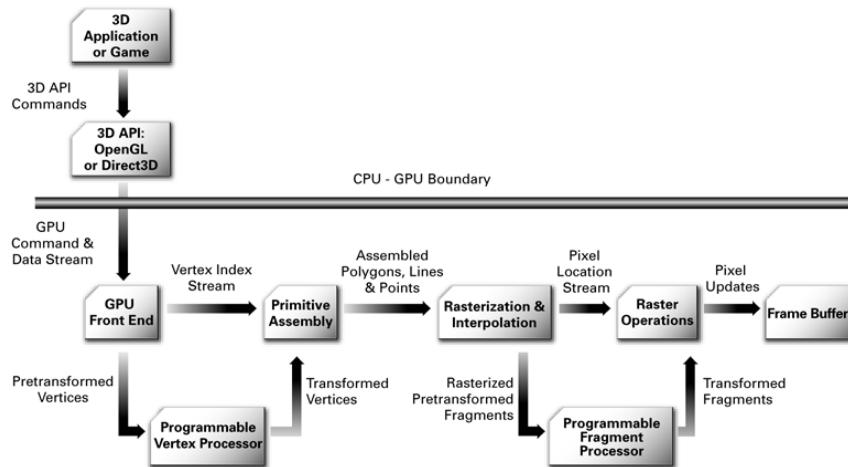


Şekil 3-17: MİB(CPU) ve GİB(GPU) Donanımsal Olarak Karşılaştırılması[35]

Grafik İşlem Birim’leri; 3 boyutlu verinin 2 boyuta dönüştürülmesi(transformation), verinin köşe noktalarının birleştirilmesi (vertex assembly), veriden parçalar oluşturma (fragmentation), parçaların piksellere dönüştürülmesi (rasterization), doku eşleme (texture mapping), verideki pürüzleri giderme (anti-aliasing) gibi çok büyük miktardaki veri üzerinde işlemler yapmak üzere tasarlanmıştır. [33]



Şekil 3-18: Grafik İşlem Biriminin İş Hattında Veri Dönüşümünü Örnекleyen Görsel[36]

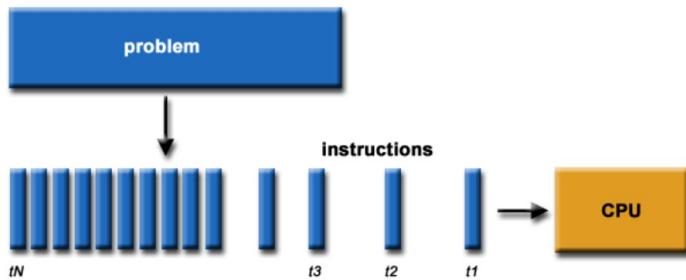


Şekil 3-19: Grafik İşlem Birimi İş Hattı Mimarisi[37]

İşlem hattı ardışık olarak gerçekleştirilen işlemlerden oluşmaktadır. Her bir adımda önceki aşamanın sonuç verisi, şimdiki aşamanın giriş verisi olarak kabul edilir ve işlenir. Bu aşamaların sonucunda 3 boyutlu düzlemede oluşturulan veri , 2 boyutlu düzlemede gösterilecek formata çevrilir.

3.3 SERİ HESAPLAMA

Bir problemin komutlara ayrılarak bu komutların tek tek sırasıyla işlemcide hesaplanması demektir. Bir komut işlemciye girer ve hesaplama bittikten sonra diğer komuta geçilir.

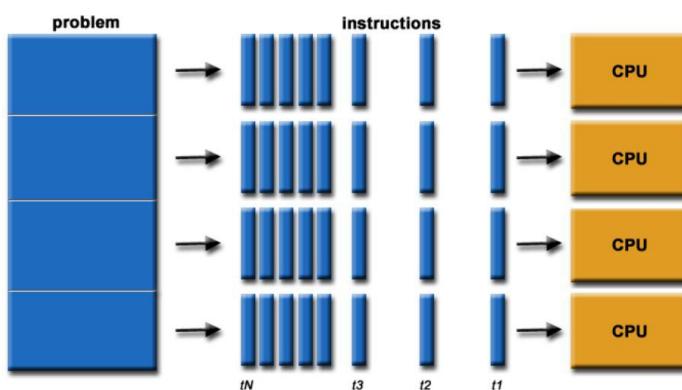


Şekil 3-20: Seri Hesaplama ile Komut Yürütülmesi[38]

Seri hesaplamada, yazılan uygulama tek işlemci üzerinde çalışmaktadır. Problem komutlara ayrılmıştır. Herhangi bir anda yalnızca bir komut yürütülmektedir.

3.4 PARALEL HESAPLAMA

Bir problemin parçalara ayrılarak, bu parçaların iki veya daha fazla işlemci üzerinde paralel olarak hesaplanması modelidir. Paralel hesaplama birden fazla MİB kullanılmaktadır. Problemin parçalanması sonucu oluşan her bir komut, kendine ait MİB'de diğerlerinden bağımsız olarak farklı zamanlarda yürütülebilir, eş zamanlı olması gereklidir.



Şekil 3-21: Paralel Hesaplama Komutlarının Yürütülmesi[38]

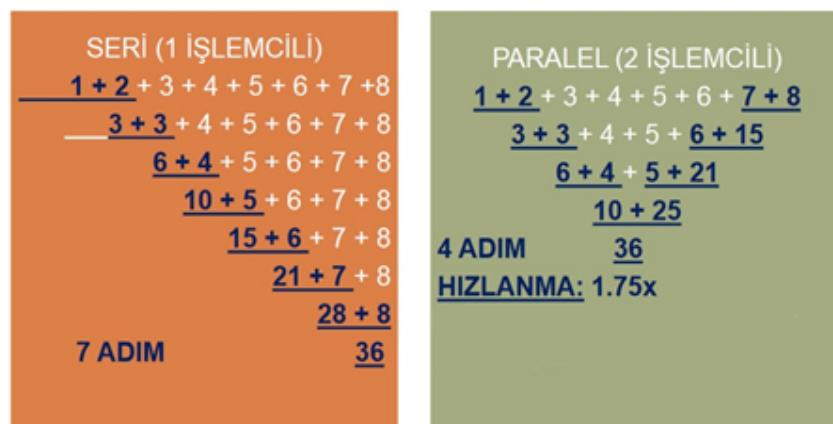
Paralel hesaplamanın avantajları:

- **Zaman:** Büyük problemleri kısa sürede çözebilmeyi sağlar
 - **Performans:** Eğer problemin çözümü için geliştirilen algoritma paralelliğe uygunsa en iyi performans bu yolla elde edilecektir.
- şeklinde sıralanabilir.

3.4.1 Paralel İşlem Örneği

Fikir: Problemlerin farklı kısımlarının farklı işlemcilere verilmesi

Beklenti: n adet işlemci ile n kat hızlanma



Şekil 3-22: Paralel Hesaplama Yöntemi ile Toplama İşlemi

Şekil 3-6'da iki adet işlemci kullanılan toplama işlemi probleminin paralel hesaplama yöntemi ile ve seri hesaplama ile çözümleri görülmektedir. 2 adet işlemci ile hızlanma 1.75 kat olmuştur. Beklenti 2 adet işlemci ile 2 kat hızlanma idi. Daha fazla sayıda işlemci kullanıldığında bekleniyeye daha çok yaklaşılacaktır.

3.5 HETEROJEN HESAPLAMA

Heterojen hesaplama, bilgisayarda bulunan iki temel işlemci; GİB ve MİB'in özelliklerinden faydalananmak amacıyla harmanlanmış bir çözüm yoludur. MİB, seri işlemlerin uygulanabileceği rasgele erişimli bellek gerektiren uygulamalarda çok iyi sonuç verir. GİB ise floating point(kayar nokta) işlemlerinde ve paralelleştirilebilen işlemlerde etkilidir. İki matrisin çarpımı örneğini düşünürsek, matris elemanlarını üretme ve ekrana yazdırma gibi işlemleri MİB ve matrislerin çarpımı işlemini GİB üzerinde yaparsak en etkili sonucu elde ederiz. Sonuç olarak heterojen hesaplama, eldeki problem için doğru işlemciyi seçerek doğru sonuca ulaşmak anlamına gelmektedir.

Seri ve paralel hesaplamlarda bilgisayarın Merkezi İşlem Birimi(CPU) kullanılmaktaydı. Heterojen hesaplamanın en önemli kısmı işin içine Grafik İşlem Birimini de dahil etmesidir. Projede kullanılan CUDA mimarisi heterojen hesaplama modelindedir.

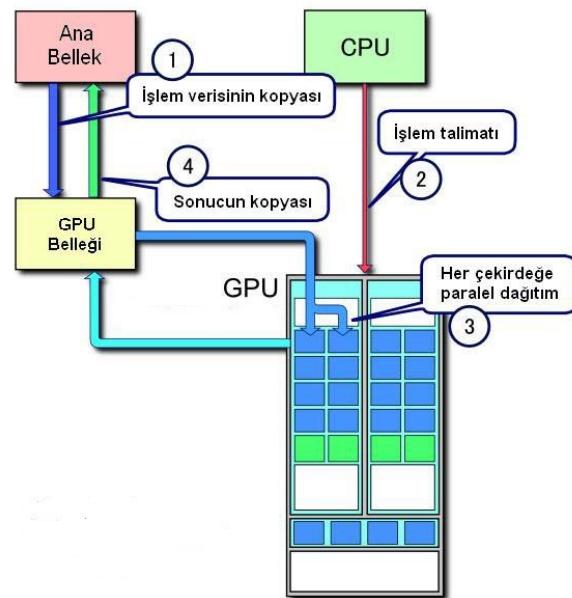
BÖLÜM 4

CUDA

Bu bölümde daha önceden tanımı yapılmış olan CUDA mimarisi detaylarıyla incelenmiştir. Ayrıca CUDA platformunun daha iyi anlaşılabilmesi için örnek uygulamalar verilmiştir.

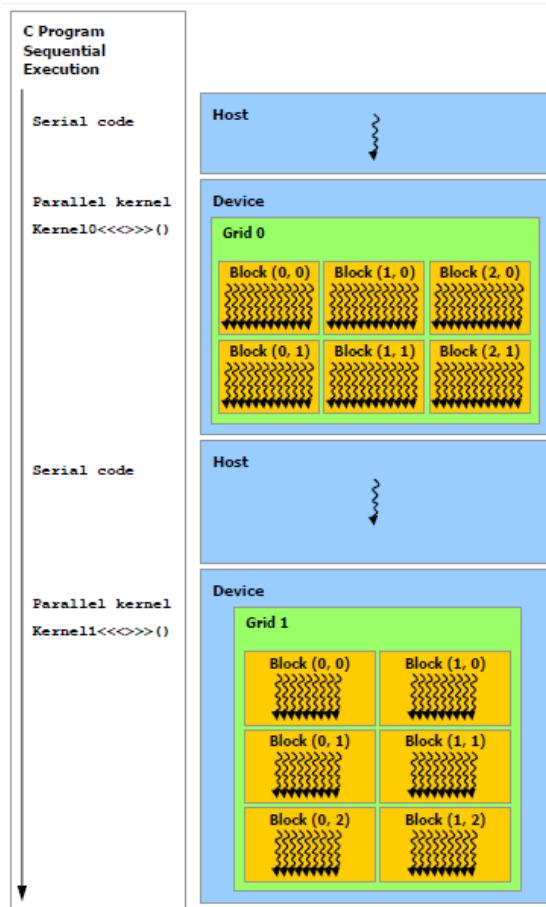
4.1 İŞ AKIŞI

CUDA mimarisinde geliştirilen uygulamalar sadece GİB üzerinde çalışmazlar. Öncelikle MİB tarafından kontrol edilen ana bellek üzerinden grafik kartı üzerindeki belleğe kopyalanması gereklidir. GİB belleğindeki veri CUDA iş parçacıkları tarafından yürütülerek paralel olarak hesaplanması tamamlanır ve ardından tekrar ana belleğe gönderilerek işlem sonlandırılır. Bu akış modeli Şekil4-1'de görülmektedir.



Şekil 4-23: CUDA iş akış modeli[39]

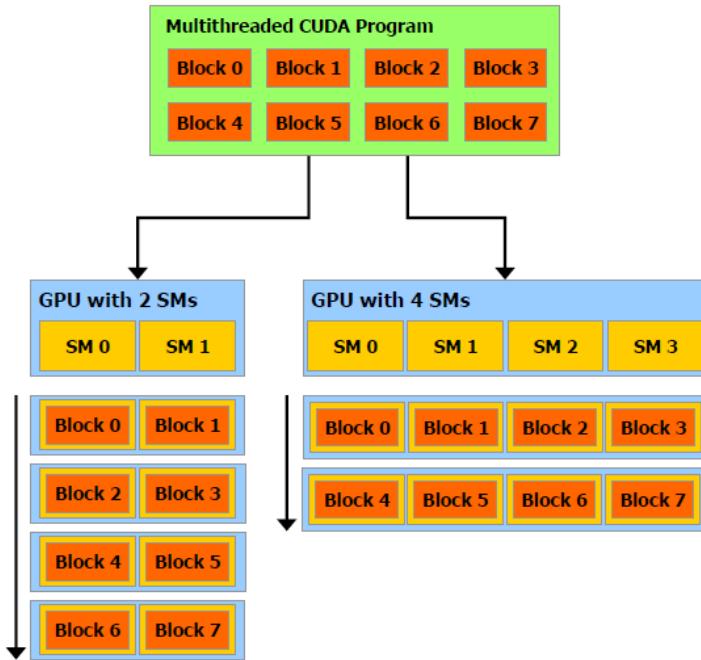
MİB üzerinde çalışan kod parçaları ile GİB üzerinde çalışan kod parçaları farklıdır. CUDA mimarisi ile uygulama geliştirilirken bu yapıya uygun bir şekilde kod yazılır. Şekil 4-2'de CUDA ile geliştirilmiş bir programın genel çalışma mantığı ifade edilmektedir. Burada "Host" MİB olarak ve "Device" ise; GİB olarak düşünülmelidir. Seri kodlar MİB üzerinde yürütülür. "Kernel" olarak adlandırılan paralel kod parçaları ise GİB üzerinde yürütülür. Host üzerinde çalıştırılan komut tektil ve aynı anda tek bir thread(iş parçası) işlenir. Device üzerinde çalıştırılan komutlar ise birden fazla thread'e bölünerek işlenir.



Şekil 4-24: CUDA ile geliştirilen bir programın MİB(Host) ve GİB(Device) üzerinde yürütülmesi[40]

4.2 ÖLÇEKLENEBİLİR PROGRAMLAMA

Gelişen GİB teknolojisi ile birlikte donanımsal farklılıklar da ortaya çıkmaktadır. CUDA mimarisi, NVIDIA'nın Geforce 8800 ve sonrasında çıkan modellerinde desteklenmektedir. Piyasaya çıkan her yeni ürün öncekilerden farklı çekirdek sayılarına sahiptir. Ancak CUDA ile yazılmış bir uygulama , GİB çekirdekleri ile ilgili bir işlem yapmak zorunda değildir. Bu özelliğe GİB'in ölçülebilirlik özelliği denilmektedir. Şekil 4-3'te çift çekirdekli bir GİB ile 4 çekirdekli bir GİB'in programa nasıl ayak uydurdukları görülmektedir.



Şekil 4-25: Çift çekirdekli GİB ile 4 çekirdekli GİB'in Gelen Programa Göre Kendiliğinden Ölçeklenmesi[40]

4.3 CUDA PROGRAMLAMA MİMARİSİ

4.3.1 Kernel

CUDA'da geliştirilen bir kodun GİB tarafından çalışacak olan kısmına “kernel” adı verilmektedir. GİB, veri kümesinin her bir elemanı için birer kernel kopyası oluşturulur. Bu kernel kopyalarına “thread” adı verilmektedir. Kernel kodu Host tarafından çağrılır ve Device üzerinde yürütülür. Kernel kodu “`__global__`” ile nitelendirilir. Kernel, thread dizilimleri(thread arrays) tarafından çalıştırılır.[40]

4.3.2 Thread

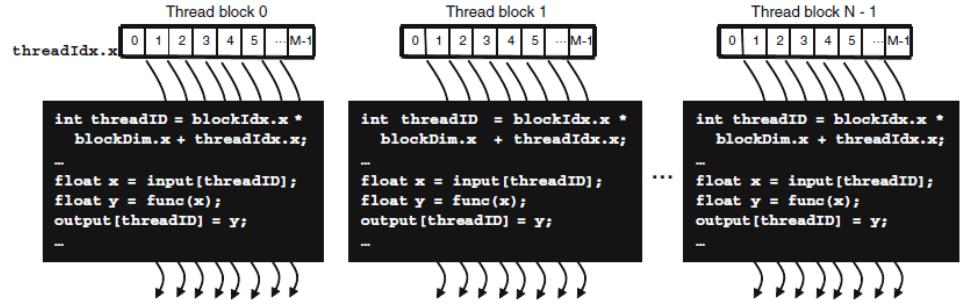
Thread, CUDA mimarisindeki en küçük programlama parçasıdır. Block'lar içerisinde 1B'lu, 2B'lu veya 3B'lu olabilirler. Kendi aralarında eş zamanlı olarak aynı kod parçasını çalıştırmaktadır. Thread'ler Block'lar içerisinde dizilerek gruplanırlar. Farklı Block'lardaki Thread'ler ortak çalışmazlar.[40]

Her bir thread'in kendine ait Program Sayacı(PC), kaydedicisi(register) ve durum kaydedicisi(State Register) vardır.[41]

Her thread'in Block içerisinde kendine ait bir ID(kimlik, indis)'si vardır. Bu indisler; “threadIdx.x”, “threadIdx.y” ve “threadIdx.z” şeklinde ifade edilmektedir.

4.3.3 Block

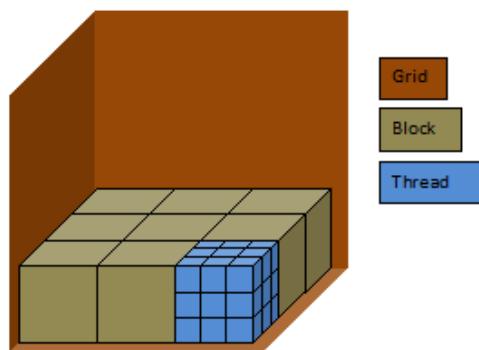
Block yapısı, paralel olarak çalışan thread'lerden oluşmaktadır. Grid içerisinde tektirler. Grid içerisinde 1B'lu, 2B'lu veya 3B'lu olabilirler. Grid'ler içerisinde dizilerek gruplanırlar. Her bir Block'un , Grid içerisinde kendine ait bir indisı vardır. Bu indisler “blockIdx.x”, “blockIdx.y” ve “blockIdx.z” şeklindedir. İçerisinde barındırdığı thread satır ve sütun sayısına göre boyutlanırlar ve bu boyutlar “blockDim.x”, “blockDim.y” , “blockDim.z” terimleri ile ifade edilmektedir. “blockDim” terimi “threadsPerBlock” terimi ile denktir ve her bir block'taki thread anlamına gelmektedir.



Şekil 4-26: Thread Block'lari[42]

4.3.4 Grid

Grid, Block'ların bir araya gelerek oluşturdukları yapılardır. Her bir Kernel çağrısı bir Grid oluşturur. Yani paralel olarak çalıştırılan kod parçası, kernel çağrısı ile Device üzerinde çalıştırılırken her bir kopyası için bir Grid oluşturulmaktadır. Grid'ler 1B'lu veya 2B'lu olabilirler. Hesaplama kapasitesi 2.0 ve yukarısı sürümlerde gridler 3B'lu da olmaktadır. Bu boyutlar “gridDim.x” , “gridDim.y” ve “gridDim.z” şeklinde ifade edilmektedir. “gridDim” terimi ile “blocksPerGrid” terimi denktirler.[40] Şekil 4-5'te thread hiyerşisi görülmektedir. Her bir Grid Block'lardan, her bir Block ise Thread'lerden oluşmaktadır.

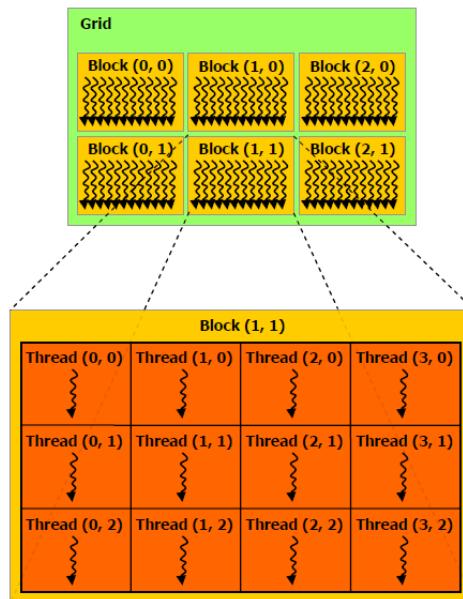


Şekil 4-27: Thread hiyerarşisi[43]

Şekil 4-6'daki 2x2'lik Block ve 4x3'lük Thread yapısı göz önüne alınırsa:

- $\text{gridDim.x} = 3$ (block'ların sütun sayısı kadar)
- $\text{gridDim.y} = 2$ (block'ların satır sayısı kadar)
- $\text{blockDim.x} = 4$ (thread'lerin sütun sayısı kadar)
- $\text{blockDim.y} = 3$ (thread'lerin satır sayısı kadar)

verileri elde edilir.

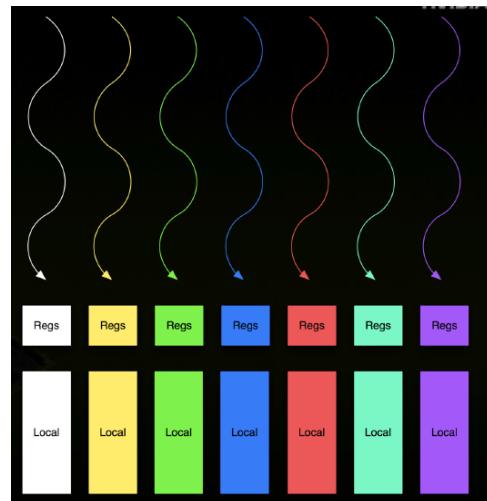


Şekil 4-28: Thread-Block-Grid İlişkisi[40]

4.4 CUDA HAFIZA ORGANİZASYONU

4.4.1 Yerel Bellek (Local Memory)

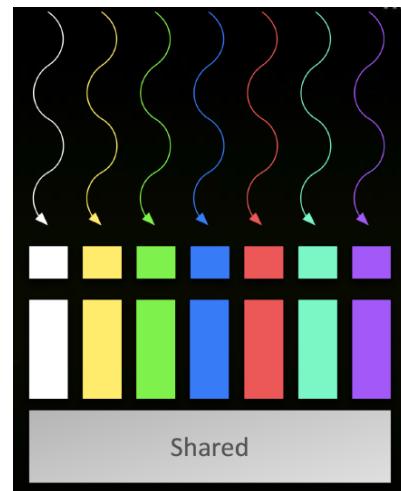
Program çalışırken kullanıldığından kullanıcıya açık bir bellek yapısı değildir. Buradaki veriler, sahip olunan her bir thread'in çalışması bitene kadar saklanır. Her local bellek yapısı, ait olduğu thread'e özeldir. Diğer thread'ler tarafından ulaşılamazlar. Kendi thread'leri tarafından üzerine veri yazılabilir ve üzerinden veri okunabilir.



Şekil 4-29:Her bir thread'in kendine özel Register ve Local Memory alanları mevcuttur [44].

4.4.2 Paylaşımlı Bellek (Shared Memory)

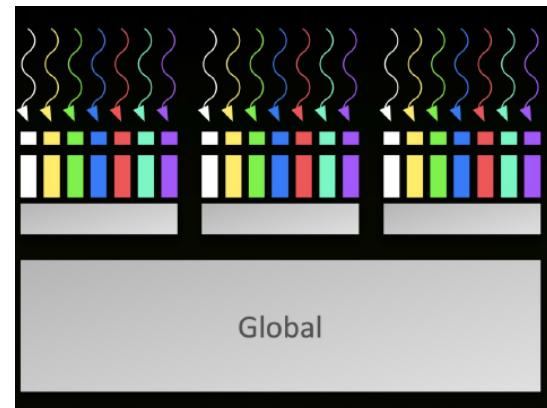
Grafik İşlem Birimi'nin sahip olduğu tüm çekirdek kümelerinde bulunmaktadır. Yalnızca aynı block'ta bulunan thread'ler bu belleğe veri yazıp okuyabilirler. Aynı block içerisindeki thread'ler çakışmadığı sürece çok hızlı çalışmaktadır.



Şekil 4-30: Her bir Thread Block'una özel Shared Memory alanı mevcuttur.[44]

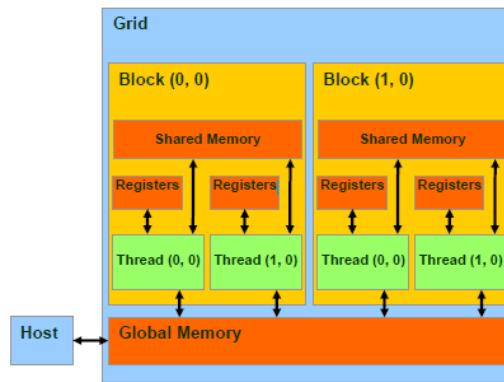
4.4.3 Genel Bellek (Global Memory)

Çalışan bütün thread'ler tarafından erişilebilen , üzerinden veri okunabilen ve üzerine veri yazılabilen bir bellek türüdür. Bu belleğin bant genişliği düşüktür.



Şekil 4-31:Tüm block'ların kullanıldığı memory alanı Global Memory'dir.[44]

Global bellek, Host ile Device arasındaki temel haberleşme birimidir. Uzun erişme süreleri vardır. Şekil 4-10'da global bellek, paylaşımı bellek ve her bir thread'e ait olan kaydedici hafıza alanları görülmektedir. Ok yönleri , ilgili bellek gözlerinden veri okuyup-yazma izinlerini belirtmektedir.



Şekil 4-32:CUDA bellek modeli[40]

4.4.4 Sabit Bellek (Constant Memory)[45][46]

Read-only(üzerinden sadece okuma yapabilen)bir bellek modelidir. Kernel çalıştırıldığından değişimmeyecek veriler için kullanılması avantaj sağlar. 64 KB'lık büyülüğe sahiptir. NVIDIA'nın değişmez bellek modelini sunmasındaki amaç bellek bant genişliğinin global memory'ye göre daha az olmasınadır. Eğer geliştirilen uygulamada veriler sadece okunacaksa ve yazılmayacaksa bu bellek modeli kullanılmalıdır. Değişmez bellekten dinamik olarak yer alınmasına izin verilmemektedir. Değişmez belleğe yalnızca MİB fonksiyonları tarafından veri yazılabilir.

Değişmez bellek üzerine veri kopyalamak için cudaMemcpyToSymbol() fonksiyonu kullanılır. cudaMemcpy() fonksiyonundan farkı, cudaMemcpy() fonksiyonun global bellek üzerine kopyalama yapması ve cudaMemcpyToSymbol() fonksiyonunun değişmez bellek üzerine veri kopyalamasıdır.

4.4.4.1 Constant Memory Kullanılması Tavsiye Edilen Durumlar

- Giriş verimizin “execution” süresince değimeyeceğini bildiğimiz durumlarda
- Tüm thread’lerimiz memory alanının aynı adres alanından veriye erişecekse. Örneğin; bir thread block’undaki tüm threadler aynı hafıza adres alanını işaret ediyorsa

4.4.4.2 Constant Memory Kullanılması Tavsiye Edilmeyen Durumlar

- Giriş verimizin “execution” süresince değişeceğini biliyorsak
- Tüm thread’lerimiz memory alanının aynı adres alanından veriye erişmeyeceksé
- Verimiz Read-only değilse

4.4.4.3 Constant Memory Performans Değerlendirmesi

64 KB'lık constant memory'den okumanın , global memory'den okumaya göre 2 avantajı vardır:

- 1) Constant memory'den 1 okuma ile yakınlardaki(nearby) thread'lere broadcast yapılabilmesi ile 15 adet okumadan tasarruf edilir.
- 2) Constant memory "cached" bir hafıza alanıdır, bu yüzden aynı adresden okumalar herhangi bir ek bellek trafiğine maruz kalmayacaktır.
- 3) Ön belleğe sahip olduğu için global bellekten daha hızlıdır.

4.4.5 Doku Bellek (Texture Memory)

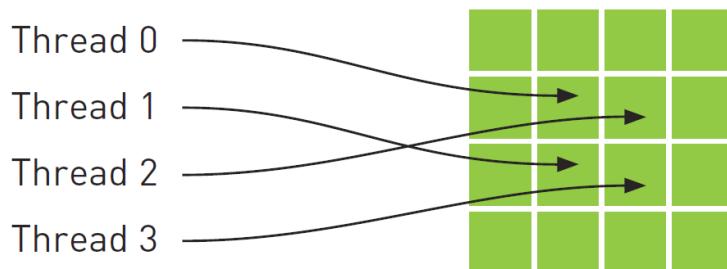
Doku belleği de değişmez bellek gibi read-only yapıya sahiptir. Bu yapı performansı artırır ve bellek trafiğini azaltır.

NVIDIA, texture birimini klasik OpenGL ve DirectX rendering iş hattı için tasarlamıştır. Texture Memory bunu kullanışlı kılmak için bazı özelliklere sahiptir.

Constant memory gibi, texture memory de chip üzerinde önbellekte(cached)dir. Bu yapı bazı durumlarda DRAM'de daha az talep trafiği ile daha etkin düzeyde

bantgenişliği olanağı sunar. Özellikle, doku önbellekleri grafik uygulamaları için tasarlanmıştır. Spatial Locality ile ilişkili bellek yapısına sahiptir.

Spatial Locality: “Belleğe bir başvuru yapıldıktan sonra, büyük olasılıkla bir sonraki başvuru yakın bir adrese olacaktır” mantığına dayalı Locality of Reference tipidir.



Şekil 4-33:Spatial Locality mantığına göre dizilmiş Texture Memory yapısı[47]

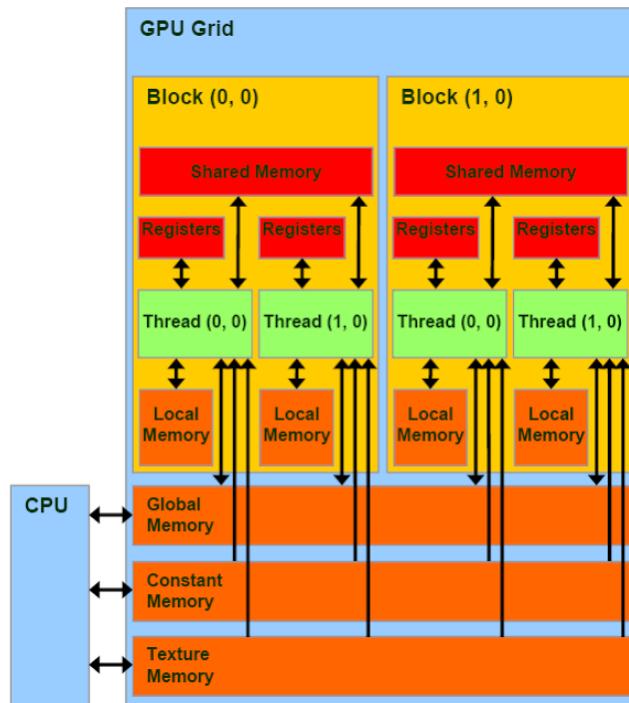
NOT: Şekil 4-11'de görülen thread'ler ardışık değildir, bu yüzden MİB mantığında önbellek üzerinde birlikte yer almazlar. Ama GİB mantığına göre thread'ler doku(texture)önbelleği üzerinde birlikte dizilirler. Bu kullanımın, global bellek kullanımına göre performansı daha da artıracağı görülebilir.

Doku bellek 2 boyutlu olarak ön belleklenmiştir, bu yüzden aynı warp'ta yer alan thread'lerin doku adresleri okuması sonucu daha iyi performans elde edilir.

Texture Fetch: Bir texture'ı okuma işlemine texture fetch adı verilmektedir.

Texture Reference: Texture fetch’ın belirlediği nesnenin ilk parametresi “texture reference” olarak adlandırılır.

Doku bellek üzerinde 3 boyuta kadar yerleşim görülebilir. Doku bellek üzerindeki dizilerin elemanlarına “texels(texture elements)” adı verilmektedir.



Şekil 4-34: CUDA Bellek Modelleri[40]

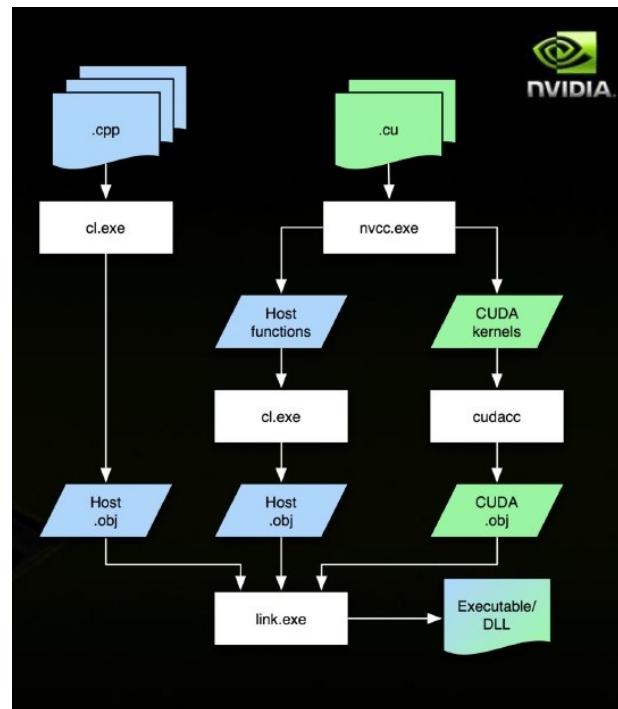
Şekil 4-12’de görülmekte olan CUDA bellek modelleri farklı özelliklere sahiptir. Kullanım amacına uygun olan model uygulama geliştirilirken doğru bir şekilde seçilmelidir. Ok yönlerine bakıldığında eğer tek yönlü ok varsa sadece okuma yapıldığı, eğer çift yönlü ok var ise hem okuma hem de yazma yapıldığı anlaşılmalıdır.

4.5 CUDA PROGRAM YÜRÜTME İŞLEMLERİ

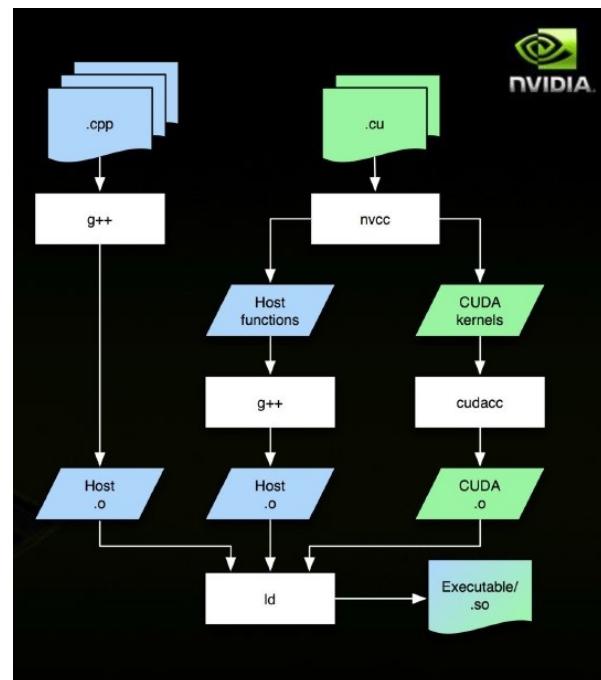
CUDA mimarisi ile geliştirilen uygulamanın tamamı sadece GİB ya da sadece MİB üzerinde çalıştırılmaz. NVCC derleyici sürücüsü “.cu” uzantılı kaynak kod dosyasını MİB kodu ve PTX(GİB) kodu olmak üzere iki ayrı çıkış dosyasına çevirir. C kodları (eğer uygulama C dili ile geliştirildiyse) standart C derleyicisi tarafından ve PTX kodları CUDA tarafından derlenir.

NVCC bir derleyici değil, bir derleyici sürücüsüdür. Kodun ayrıştırılması için gerekli araçları ve derleyicileri çağrırmakla görevlidir.

CUDA mimarisinde geliştirilmek istenen uygulamaların Windows, Linux ve Mac OSx platformlarında çalıştırılabildeği daha önce ifade edilmiştir. Şekil 4-13 ve Şekil 4-14’te CUDA kodlarının Windows ve Linux ortamlarında çalıştırılma aşamaları gösterilmektedir.



Şekil 4-35: Windows ortamında CUDA kodlarının derlenme ve yürütülme aşamaları[44]



Şekil 4-36:Linux ortamında CUDA kodlarının derlenme ve yürütülme aşamaları[44]

4.6 BELLEK ERİŞİM FONKSİYON NİTELEYİCİLERİ

4.6.1 __global__

Kernel kodu , __global__ ile nitelendirilir. main() içerisinde kernel çağrıları yapılırken kernel fonksiyon adı yazılır ve “<<< >>>” açılı parantezleri yazılır. Bu yazım şekli CUDA'ya özgüdür. Açılı parantezler arasında yazılan ifadeler sırasıyla blocksPerGrid(her bir grid'teki block sayısı) ve threadsPerBlock(her bir block'taki thread sayısı) bilgilerini göstermektedir.Kısacası bir kernel çağrısı:

Kernel_fonksiyon_adı<<<blocksPerGrid,threadsPerBlock>>>(fonk_alacağı_parametreler);

şeklindedir.

__global__ ile nitelenmiş fonksiyonunun dönüş tipi void olmalıdır. Bu fonksiyon MİB tarafından çağrılabılır, GİB tarafından çağrılamaz. Başka bir __global__ fonksiyonu tarafından da çağrılamaz. GİB üzerinde işlem正在执行中[47]

“__global__” ile nitelenmiş fonksiyonlar asenkron olarak çalışırlar. Yani aygit(GİB) üzerinde çalışması bitmeden döner.

4.6.2 __device__

GİB üzerinde çalışırlar ve MİB tarafından çağrılamazlar. Yalnızca GİB tarafından çağrılabılır.[40]

4.6.3 __host__

Nitelediği fonksiyonun sadece MİB tarafından çağrılabildiğini ve sadece MİB üzerinde çalıştırılabilğini ifade etmektedir. Varsayılan fonksiyon niteleyicisi “host” olarak belirlenmiştir. Yani hiçbir niteleyici yazılmasa bile direkt olarak “host” niteleyicisi olduğu varsayılmaktadır. Bazı durumlarda “device” niteleyicisi ile birlikte kullanımları da vardır. “global” niteleyicisi ile birlikte kullanımı sözkonusu değildir. [40]

4.7 DEĞİŞKEN NİTELEYİCİLERİ

4.7.1 __device__

GİB üzerinde konumlandırılmış bir değişkeni niteler. Bir uygulamada yaşam süresi bellidir. Global bellek üzerinde yer almaktadır. Tüm thread’ler tarafından yürütülebilir.[40]

4.7.2 constant

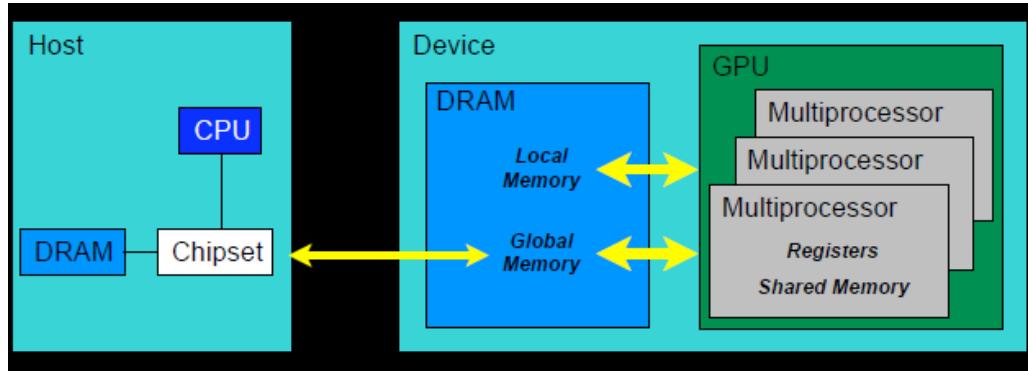
Seçime bağlı olarak “device” niteleyicisi ile birlikte kullanılabilir. Değişmez bellek üzerinde yer alan, belli bir yaşam süresi olan uygulamalarda yer alan ve tüm thread’ler tarafından erişilebilen değişkenleri tanımlamaya olanak sağlar.[40]

4.7.3 shared

Seçime bağlı olarak “device” niteleyicisi ile birlikte kullanılabilir. Paylaşımlı bellek üzerinde bir thread block’unda yer alan, belli bir yaşam süresi olan block içerisinde bulunan ve sadece aynı block üzerinde yer alan thread’lerin erişebildiği değişkenleri nitelemektedir.[40]

4.8 BELLEK YÖNETİMİ

Bir CUDA uygulamasında MİB ve GİB ayrı bellek alanlarını kullanırlar. Programın çalıştırılması sırasında bu bellek alışverişinin sağlanması için fonksiyonlar geliştirilmiştir. Şekil 4-15’te “Host(MİB)” ile “Device (GİB)” arasındaki veri aktarımı görülmektedir.



Şekil 4-37: Host ile Device Arasında Veri Aktarımı[48]

4.8.1 Hafıza Ayırma Fonksiyonları

1. **cudaMalloc (void ** pointer, size_t nbytes);**

C dilindeki Malloc ile eşdeğerdir, ancak hafıza GİB üzerinde ayılır .

2. **cudaMemset (void * pointer, int value, size_t count);**

İstenilen değer, istenildiği kadar bellek alanına yerleştirilebilir.

3. **cudaFree (void* pointer);**

C dilindeki Free ile eşdeğerdir, ancak burada GİB üzerinden ayrılan hafıza serbest bırakılır.

4.8.2 Veri Kopyalama Fonksiyonu

1. **cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind_direction);**

Burada “dst” ile kopyalanacak verinin hedefi, “src” ile kopyalanacak verinin kaynağı, “nbytes” ile kopyalanacak verinin boyutu ve “cudaMemcpyKind_direction” ile kopyalama yönü belirtilmektedir.

“enum” veri tipi ile ifade edilen değişkenler daha önceden belirlenmiş kopyalama yönleridir. Buna göre veri alışverişinin yönü MİB’den GİB’e, GİB’den MİB’e ya da

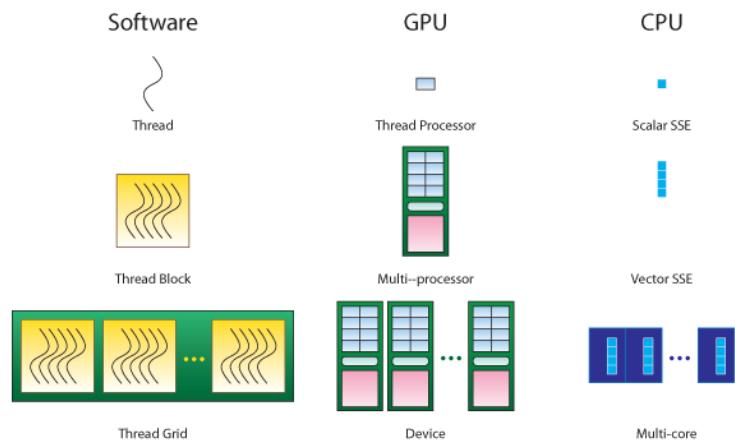
GİB'in kendi içerisinde olmaktadır. MİB'den MİB'e bir yön bulunmamaktadır çünkü bu durumda uygulama heterojen değil klasik bir MİB uygulaması olmaktadır.

- cudaMemcpyHostToDevice (MİB → GİB)
- cudaMemcpyDeviceToHost (GİB → MİB)
- cudaMemcpyDeviceToDevice (GİB → GİB)

4.9 HESAPLAMA KAPASİTESİ

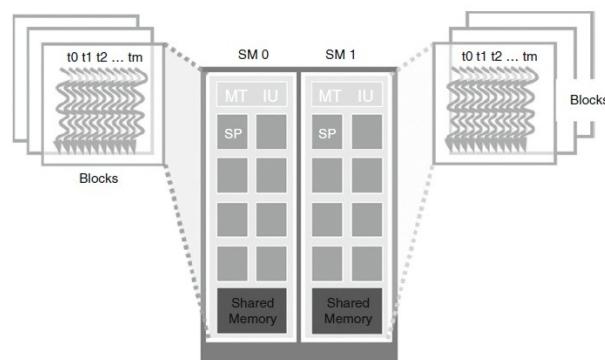
Bir GİB'in hesaplama kapasitesi "major version number(birincil versiyon numarası)" ve "minor version number(ikincil versiyon numarası)" ile ifade edilmektedir. Birincil versiyon numaraları aynı olan ürünler aynı fiziksel çekirdek mimarisindedirler. Birincil versiyon numarası 1 olan aygıtlar Tesla mimarisine, 2 olan aygıtlar Fermi mimarisine ve 3 olan aygıtlar Kepler mimarisine sahiptirler. İkincil versiyon numarası ise mimari yenilendikçe değişir. [40]

CUDA mimarisine donanımsal olarak bakıldığından her bir thread'in "Streaming Processor(Akış İşlemcisi)" üzerinde çalıştığı görülmektedir (Şekil 4-16). Sekiz adet akış işlemcisinin bir araya gelerek oluşturduğu yapıya "Streaming Multiprocessor(Çoklu-İşlemci)" adı verilmektedir.



Şekil 4-38: Thread'lerin yazılımsal ve donanımsal yapıları[49]

SM'ler ise ekran kartı donanımını oluşturmaktadır . Örneğin, nVIDIA GT200 grafik kartı 30 adet SM'ye sahiptir.Bunlardan 2 tanesi Şekil 4-17'de görülmektedir. Her bir SM, 8 block'a kadar görevlendirilebilir şekilde dizayn edilmiştir. GT200 işlemcide 30 SM ile 240 block'a kadar eş zamanlı görevlendirme olabilir. Çoğu grid 240 block'tan fazlasını içermektedir. GT200'de her bir SM 1024 thread'e kadar görevlendirilebilecek şekilde yapılandırılmıştır.Bu yapıya 4 block-256 thread veya 8 block-128 thread grupları örnek olarak verilebilir. Ancak 16 block-64 thread olmaz çünkü , her bir SM sadece 8 blocka kadar yerleştirilebilir.[42]



Şekil 4-39: Thread'lerin üzerinde çalıştığı SP ve SM yapıları[42]

CUDA'da uygulamalar GİB'in karakteristiğine göre geliştirilmelidir. Bunun için birincil ve ikincil versiyon numaraları ile birlikte teknik özelliklerini verilen Tablo 4-1'den yararlanılabilir.

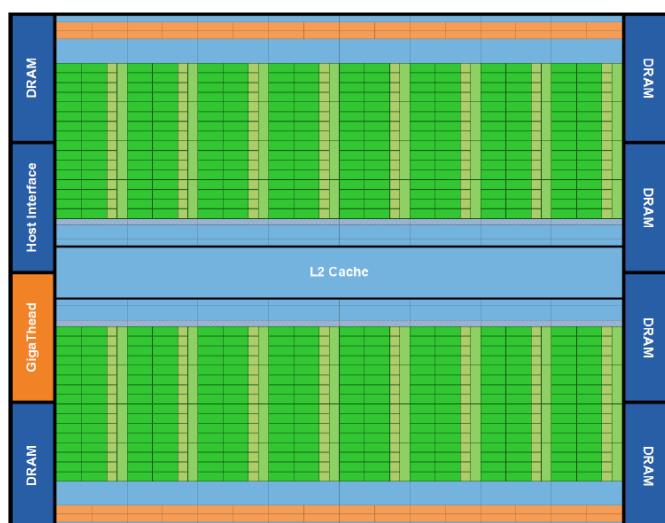
Çizelge 4-1: Teknik Özellikler ve GİB'in Versiyon Numaraları[45]

Teknik Özellikler	1.0	1.1	1.2	1.3	2.x	3.0	3.5
Maksimum grid boyutu		2			3		
Bir grid'in x-boyutunda içerebileceği en fazla thread sayısı			65535		$2^{31}-1$		
Bir grid'in y- ya da z-boyutunda içerebileceği en fazla thread sayısı				65535			
Maksimum block boyutu					3		
Bir block'un x-ya da y- boyutunda sahip olabileceği en fazla thread sayısı		512			1024		
Bir block'un z- boyutunda sahip olabileceği en fazla thread sayısı			64				
Bir block'ta yer alabilecek maksimum thread sayısı	512			1024			
Warp (planlanan kanal) boyutu			32				
Çoklu işlemci başına en fazla block sayısı		8			16		
Çoklu işlemci başına en fazla warp sayısı	24	32	48	64			

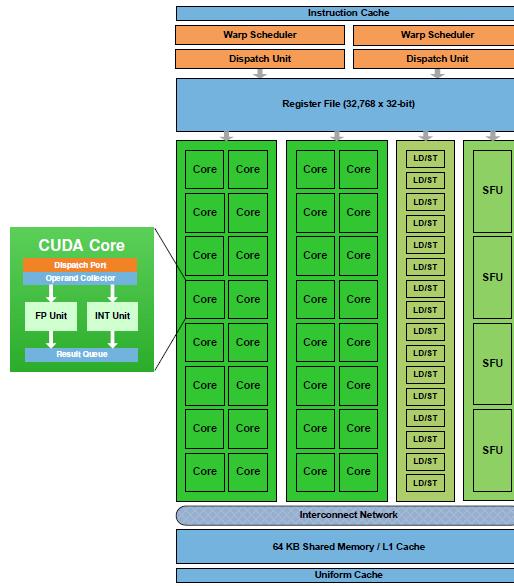
Çoklu işlemci başına en fazla thread sayısı	768	1024	1536	2048
---	-----	------	------	------

4.9.1 Fermi Mimarisi[50]

Fermi mimarisi tabanlı ilk GİB 3 milyar transistöre ve 512 adet CUDA çekirdeğine sahipti. Bu 512 adet CUDA çekirdeğinin her biri 16 adet SM(çoklu- işlemci)'den meydana gelmiştir ve her bir SM'de 32 adet fiziksel çekirdek(core) bulunmaktadır. Şekil 4-18'de Fermi mimarisinin genel yapısı ve Şekil 4-19'da her bir Fermi SM'sinin yapısı görülmektedir.



Şekil 4-40:Fermi mimarisi genel yapısı



Şekil 4-41: Bir Fermi SM yapısı

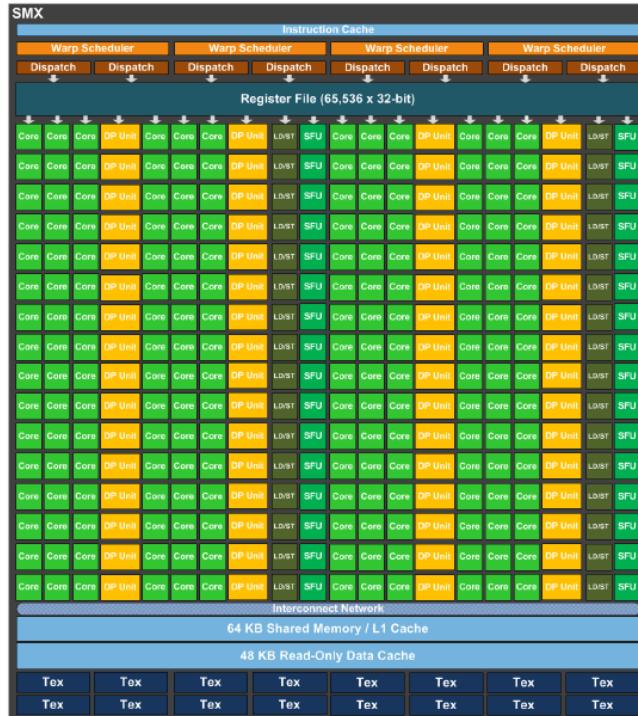
4.9.2 Kepler Mimarisi[51]

Fermi mimarisine ek olarak bir çok yeni özelliği barındırmaktadır. Bunlar:

- Gelişmiş SM'ler(SMX)
- Dinamik Paralellik
- Hyper-Q Teknolojisi



Şekil 4-42: Kepler Mimarisi Genel Yapısı



Şekil 4-43: Kepler Mimarısında bir SMX'nin iç yapısı

4.9.3 Fermi ile Kepler Mimarilerinin Teknik Karşılaştırması

Çizelge 4-2: Fermi ile Kepler Mimarilerinin Teknik Karşılaştırması[51]

	Fermi GF10 0	Fermi GF10 4	Kepler GK104	Kepler GK110
Hesaplama Kapasitesi	2.0	2.1	3.0	3.5
Her bir Warp içinde yer alan Thread sayısı(Thread/Warp)	32	32	32	32
Maksimum [Warp/Multiprocessor] sayısı	48	48	64	64
Maksimum [Thread/Multiprocessor] sayısı	1536	1536	2048	2048
Maksimum [(Thread Block'u)/Multiprocessor] sayısı	8	8	16	16
[32-Bit Register/Multiprocessor] sayısı	32768	32768	65536	65536
Maksimum [Register/Thread] sayısı	63	63	63	255

Maksimum [Thread/Thread Block'u] sayısı	1024	1024	1024	1024
Hyper-Q	Yok	Yok	Yok	Var
Dinamik Paralellik	Yok	Yok	Yok	Var

4.9.4 Warp Yapısı

CUDA mimarisinde bir grid tanımlandığında thread'lerden “Warp Scheduler(Warp Zamanlayıcı)” tarafından 32'li gruplar oluşturulur ve bu gruplar işlenecekleri yer olan SP'ye gönderilmektedir. SP'ye gelen thread'ler işlenmek üzere çekirdeklere dağılırlar. Warp üzerindeki her bir thread farklı verinin aynı komutunu yürütmektedir.

Özetle:

- Grid'ler \square GİB ile eşleşir.
- Block'lar \square MultiProcessor(MP) ler ile eşleşir.
- Thread'ler \square Stream Processor(SP)ler ile eşleşir.
- Warp'lar \square Eşzamanlı yürütülen 32'lik thread blocklarıdır.

4.10 PERFORMANS ÖLÇÜMÜ[52]

Kernel yürütme süresi “bandwidth(bantgenişliği)” ile ölçülmektedir. Etkili bir değerlendirme yapılabilmesi için “Teorik Bantgenişliği” ve “Etkin Bantgenişliği” ölçümlerinin yapılması gerekmektedir.

4.10.1 Teorik Bantgenişliği

Teorik bantgenişliği, ürünün literatüründe belirtilen donanımsal özelliklere göre hesaplanır. Örneğin, NVIDIA Tesla M2050 GİB, DDR(Double Data Rate) RAM kullanır. Bir bellek saat oranı(memory clock rate) 1,546 MHz ve 384-bit genişliğinde

bellek arayüzüne sahiptir. Bu veriler ile teorik bantgenişliği hesaplaması Şekil 4-22'de ifade edilmiştir.

$$BW_{\text{Theoretical}} = 1546 * 10^6 * (384/8) * 2 / 10^9 = 148 \text{ GB/s}$$

Şekil 4-44:Teorik Bantgenişliği Örneği

4.10.2 Etkin Bantgenişliği

Etkin bantgenişliği “ $BW_{\text{Effective}} = (R_B + W_B) / (t * 10^9)$ ” formülü ile hesaplanmaktadır. Burada “ R_B ” ile her kernel’den okunan byte sayısı ifade edilmektedir. “ W_B ” ile her bir kernel’e yazılan byte miktarı gösterilmektedir. “ t ” saniye cinsinden geçen zamanı ifade etmektedir. Örneğin, N adet float sayının kopyalanması işleminin etkin bantgenişliği hesaplanacak olursa:

$(R_B + W_B)$ =her float sayı için $(1+1)=2$ olarak formülde yerine yazılır. Float sayı kopyalanacağı için $N*4$ çarpımı, $(R_B + W_B)$ ile çarpılarak hesaplanır.

CUDA’daki etkin bantgenişliği hesaplamaları “Event API” ile yapılmaktadır.

4.10.3 CUDA Event API

4.10.3.1 Fonksiyonları[47]

- **cudaEventCreate:** *Event nesnesi yaratır.*
- **cudaEventCreateWithFlags:** *Belirlenmiş bayraklar ile event nesnesi oluşturur.*
- **cudaEventDestroy:** *Bir event nesnesini yok eder.*
- **cudaEventElapsedTime :** *Event'ler arasındaki geçen zamanı hesaplar.*
- **cudaEventQuery :** *Bir event'in durumunu sorgular.*
- **cudaEventRecord :** *Bir event'i kaydeder.*
- **cudaEventSynchronize :** *Bir event'in tamamlanmasını bekler.*

4.10.3.2 Kullanım Senaryoları

- CUDA çağrısı için geçen süre ölçülmesi (clock cycle precision)
- Asenkron CUDA çağrısının durum sorgulaması

4.11 CUDA-OPENGL İŞBİRLİĞİ

4.11.1 OpenGL

OpenGL (Open Graphics Library), 2B'lu ve 3B'lu grafikleri bilgisayar ekrarına çizdirmek için kullanılan bir API'dir. Donanımsal olarak NVIDIA, AMD/ATI, SGI, HP, Microsoft gibi firmalar tarafından desteklenmekle birlikte yazılımsal olarak da Windows, Linux, Mac Osx gibi ortamlarda çalışabilme özelliklerine sahiptir. OpenGL, yukarıda ismi yazılan firmaların üyesi olduğu ARB(Architectural Review Board) tarafından kontrol edilmektedir. Bu kurulun (ARB) amacı:

- OpenGL API'sinin kararlı kalmasını sağlamak
- Yeni donanımlara ayak uydurabilmesi için belirli standartları sağlamak
- Platformlara yapılan eklemelerin mümkün kılınmasını sağlamak

olarak sıralanabilir.[53]

4.11.2 CUDA ile OpenGL Etkileşiminin Sağlanması İçin Gerekli İşlemler

CUDA içeriği ile OpenGL içeriği başlangıçta etkileşim içinde değildir. Uygulama geliştiren kişi bu etkileşimi kendisi sağlamalarıdır. Etkileşim için gereken işlem basamakları:

1. Gerekli kütüphane dosyaları uygulama kodlarına eklenir.
2. “cudaGLSetGLDevice” fonksiyon çağrıları ile bağlantı sağlanacak “Device” seçilir.
3. Uygun görülen tampon nesnesi oluşturulur.
4. CUDA kaynağı belirlenir.
5. Kaynak ile hedef tampon arasındaki kayıtlar gerçekleştirilir.
6. Kaynaklar eşleştirilir.

4.11.2.1 OpenGL-CUDA İşbirliği için Gerekli Kütüphaneler

“cudaGL.h” ve “cuda_gl_interop.h” kütüphaneleri ile kullanılacak fonksiyonlar uygulamaya tanıtılmış olur.

4.11.2.2 Aygıt(Device) Seçimi

“cudaGLSetGLDevice” fonksiyon çağrıları ile gerçekleştirilir. Bu fonksiyonun alacağı parametre kullanılacak olan aygıtın ID'sidir. İstenirse aygıtı seçmek ve aygıtın ID'sini geri döndürmek için “cudaChooseDevice” fonksiyonu kullanılabilir.

4.11.2.3 Tampon Nesnesi Oluşturma

Eşleme yapılabilmesi için verileri tutacak tampon nesneler oluşturulmalıdır. 2B'lu grafikler ve 3B'lu grafikler için ayrı ayrı tampon yapıları mevcuttur.

Pikselt Tampon Nesnesi(Pixel Buffer Object-PBO)

İki boyutlu görüntüler için piksel tampon nesnesi oluşturularak GİB ile eşleme sağlanır. Piksel tampon nesnesi OpenGL'de pikselleri depolamaya yarayan bir bellek bölgesidir.[53]

Pikselt Tampon nesnesi oluşturmak için sırasıyla aşağıdaki fonksiyonlar kullanılmaktadır:[54]

GLuint tamponNesne;

glGenBuffers(1, &tamponNesne);

glBindBuffer(GL_PIXEL_UNPACK_BUFFER, tamponNesne);

***glBufferData(GL_PIXEL_UNPACK_BUFFER, size, NULL,
GL_STREAM_DRAW);***

glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);

Yukarıda yazılan kodların ilk iki satrı ile “tamponNesne” isimli bir piksel tampon nesnesi oluşturuldu. “glBindBuffer” fonksiyonu ile oluşturulan nesne ile hedef arasındaki bağlantı sağlanır. “glBufferData” fonksiyonu ise piksel veri tamponu için ayrılacak hafıza alanının büyüklüğünü ayarlar.

4.11.2.3.1 Köşe Tampon Nesnesi (Vertex Buffer Object-VBO)

Üç boyutlu köşeler için köşe tampon nesnesi oluşturularak GİB ile eşleme sağlanır. Köşe tampon nesnesi OpenGL'de köşe noktaları depolamaya yarayan bir bellek bölgesidir. Renklendirilmiş yüzeyler, tel kafes biçimindeki bir görüntü ya da 3B'lu noktalardan oluşmuş kümelerin ağ bilgisini görüntülemek için kullanılır.[53]

Köşe tampon nesnesi oluşturmak için gerekli fonksiyonlar, piksel tampon nesnesi oluşturmada kullanılan fonksiyonlardır.

Köşe tamponlar ve piksel tamponlar C dilindeki dizilere benzerdir. Farklı olarak bu tamponlar sistem belleği yerine Device(GİB) belleğini kullanırlar.[54]

4.11.2.4 CUDA Kaynağının Belirlenmesi

“cudaGraphicsResource * cudaKaynagi ” yapısı ile kaynak belirlenir.

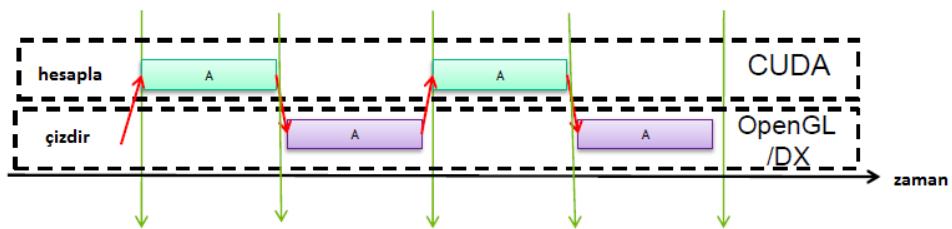
4.11.2.5 Kaynak ile Hedef Arasında Kayıt Yapılması

```
cudaGraphicsGLRegisterBuffer(&cudaKaynagi,tamponNesne,cudaGraphicsMapFlagsNone);
```

tanımlaması ile kaynak ile hedef arasındaki kayıt gerçekleştirilmiş olur. Fonksiyonun aldığı son parametre ile eşleşme sonrası erişim izinleri belirlenir.

4.11.2.6 Kaynak Eşlemesi

“cudaGraphicsMapResources” fonksiyonu ile eşleme yapılır. Eşlemeyi kaldırmak için “cudaGraphicsUnmapResources” fonksiyonu kullanılır. Döndürülen işaretçi ile kernel veri tamponundaki verilere erişim sağlamak için “cudaGraphicsResourcesGetMappedPointer” fonksiyonundan yararlanılmaktadır.



Şekil 4-45: OpenGL ile etkileşimli olarak çalışan bir CUDA uygulamasının şematik gösterimi

BÖLÜM 5

ÇİFT YÖNLÜ KARAYOLU TRAFİĞİNİN HÜCRESEL OTOMAT TABANLI 2B BENZETİMİ VE CUDA MİMARİSİ İLE HIZLANDIRILMASI

5.1 MATERYALLER

5.1.1 CUDA Destekli Ekran Kartı(NVIDIA GeForce GT 520M)

Uygulamanın gerçekleştirileceği grafik kartının teknik özellikleri Şekil 5-1'de görülmektedir.

```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v7.0\1_Utilities\deviceQuery\..\bin\win64...
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v7.0\1_Utilities\deviceQuery\..\bin\win64...
./bin/win64/Debug/deviceQuery.exe Starting...
CUDA Device Query <Runtime API> version <CUDART static linking>
Detected 1 CUDA Capable device(s)

Device 0: "GeForce GT 520M"
  CUDA Driver Version / Runtime Version      7.0 / 7.0
  CUDA Capability Major/Minor version number: 2.1
  Total amount of global memory:             1024 MBytes (1073741824 bytes)
    < 1> Multiprocessors, < 48> CUDA Cores/MP:   48 CUDA Cores
    GPU Clock rate:                         1344 MHz (1.34 GHz)
    Memory Clock rate:                     800 Mhz
    Memory Bus Width:                      64-bit
    L2 Cache Size:                         131072 bytes
    Maximum Texture Dimension Size <x,y,z>  1D=<65536>, 2D=<65536, 65535>,
3D=<2048, 2048, 2048>
    Maximum Layered 1D Texture Size, <num> layers 1D=<16384>, 2048 layers
    Maximum Layered 2D Texture Size, <num> layers 2D=<16384, 16384>, 2048 layers
    Total amount of constant memory:          65536 bytes
    Total amount of shared memory per block:  49152 bytes
    Total number of registers available per block: 32768
    Warp size:                            32
    Maximum number of threads per multiprocessor: 1536
    Maximum number of threads per block:        1024
    Max dimension size of a thread block <x,y,z>: (1024, 1024, 64)
    Max dimension size of a grid size <x,y,z>: (65535, 65535, 65535)
    Maximum memory pitch:                  2147483647 bytes
    Texture alignment:                    512 bytes
    Concurrent copy and kernel execution: Yes with 1 copy engine(s)
    Run time limit on kernels:           Yes
    Integrated GPU sharing Host Memory: No
    Support host page-locked memory mapping: Yes
    Alignment requirement for Surfaces: Yes
    Device has ECC support:              Disabled
    CUDA Device Driver Mode <TCC or WDDM>: WDDM <Windows Display Driver Mode>
  Device supports Unified Addressing <UVA>: Yes
  Device PCI Domain ID / Bus ID / location ID: 0 / 1 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

Sekil 5-46:NVIDIA GeForce GT 520M ekran kartı aygit sor gulama ekran ciktisi

5.1.2 Microsoft Visual Studio 2012 Ultimate

Microsoft firması tarafından geliştirilmiş olan bir tümleşik geliştirme ortamıdır. Microsoft Windows platformunda çalışmaktadır. Kapalı kaynak kod tipinde bir uygulamadır. Aslında herhangi bir dili ya da aracı desteklemez. Bağlantıları “VSPackage” ile sağlar.[55]

5.1.3 CUDA Toolkit V7.0

NVIDIA firması tarafından CUDA mimarisi ile uygulamalar geliştirebilmek için sunulan yazılım geliştirme platformudur. cuFFT, cuBLAS,cuRAND gibi kütüphanelerin yanında Nsight IDE, Visual Profiler gibi geliştirme araçları CUDA Toolkit kurulumu ile birlikte gelmektedir. C++11’i desteklemektedir.

5.2 YÖNTEM

5.2.1 Özet

Projede uygulama iki aşamada gerçekleştirilmiştir. Birinci aşamada sadece MİB üzerinde yürütülmekte olan bir OpenGL uygulaması bulunmaktadır. İkinci aşamada ise uygulama CUDA mimarisinden faydalılarak hem MİB hem de GİB üzerinde yürütülmektedir. Her iki aşamada da hücresel otomat teknüğinden faydalılmış ve NaSch Modeli temel alınarak kodlamalar yapılmıştır. Her bir hücre ya dolu ya da boş olacak şekilde tasarlanmıştır. Hücre durumunun “false” olması hücrenin boş, “true” olması hücrenin dolu olduğunu ifade etmektedir. Şeritler kapalı bir sistem modelinde tasarlanmıştır, bir döngüye sahiptir. Bu modele göre aracın haritadan çıkış yapması durumunda tekrar ilk başladığı noktaya geri dönmesi sağlanmıştır. Bunun sebebi hücresel otomatın paralel bir güncellemeye ihtiyaç duyuyor olmasıdır.



Şekil 5-47:Hücresel otomat modelinde aracın haritaya giriş-çıkış tipleri[56]

Şekil 5-2'de bir hücresel otomat modelinde araçların haritaya giriş ve çıkış tipleri görülmektedir. Soldaki yöntemdeikan her araç benzetim dışına alınır ve yeni bir araç oluşturularak haritaya dahil edilir. Sağdaki yöntemde ise haritadan çıkan araç başladığı konuma geri dönmektedir. Bu projede sağdaki yöntem uygulanmıştır.

Uygulamada NaSch modelinin stokastik versiyonu seçilmiştir. Buna göre NaSch modelindeki 3.kural için bir “Random Brake Probability(Rasgele Fren Olasılığı)” değeri belirlenmiştir. Bu model sadece bölünmüş anayol ve şehirlerarası çift yönlü yol olan D567 yoluna uygulanmıştır. Diğer yollarda araçlar sabit hızlar ile ilerlemektedir.

Uygulama C++ dilinde nesneye yönelik tasarım modeli ile yazılmıştır. Her bir hücrenin uzunluğu 7.5 metre ve genişliği 3.75 m olarak belirlenmiştir. Benzetimin güncellenme süresi(time-step) 1 saniyedir. OpenGL kodları yazılırken GLUT kütüphanesinden faydalanılmıştır.

5.2.2 Benzetim Verileri

5.2.2.1 Harita Bilgisi

Projede modellenen karayolu, Tekirdağ ilinin Marmara Ereğlisi ilçesi temel alınarak, İstanbul-Tekirdağ karayolunun D110 ile E-84 kodlu yolların kesiştiği bölgedir.



Şekil 5-48:Modellenecek Karayolunun GoogleMaps'ten Alınan Harita Bilgisi

Şekil 5-3'te ve Şekil 5-4'te görülmekte olan bölgede modellenen ayrılmış yol uzunluğu 1km olarak planlanmıştır. Ayrıca Edirne-Marmara Ereğlisi Yolu (D567), alt geçitli dönel kavşak(Bulvar Cd.) ve Marmara Ereğlisi merkezine ait muhtelif cadde ve sokakların benzetimde modellenmiştir.



Şekil 5-49:Modellenecek Karayolunun GoogleMaps'ten Alınan Yeryüzü Bilgisi

5.2.2.2 Araç Bilgileri

Benzetimde 5 araç tipi yer almaktadır. Bunlar otomobil,tır,kamyon,otobüs ve kamyonettir. Her araç tipinin uzunluk ve genişlik bilgileri birbirinden farklıdır. Araçların uzunluklarına göre kapladıkları hücre sayıları da değişkenlik göstermektedir. Benzetimde görsel olarak belirginlik amaçlandığı için her bir araç tipine göre ayrı renk değerleri verilmiştir. Araçların fiziksel bilgileri Tablo 5-1'de görülmektedir.

Çizelge 5-1: Benzetimde yer alan araçların fiziksel özelliklerini

Araç Tipi No	Araç Tipi	Genişlik	Uzunluk	Renk	Kapladığı Hücre Sayısı
0	Otomobil	2.0	4.6	Siyah	1
1	Tır	2.45	13.6	Yeşil	2
2	Kamyon	2.45	7.2	Kırmızı	1
3	Otobüs	2.55	12.8	Gri	2
4	Kamyonet	2.0	4.0	Mavi	1

5.2.2.3 Araçların Hız Sınırları

Türkiye'de araçların uyması gereken hız sınırları Şekil 5-5'te görülmektedir.Buna göre benzetimdeki araçların uyacakları hız sınırları Tablo 5-2'de gösterilmiştir.

TÜRKİYE'DE ARAÇLARIN UYMASI GEREKEN YASAL HIZ SINIRLARI				
ARAÇ CİNSİ	VERLEŞİM YERİ İÇİNDE	VERLEŞİM YERİ DİŞİNDƏ		OTOYOLLARDA
		SEHİRLERARASI ÇİFT YÖNLÜ KARAYOLLARINDA	BÖLÜNMÜŞ YOLLarda	
Otomobil (M1), (M1G),	50	90	110	120
Minibüs (M2),	50	80	90	100
Otobüs (M2-M3),	50	80	90	100
Kamyonet (N1), (N1G),	50	80	85	95
Panelvan (N1)	50	85	100	110
Kamyon (N2-N3),	50	80	85	90
Çekici (N2-N3),	50	80	90	100
Motosiklet (L3),	50	70	80	80
Motosiklet (L4, L5, L7),	50	45	45	Giremez
Motorlu Bisiklet (L1, L2, L6),	30	50	50	60
Motorsuz Bisiklet				
Tehlikeli madde taşıyan araçlar ve özel yük taşıma izin belgesi veya özel izin belgesi ile karayoluna çıkan araçlarda (Belgelerinde aksine bir hüküm yoksa)	30	30	40	Giremez
Lastik tekerlekli traktörler	20	20	30	40
Anızalı bir aracı çeken araçlar	20	20	20	Yolun yapım, bakım veya işletilmesinden sorumlu kuruluştan izin alınmadan giremez
İş makineleri	20	20	20	

Şekil 5-50: Türkiye'de araçların uyması gereken hız sınırları[57]

Çizelge 5-2: Benzetimde yer alan araçların maksimum hızları

Araç Tipi	Bölünmüş Ana Yol	Yerleşim İçi	Otoyol	D567 Çift Yönlü Şehirlerarası Yol
Otomobil	110 km/sa	50km/sa	120km/sa	90km/sa
Tır	50km/sa	30km/sa	60km/sa	50km/sa
Kamyon	85km/sa	50km/sa	90km/sa	85km/sa
Otobüs	90km/sa	50km/sa	100km/sa	90km/sa
Kamyonet	85km/sa	50km/sa	95km/sa	85km/sa

Benzetimde uygulanan hücresel otomat modelinin hücre uzunlukları 7.5 metredir ve güncellenme süresi 1 saniyedir. Bu durumda her bir hücrenin geçilmesi için gereken hız 27km/sa olarak hesaplanır. Benzer şekilde saniyede 2 hücre ilerleyen bir aracın hızı 54km/sa'tır. Saniyede 3 hücre ilerleyen aracın hızı 81km/sa ve 4 hücre ilerleyen aracın da hızı 108km/sa olarak belirlenir. Bu hız benzetimde uygulanabilecek maksimum hızdır.

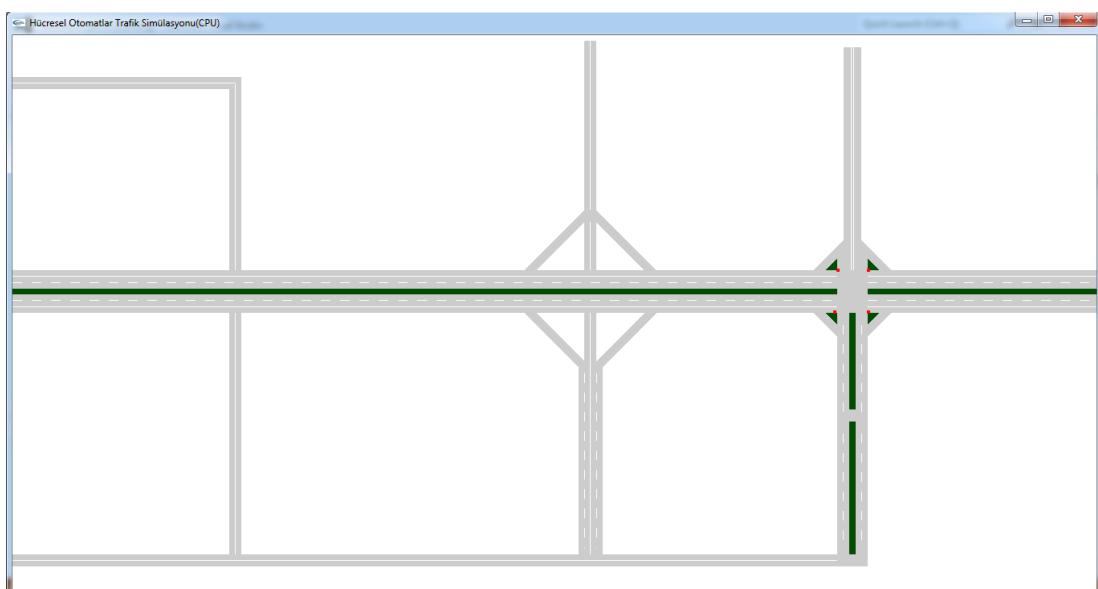
5.2.2.4 Yol Bilgisi

Benzetimde yer alan kavşaklardan birinin sinyalizasyon sistemi mevcuttur. Diğer kavşağın sinyalizasyonu bulunmamakta ve araçlar müsait durumlarda geçiş yapabilmektedir. Şeritlerinde ilerlemekte olan araçlar dönüş yolu gördüklerinde random olarak yön seçmekte ve yani yönlerine göre ilerlemeye devam etmektedirler.

5.2.3 Benzetimin MİB Tarafı

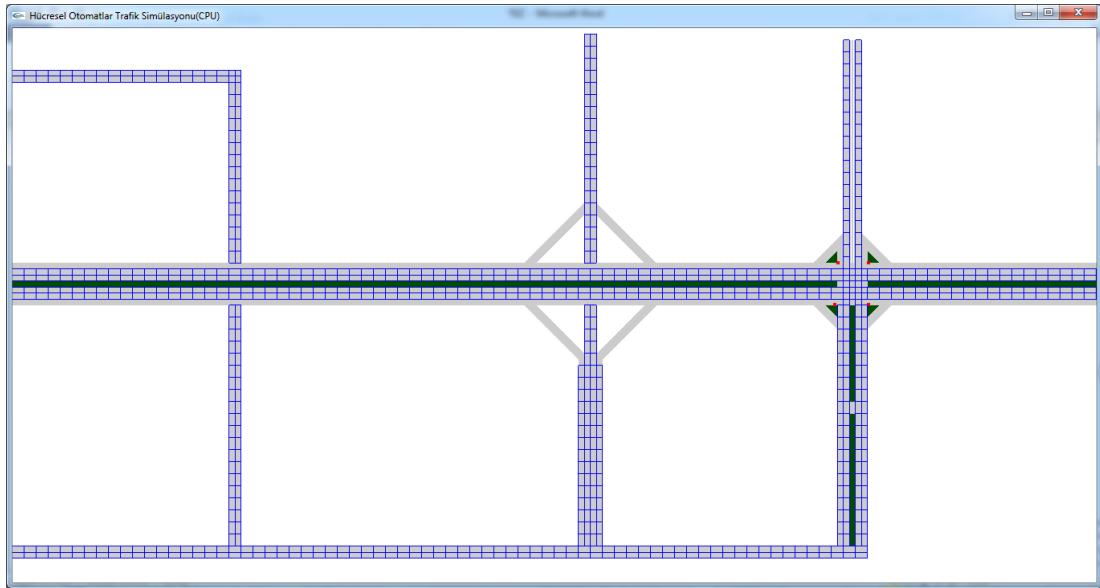
5.2.3.1 Haritanın Hazırlanması

Harita hazırlanırken OpenGL kütüphanesi kullanılmış, GLUT'tan yararlanılmış ve kodlar C++ ile yazılmıştır. “glRectf” , “glVertex3f” gibi fonksiyonların yanısıra “GL_QUADS” ve “GL_POLYGON” gibi temel OpenGL şekilleri ile harita görsel olarak oluşturulmuştur. Şekil 5-6'da modellenen harita görülmektedir.



Şekil 5-51: Benzetimde modellenen harita

Benzetim hücresel otomat modeline uygun olarak yapılandırıldığı için, görsel olarak da bu hücreleri ifade etmek benzetimi izlemeyi kolaylaştıracaktır. Bu amaçla hücreler Şekil 5-7'deki görüldüğü gibi oluşturulmuştur.



Şekil 5-52: Hücrelerin görsel olarak ifade edilmesi

Haritada görülen hücrelerin her biri 7.5 metre uzunluğu ve 3.75 metre genişliği ifade etmektedir. Hücre ızgaralarını oluşturmada kullanılan aşağıdaki örnek kod parçası incelendiğinde yapı daha iyi anlaşılacaktır.

```

glLineWidth(1.f);
//Tekirdağ-İstanbul yolu (bölünmüş yol)
for(i=0;i<90;i++)
    for( j=47;j<49;j++)
    {
        glColor3f(0,0,1);
        glBegin(GL_LINE_LOOP);
        glVertex2f(i*7.5f,j*3.75f);
        glVertex2f((i*7.5f)+7.5,j*3.75);
        glVertex2f((i*7.5f)+7.5, (j*3.75)+3.75);
        glVertex2f(i*7.5f, (j*3.75)+3.75);
        glEnd();
    }
}

```

Harita oluşturulabilmesi için “main.cpp” içerisinde bulunan “RenderScene()” fonksiyonu içerisinde “Harita()” fonksiyonu çağrılmalıdır. Aynı şekilde hücrelerin harita üzerine yerleştirilmesi için de “Grid()” fonksiyonu çağrılmalıdır.

5.2.3.2 Araçların Oluşturulması

Benzetimde oluşturulan araçların her biri nesne yapısı ile oluşturulmuştur. Uygulamada düzen olması açısından aşağıda kodları verilen “arac.h” isimli başlık dosyasında gerekli tanımlamalar yapılmış ve ardından “arac.cpp” dosyasında nesne oluşturmak için gerekli fonksiyonlar yazılmıştır.

```
//arac.h header dosyası
#ifndef ARAC_H
#define ARAC_H
#include <iostream>
#include <GL/glut.h>
#include <GL/gl.h>
#include <GL/glu.h>
using namespace std;

class ARAC{
public:
    float solX,solY,genislik,uzunluk,yukseklik;
    int yon;
    int yol_id;
    int serit_id;
    int hız;
    GLint renk;
    int a_tipi;//araç tipi:0-otomobil,1-tır,2-kamyon,3-kamyonet,4-otobüs
    int hc_id;
    //solX,solY,arac tipi,hız,yön,yol_id,serit_id,hc_id
    ARAC(float,float,int,int,int,int,int,int);
    void AracCiz();//aracın görsel olarak ekrana yansıtılmasını sağlar
    void Update(int);//oluşturulan aracın id'si ile güncelleme yapar

};

#endif
```

“arac.h” dosyasında görülmekte olan “Update()” fonksiyonu “main.cpp” içerisindeki tetikleyici fonksiyon olan “glutTimerFunc()” a parametre olarak giren “waitAndRedraw()” fonksiyonu içerisinde çağrılmaktadır. “glutTimerFunc” aldığı ilk parametre güncellenme süresi ve ikinci parametresi ise güncellenme yapmak üzere tetikleyeceği fonksiyon ismini belirtmektedir. “Update” fonksiyonu NaSch modeline göre kuralları işleterek araçların ilerlemesini kontrol etmektedir. Aşağıdaki kod bloğunda “Update()” fonksiyonunun çağrılmama aşaması görülmektedir.

```
void waitAndRedraw(int deger)
{
    for(int id=0;id<arac_sayisi;id++)
    {
```

```

    araclar[id]->Update(id);
}

glutPostRedisplay();
glutTimerFunc(WAIT,waitAndRedraw,1);
}

```

Araçların nesne yapısında oluşturulabilmesi için parametreli kurucu fonksiyon kullanılmıştır. Bu parametreli kurucu fonksiyonun aldığı değerler “arac_dizisi[arac_sayisi][10]” dizisinden çekilmektedir. Her araç oluşturulduğu konumdaki hücreyi doldurur ve hücre durumu “false”dan “true”ya dönüştürülür. Bunun için “HucreDoldur” isimli bir fonksiyon tanımlanmıştır. Aşağıdaki kod parçasında bir aracın oluşturulurken aldığı ilk değerlerin atanması aşaması görülmektedir. Bu amaçla oluşturulan fonksiyonun ismi “Initial”dır

```

void Initial()
{
    for(int id=0;id<arac_sayisi;id++)
    {
        araclar[id]=new ARAC(arac_dizisi[id][0],arac_dizisi[id][1],arac_dizisi[id][2],
                             arac_dizisi[id][3],arac_dizisi[id][4],arac_dizisi[id][5],
                             arac_dizisi[id][6],arac_dizisi[id][7]);
        HucreDoldur(araclar[id]->yol_id,araclar[id]->serit_id,
                    araclar[id]->hc_id,araclar[id]->a_tipi);
    }//for döngüsü sonu
}// Initial() fonksiyonu sonu

```

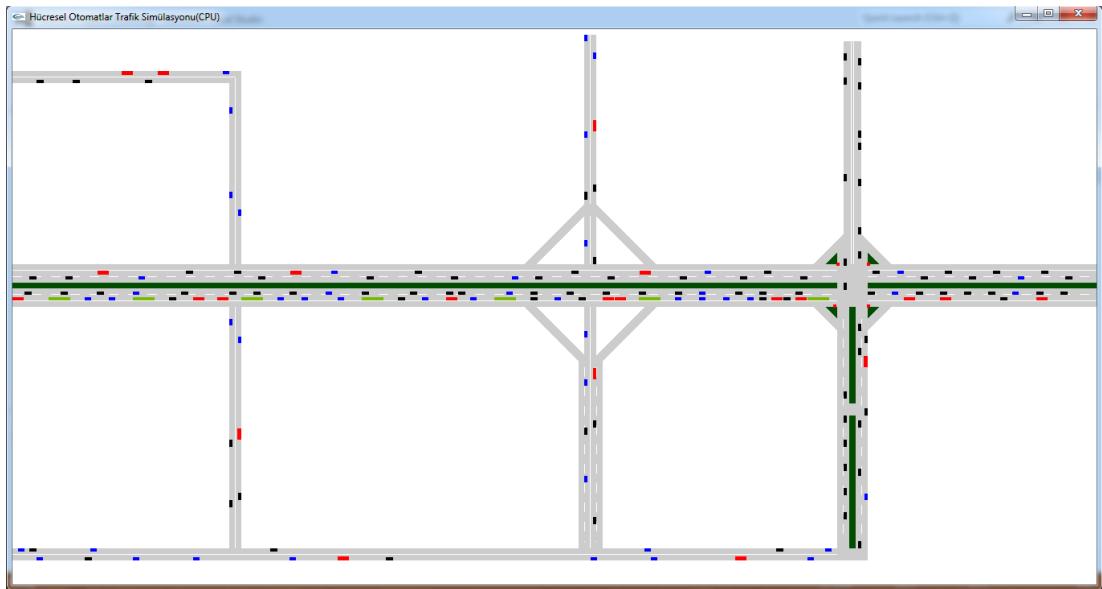
OpenGL-GLUT işbirliğiyle oluşturulan araçların ekranda gösterilmesini sağlayan fonksiyon “glutDisplayFunc()” fonksiyonuna parametre olarak geçirilen “RenderScene()” fonksiyonudur. Bu fonksiyon içeriği aşağıdaki kod parçasında ifade edilmiştir.

```

void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glPushMatrix(); //Harita ve hücreler hep sabit olduğu için
    Harita(); //glPushMatrix() ile glPopMatrix() arasına yazıldı
    Grid();
    glPopMatrix();
    for(int i=0;i<arac_sayisi;i++)
    {
        araclar[i]->AracCiz();
    }
    glutSwapBuffers();
}

```

}



Şekil 5-53: Değişik konumlarda bulunan araçların harita ve hücreler üzerinde yerleşimi

5.2.3.3 Araçların Hareket Ettirilmesi

Araçların hareket ettirilmesi gitmek istediği hücrenin dolu olup olmamasına göre değişmektedir. Eğer boş ise konum değiştirir, değilse hareket edemez. Benzetimde yer alan tüm yolların, yollarda bulunan şeritlerin ve her bir şeritteki hücrenin birer kimliği (ID) vardır. Bu kimlik numarası ile hücrenin dolu olup olmadığı kontrolü yapılır.

```
if(yol1[idx+1][serit_id]==false)
```

kod satırı ile “yol1” isimli bir yol üzerinde bulunan aracın bulunduğu hücrenin bir sonraki hücresinin boş olup olmadığı kontrolü yapılmaktadır. Eğer kontrol ettiği hücre boş ise aşağıdaki kod satırları çalıştırılacak ve arac hareket edecektir.

```
araclar[id]->solX+=7.5;  
yol1[idx+1][serit_id]=true;  
yol1[idx][serit_id]=false;  
araclar[id]->hc_id++;
```

Kavşak noktalarına gelen araçlar eğer seçenekleri varsa random(rasgele) bir seçim yapacak ve buna göre ilerlemelerine devam edeceklerdir.

5.2.3.4 Hız Kontrolü

Benzetimde hız kontrolü NaSch modelindeki kurallara göre yapılmaktadır. Birinci kuralı gerçekleştirmek için önceden araç tipine göre tanımlanmış maksimum hızlara ulaşılımadığı sürece aracın hızı artırılmaktadır. Bunun için geliştirilen kod parçası:

```
//Adım 1: Hızlanma(Acceleration)
if(araclar[id]->hiz < vmax)
    araclar[id]->hiz++;
```

İkinci adımda öndeği araç ile aralarındaki boş hücre sayısına göre hız azaltma işlemi bulunmaktadır. Bunun için “getForwardGap” isimli fonksiyondan yararlanılmakta ve bu fonksiyon sonucu dönen değer ile aracın mevcut hızı “minimum” fonksiyonuna parametre olarak geçirilerek hız güncellemesi gerçekleştirilir.

```
//Adım 2: Yavaşlama (Braking)
araclar[id]->hiz=minimum(araclar[id]->hiz,getForwardGap(araclar[id]->yol_id,
araclar[id]->serit_id,araclar[id]->hc_id,araclar[id]->a_tipi));
```

“getForwardGap()” fonksiyonundan bir kod parçası:

```
int gap=0;
int count;
if(tip==3 || tip==1)
    count=2;
else
    count=1;
if(y_id==8)
{
    while( yol8[h_id+count][s_id]==false && h_id+count<90)//max. hücre sayısı 90
    {
        count++;
        gap++;
    }
    return gap;
}
```

olarak örnek verilebilir.

Üçüncü kural için öncelikle bir rasgele fren olasılığı değeri belirlenmelidir. Bu değere göre yapılan kontroller sonucu aracın hızını 1 birim azaltması sözkonusu olabilecektir.

```
//Adım 3: Random Braking
float x=float(rand())/RAND_MAX;
if(x<=RBP)
{
    araclar[id]->hiz=maximum(0,araclar[id]->hiz-1);

}
```

Dördüncü kuralın uygulanması ise Bölüm 5.2.3.3’de belirtildiği gibi aracın hareket ettirilmesi aşamasıdır. Güncellenen hız değeri ile birlikte araç hareket ettirilmektedir.

5.2.3.5 Trafik Işıkları Sinyalizasyon İşlemleri

Trafik ışıklarının üç durumu sözkonusudur. Bunlar araçların durması gerektiğini ifade eden “kızıl ışık”, araçların geçişine izin verilen durumu ifade eden “yeşil ışık” ve bu iki durum arasındaki hazırlık/kontrollü geçiş'i ifade eden “sarı ışık” durumlarıdır. Benzetimde yer alan sinyalizasyon işlemi ile araçların kontrollü geçişleri sağlanmaktadır. Benzetimde 40 saniye yeşil ışık, 5 saniye sarı ışık ve 40 saniye de kızıl ışık sonsuz döngü içerisinde yanmaktadır.

“TrafficLight[ışık_id]” değişkeninin değeri 0 ise kırmızı ışık, 1 ise sarı ışık, 2 ise yeşil ışık yanmaktadır.

5.2.3.6 Kullanıcı Etkileşimi İçin Basit Düzeyde Arayüz Oluşturulması

Benzetimin görsel olarak izlenmesi sırasında ihtiyaca göre “Hücre Çizdir/Kaldır”, “Benzetimi Duraklat/Devam Et” ve “Benzetimden Çıkış Yap” seçenekleri eklenmiştir. Bu seçenekleri kullanıcı klavyeden ya da fare ile menü açarak aktifleştirebilmekte ve pasifleştirebilmektedir.

Menü oluşturmak için gerekli kod bloğu:

```
void initMenus()
{
    glutCreateMenu(mainMenu);
    glutAddMenuEntry("Hücreler[H]", 'h');
    glutAddMenuEntry("Duraklat/Devam [D]", 'd');
    glutAddMenuEntry("Çıkış (Q)", 'q');
    glutAttachMenu(GLUT_RIGHT_BUTTON);
}
```

Klavyeden kullanıcı etkileşimi sağlamak için gerekli kod bloğu:

```
void keyboard(unsigned char key, int /*x*/, int /*y*/)
{
    switch (key)
    {
        case ('q'):
            exit(EXIT_SUCCESS);
        case 'd':
            animate = !animate;
            break;
        case 'h':
            hucreToggle = !hucreToggle;
            break;
    }
}
```



Şekil 5-54: Kullanıcı Etkileşimi İçin Oluşturulan Menü

5.2.4 Benzetimin GİB Tarafı

Grafik İşlem Birimi'ne yaptırılan işlemler sadece araçların hız ve konum bilgileriyle ilgilidir. Harita çizimi, hücre çizimi, araç çizimi, araç nesnelerinin oluşumu yine MİB üzerinde yapılmaya devam etmektedir. Araçların oluşturulmasından sonra her bir aracın konum, hız ve yön bilgileri GİB'e aktarılmalıdır. Ayrıca araçlar bulundukları hücrelerin ve hedef hücrelerin boş ya da dolu olma bilgisine de sahip olmalıdır ki doğru bir güncelleme yapılabilse. Sinyalizasyon sistemine sahip olan kavşağın trafik ışık bilgisi de Grafik İşlem Birmine aktarılmalıdır. Tüm bu bilgiler aktarıldıkten sonra GİB, paralel olarak verileri işleyip MİB'e geri aktarmakta ve veriler bundan

sonra eskisi gibi seri olarak işlenmeye devam etmektedir. Bu heterojen hesaplama işi CUDA mimarisi sayesinde yapılmaktadır.

5.2.4.1 MİB Üzerindeki Verilerin GİB İçin Bölünmesi

Initial() fonksiyonu, GİB üzerine nesne biçiminde aktarım yapılmadığı için aşağıdaki şekilde yeniden yazılmıştır:

```
void Initial()
{
    for(int id=0;id<arac_sayisi;id++)
    {
        SOLX[id]=arac_dizisi[id][0];
        SOLY[id]=arac_dizisi[id][1];
        TIP[id]=arac_dizisi[id][2];
        HIZ[id]=arac_dizisi[id][3];
        YON[id]=arac_dizisi[id][4];
        YOLID[id]=arac_dizisi[id][5];
        SERITID[id]=arac_dizisi[id][6];
        HCID[id]=arac_dizisi[id][7];

        HucreDoldur(YOLID[id],SERITID[id],HCID[id],TIP[id]);
    }
}
```

5.2.4.2 Verilerin MİB - GİB Arasında Kopyalanması

GİB, üzerinde veri tanımlanmadıkça ve üzerine veri kopyalanmadıkça MİB üzerindeki verilere yabancıdır, tanımaz. Bu amaçla waitAndRedraw() tetikleyici güncelleme fonksiyonu içerisinde aygit(device) üzerinde yer ayırma, aygit üzerine veri kopyalama, kernel fonksiyonu çağrıldıktan sonra verilerin tekrar host(MİB) üzerine geri kopyalanması ve aygit üzerinde ayrılan dinamik bellek bölgesinin serbest bırakılması gibi bir dizi işlem gerçekleştirilmektedir.

Aşağıdaki kod satırları örnek olarak araç bilgilerini, trafik ışık bilgisini ve benzetimde yer alan bir yol bilgisini MİB-GiB arasında kopyalamakta ve gerekli diğer işlemleri yapmaktadır:

```
float *dev_SOLX,*dev_SOLY;
int *dev_TIP,*dev_HIZ,*dev_YON,*dev_YOLID,*dev_SERITID,*dev_HCID;
int *dev_yol1;
int *dev_TL;
cudaMalloc((void**)&dev_SOLX,arac_sayisi*sizeof(float));
cudaMalloc((void**)&dev_SOLY,arac_sayisi*sizeof(float));
cudaMalloc((void**)&dev_TIP,arac_sayisi*sizeof(int));
cudaMalloc((void**)&dev_HIZ,arac_sayisi*sizeof(int));
cudaMalloc((void**)&dev_YON,arac_sayisi*sizeof(int));
cudaMalloc((void**)&dev_YOLID,arac_sayisi*sizeof(int));
cudaMalloc((void**)&dev_SERITID,arac_sayisi*sizeof(int));
cudaMalloc((void**)&dev_HCID,arac_sayisi*sizeof(int));
cudaMalloc((void**)&dev_yol1,35*2*sizeof(int));
cudaMalloc((void**)&dev_TL,4*sizeof(int));
cudaMemcpy(dev_SOLX,SOLX,arac_sayisi*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_SOLY,SOLY,arac_sayisi*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_TIP,TIP,arac_sayisi*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dev_HIZ,HIZ,arac_sayisi*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dev_YON,YON,arac_sayisi*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dev_YOLID,YOLID,arac_sayisi*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dev_SERITID,SERITID,arac_sayisi*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dev_HCID,HCID,arac_sayisi*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dev_yol1,yol1,35*2*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dev_TL,TrafficLight,4*sizeof(int),cudaMemcpyHostToDevice);

cudaMemcpy(SOLX,dev_SOLX,arac_sayisi*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(SOLY,dev_SOLY,arac_sayisi*sizeof(float),cudaMemcpyDeviceToHost);
cudaMemcpy(TIP,dev_TIP,arac_sayisi*sizeof(int),cudaMemcpyDeviceToHost);
cudaMemcpy(HIZ,dev_HIZ,arac_sayisi*sizeof(int),cudaMemcpyDeviceToHost);
cudaMemcpy(YON,dev_YON,arac_sayisi*sizeof(int),cudaMemcpyDeviceToHost);
cudaMemcpy(YOLID,dev_YOLID,arac_sayisi*sizeof(int),cudaMemcpyDeviceToHost);
cudaMemcpy(SERITID,dev_SERITID,arac_sayisi*sizeof(int),cudaMemcpyDeviceToHost);
cudaMemcpy(HCID,dev_HCID,arac_sayisi*sizeof(int),cudaMemcpyDeviceToHost);
cudaMemcpy(yol1,dev_yol1,35*2*sizeof(int),cudaMemcpyDeviceToHost);
cudaMemcpy(TrafficLight,dev_TL,4*sizeof(int),cudaMemcpyDeviceToHost);

cudaFree(dev_SOLX);
cudaFree(dev_SOLY);
cudaFree(dev_TIP);
cudaFree(dev_HIZ);
cudaFree(dev_YON);
cudaFree(dev_YOLID);
cudaFree(dev_SERITID);
cudaFree(dev_HCID);
cudaFree(dev_yol1);
```

```
cudaFree(dev_TL);
```

5.2.4.3 Kernel Fonksiyonunun Çağrılması

Kernel fonksiyonu, GİB üzerinde işlemleri gerçekleştiren fonksiyondur. Aldığı parametreleri işler ve ardından MİB'e sonuçları gönderir.

Kernel fonksiyonunun işlenmesi için grid, block ve thread boyutlarının ayarlanması gereklidir. Bu amaçla aşağıdaki kod satırları yazılmıştır:

```
dim3 threads,blocks;
threads.x=arac_sayisi;
blocks.x=1;
```

Buna göre grid ve block tek boyutludur(sadece x-boyutu var). Kernel çağrıldığında her bir araç için grid kopyası oluşturulur ve ardından paralel olarak işlenir. Araç sayısı adedince block bulunmakta ve her bir block bir adet thread içermektedir.

```
kernel<<<blocks,threads>>>(dev_SOLX,dev_SOLY,dev_TIP,dev_HIZ,dev_YON,dev_YOLID,dev_SERITI
D,
dev_HCID,dev_yol1,dev_TL);
```

5.2.4.4 Kernel Fonksiyonu İşlenmesi

CPU tarafına benzer olarak işlenecek parametreler araç bilgileridir. Aşağıdaki kod bloğunda “yol1” isimli yol üzerinde araçların konum değiştirme sorgulamaları ve işlemleri gerçekleştirilmektedir.

```
__global__ void kernel(float *sx,float *sy,int *tip, int *hiz,int
*yon,int *yolID,int *srtID,int *hcID,int *yol1, int *TrafficLight)
{
    int id=threadIdx.x + blockIdx.x * blockDim.x;

    //YOL1
    if(yolID[id]==1 )//yol1 mi?
    {
        int hdx=hcID[id];
        int sdx=srtID[id];
        int serit_sayi=2;
        int idx=hdx*serit_sayi+sdx;

        //kavşak noktasına kadar ilerle
        if(sdx==0 && yon[id]==1)
        {

            if(yol1[idx+serit_sayi]==0)
            {
                sx[id]+=7.5;
                yol1[idx+serit_sayi]=1;
                yol1[idx]=0;
                hcID[id]++;
            }
        }
    }
}
```

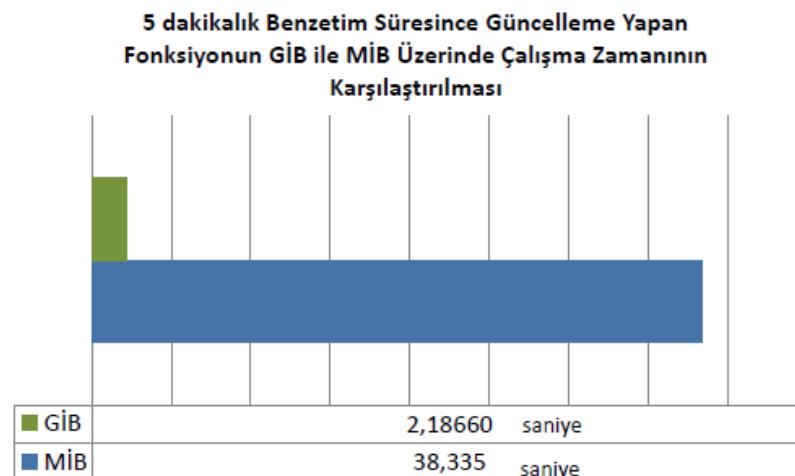
BÖLÜM 6

SONUÇLAR VE ÖNERİLER

Bu çalışmada, konum bilgileri ve kategorileri bilinen araçların yer aldığı bir trafik akışının bilgisayar grafikleri kullanılarak MİB ve GİB üzerinde 2B benzetimi gerçekleştirilmiş, performans karşılaştırılması yapılmıştır.

CUDA mimarisi ile geliştirilen uygulamanın tamamı GİB üzerinde yürütülmez. Bu projede sadece araçların konum ve hız güncelllemelerini yapan fonksiyon GİB üzerinde gerçekleştirilmiş ve performans karşılaştırması da sadece MİB üzerinde aynı görevi yapan fonksiyonlar arasında yapılmıştır.

5 dakika boyunca devam ettirilen benzetim verileri izlenmiş ve elde edilen bulgular kaydedilmiştir. Benzetimde 200 adet araç yer almaktadır. Güncelleme yapan fonksiyonun zaman verileri Şekil 6-1'de görülmektedir.



Şekil 6-55: GİB ile MİB Üzerinde Çalıştırlan Fonksiyonların Zaman Verileri

Yapılan karşılaştırma sonucu GİB üzerinde, MİB'e göre yaklaşık 17 kat daha az bir zamanda işlemi tamamlama süresi elde edilmiştir. Diğer bir deyişle CUDA sayesinde yaklaşık 17 kat hızlanma gerçekleşmiştir.

İlk bakışta bu hızlanma üstün bir performans bulgusu gibi görünse de projede az sayıda yol bilgisi olduğu unutulmamalıdır. Bunun önemi, kernel fonksiyonun yol dizilerini parametre olarak alıyor olmasıdır. Ne kadar çok yol dizisi MİB-GİB arasında kopyalanırsa o kadar zaman kaybı olacaktır.

Bu çalışmada araçlar nesne yapısıyla oluşturulmuştur ve MİB üzerinde buna göre yazılım gerçekleştirilmiştir. Adil bir performans karşılaştırması yapmak amacıyla GİB için de bu yapı bozulmadan uygulanmaya çalışılmıştır. Ancak bu durumda kernel fonksiyonunun aldığı parametre sayısı bir hayli fazla olmuştur. Bu performansı düşüren çok önemli bir faktördür. Araçların konum ve hız güncellemeleri paralel olarak hızlı bir şekilde gerçekleştirirken GİB-MİB arası veri aktarma işlemleri beklenen performansın icra edilemesini engellemektedir.

Bunun için CUDA 5.0 ile birlikte gelen “Dinamik Paralellik” özelliğinden yararlanılabilir. Daha büyük yol verilerinin olduğu harita düşünülerek ve daha fazla araç sayısının olacağı varsayılarak C++’ın Standart Şablon Kütüphanesi kullanılarak yazılım gerçekleştirilebilir. CUDA tarafı için Thrust C++ Şablon Kütüphanesinden faydalanaılabilir.

KAYNAKLAR

1. İnternet: “Traffic Simulation”
http://en.wikipedia.org/wiki/Traffic_simulation (Erişim Tarihi: 28.04.2015)
2. KURNAZ, İ. (2012) “Etkileşimli Sürücü Eğitimi İçin Kural Tabanlı Bir Platform Geliştirilmesi”, Doktora Tezi, **Karabük Üniversitesi Fen Bilimleri Enstitüsü**,sf.3-5
3. UNCU, C.(2006) “Boğaziçi Köprüsü Gişe Sahasının AIMSUN Mikrosimülasyon Yazılımı ile Modellenmesi”, Yüksek Lisans Tezi, **Yıldız Teknik Üniversitesi Fen Bilimleri Enstitüsü**
4. NABİYEV, V.V. (Eylül 2012). Yapay Zeka: İnsan-Bilgisayar Etkileşimi, **Seçkin Yayıncılık** (4),sf:407
5. İnternet: “Nursing Decision Support and Expert Systems & Artificial Intelligence”,
<http://www.nursing-informatics.com/kwantlen/nrsg3160.html> (Erişim Tarihi: 28.04.2015)
6. NABİYEV, V.V. (Eylül 2012). Yapay Zeka: İnsan-Bilgisayar Etkileşimi, **Seçkin Yayıncılık** (4),sf:409-410
7. İnternet: AKDOĞAN, E. ve TACGIN,E. “Zeki Bir Trafik Kontrolü Denemesi:Bir Kavşağıın Kontrolü İçin Geliştirilen Prototip Bir Uzman Sistem ”,
<http://www.yildiz.edu.tr/~eakdogan/publication/zeki%20trafik%20kontrolu.pdf> (Erişim Tarihi: 28.04.2015)

8. İnternet: “Bulanık Mantık”,
http://tr.wikipedia.org/wiki/Bulan%C4%B1k_mant%C4%B1k (2015)
9. ELMAS, Ç. (Ekim 2010). Yapay Zekâ Uygulamaları, **Seçkin Yayıncılık** (2),sf.186
10. MAHMOOD, M. (2010). “Bulanık Mantık Kullanılarak Trafik Kontrolünün Tasarımı ve Uygulanması” , Yüksek Lisans Tezi, **Ankara Üniversitesi Fen Bilimleri Enstitüsü**
11. KUBAT, C. (Eylül 2014). MATLAB: Yapay Zekâ ve Mühendislik Uygulamaları, **Pusula Yayıncılık**(2),sf.635-636
12. KUBAT, C. (Eylül 2014). MATLAB: Yapay Zekâ ve Mühendislik Uygulamaları, **Pusula Yayıncılık**(2),sf.625-640
13. KURNAZ, İ. (2012) “Etkileşimli Sürücü Eğitimi İçin Kural Tabanlı Bir Platform Geliştirilmesi”, Doktora Tezi, **Karabük Üniversitesi Fen Bilimleri Enstitüsü**, sf.8
14. İnternet:“Dönüştürücüler”,
http://tr.wikipedia.org/wiki/Sonlu_durum_makinesi
(Erişim Tarihi:30.11.2014)
15. İnternet: “How to Implement State Machines in Your FPGA” <http://www.rs-online.com/designspark/electronics/knowledge-item/how-to-implement-state-machines-in-your-fpga> (Erişim Tarihi: 30.11.2014)
16. SOYSAL,K. (2008). “Çoklu Etmen Tabanlı Trafik Yönetim Sistemi”, Yüksek Lisans Tezi, **Yıldız Teknik Üniversitesi Fen Bilimleri Enstitüsü**

- 17.** ÇAMOĞLU, K.(2010) “Akıllı Etmenler ve Akıllı Etmen Yönelimli Programlama Yaklaşımı”, Yüksek Lisans Tezi, **Maltepe Üniversitesi Fen Bilimleri Enstitüsü**,sf.10-13
- 18.** SOYSAL,K. (2008). “Çoklu Etmen Tabanlı Trafik Yönetim Sistemi”, Yüksek Lisans Tezi, **Yıldız Teknik Üniversitesi Fen Bilimleri Enstitüsü**, sf. 2-3
- 19.** ÇAMOĞLU, K.(2010) “Akıllı Etmenler ve Akıllı Etmen Yönelimli Programlama Yaklaşımı”, Yüksek Lisans Tezi, **Maltepe Üniversitesi Fen Bilimleri Enstitüsü**,sf.45-48
- 20.** PATAÇI, M. (2014) “Çoklu Etmen Ortamında Nesne Tabanlı Dağıtık Bellek Paylaşımı”, Yüksek Lisans Tezi, **İstanbul Teknik Üniversitesi Fen Bilimleri Enstitüsü**
- 21.** GÖNCÜ, E. (2013) “Hafızalı Hücresel Otomat Sayısal Tasarımı” , Yüksek Lisans Tezi, **İstanbul Teknik Üniversitesi Fen Bilimleri Enstitüsü**
- 22.** İnternet: “Stephen wolfram cellular automata-a new kind of science-(yeni bir bilim dalı)” <http://myo.bartin.edu.tr/aliozsoy/dokuman/ca.html> (Erişim Tarihi: 24.11.2014)
- 23.** Nagel, K. and Schreckenberg, M. (1992) , “A cellular automaton model for freeway traffic” , **J. Physique I**
- 24.** İnternet: “Traffic Flow with Cellular Automata ”
<http://sjsu.rudyrucker.com/~han.jiang/paper/>(Erişim Tarihi: 24.11.2014)
- 25.** İnternet: “Nagel–Schreckenberg Model” http://en.wikipedia.org/wiki/Nagel-Schreckenberg_model (Erişim Tarihi: 24.11.2014)

- 26.** Rickert, M. ,Nagel ,K., Schreckenberg, M. Ve Latour,A. “Two Lane Traffic Simulations using Cellular Automata”(1996), Physica A: Statistical Mechanics and its Applications,sf.534-550
- 27.** Maerivoet, S. ve DeMoor, B. (2005), “Cellular Automata Models of Road Traffic” ,**Physics Reports** **419**(1), 1-64
- 28.** SPYROPOULOU, I. (2007) “Modelling a signal controlled traffic stream using cellular automata” **Transportation Research Part C**, 175-190
- 29.** İMREM, A.N. (2008), “Modelling of Multi Lane Highway Traffic and Analysis of Jam Formation”, Yüksek Lisans Tezi, **Boğaziçi Üniversitesi Fen Bilimleri Enstitüsü**
- 30.** İnternet: “DirectCompute” , <http://en.wikipedia.org/wiki/DirectCompute> , (Erişim Tarihi: 30.04.2015)
- 31.** İnternet: “What is CUDA ”,
http://www.nvidia.com/object/cuda_home_new.html
(Erişim Tarihi: 30.04.2015)
- 32.** İnternet: “OpenCL”, <https://www.khronos.org/opencl/> , (Erişim Tarihi: 30.04.2015)
- 33.** ESKİKAYA, B. (2012) “Distributed opencl - opencl platformunun ağ ölçüğinde dağıtılması”, Yüksek Lisans Tezi, **İstanbul Teknik Üniversitesi Fen Bilimleri Enstitüsü**
- 34.** İnternet: “CUDA/OpenCL Terminology”, <http://sett.com/gpgpu/cuda-opencl-terminology> (Erişim Tarihi: 30.04.2015)
- 35. NVIDIA** (2007),“CUDA Programming Guide Version 0.8”

36. İnternet: “Implementing the lattice Boltzmann model on commodity graphics hardware” <http://iopscience.iop.org/1742-5468/2009/06/P06016/fulltext/> (Erişim Tarihi: 30.04.2015)

37. İnternet: “Lecture : Graphics pipeline and animation ”
<http://goanna.cs.rmit.edu.au/~gl/teaching/Interactive3D/2013/lecture2.html>
(Erişim Tarihi: 30.04.2015)

38. İnternet: “What is Parallel Computing ?”
<http://www.gumuskaya.com/Teaching/Courses/COM444/Lectures/L0/L0b.htm> (Erişim Tarihi: 30.04.2015)

39. İnternet: “CUDA Teknolojisi”
http://www.hwa.com.tr/nvidia/wpcontent/uploads/2013/08/cuda_teknolojisi_3.jpg (Erişim Tarihi: 01.05.2015)

40. NVIDIA,(2012) “NVIDIA CUDA C Programming Guide Version 4.2”

41. AKÇAY, M., ŞEN, B., ORAK, İ.M., ÇELİK,A. (2011), “Paralel Hesaplama ve CUDA”, **6. Uluslar arası İleri Teknolojiler Sempozyumu (İATS’11)**

42. KIRK, D. ve HWU, W.W. (2010) “Programming Massively Parallel Processor”,
Morgan Kaufmann Publishers

43. İnternet: “Matrix Multiplication: Short Theory & first measurements”
http://charrer.net/?page_id=236 (Erişim Tarihi: 01.05.2015)

44. BRADLEY, T. (2010) ,“Advanced CUDA Optimization:1-Introduction”,
NVIDIA Corporotion

45. İnternet: “Introduction to CUDA 5.0” <http://www.3dgep.com/introduction-to-cuda-5-0/> (Erişim Tarihi: 01.05.2015)

- 46.** İnternet: “Constant Memory in CUDA” , <http://cuda-programming.blogspot.com.tr/2013/01/what-is-constant-memory-in-cuda.html> , (Erişim Tarihi: 01.05.2015)
- 47.** SANDERS, J. ,KANDROT, E. (2010) “CUDA by Example:An Introduction to General-Purpose GPU Programming”, **NVIDIA Corporotion**
- 48.** RUETSCH, G., OSTER, B. (2008) “Getting Started with CUDA”, NVISION 08: The World of Visual Computing, **NVIDIA Corporotion**
- 49.** İnternet: “PGI CUDA-x86: CUDA Programming for Multi-core CPUs” <https://www.pgroup.com/lit/articles/insider/v2n4a1.htm> (Erişim Tarihi: 03.05.2015)
- 50.** NVIDIA (2009), “Fermi Compute Architecture White Paper”, **NVIDIA Corporotion**
- 51.** NVIDIA (2012), “Kepler GK110 Architecture White Paper”, **NVIDIA Corporotion**
- 52.** İnternet: “How to Implement Performance Metrics in CUDA C/C++” <http://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/> (Erişim Tarihi: 03.05.2015)
- 53.** FARBER, R & NVIDIA Corporotion (2011), “CUDA Application Design and Development”, **Morgan Kaufmann Publishes**
- 54.** İnternet: “OpenGL Interoperability with CUDA” <http://www.3dgep.com/opengl-interoperability-with-cuda/> (Erişim Tarihi: 03.05.2015)

55. İnternet: “Microsoft Visual Studio”

http://tr.wikipedia.org/wiki/Microsoft_Visual_Studio (Erişim Tarihi: 03.05.2015)

56. HELD, T., BITTIHN, S. (2011) “Cellular automata for traffic simulation Nagel-Schreckenberg model” **Project report in Computational Physics**

57. İnternet: “Hız Sınırları”

<http://www.kgm.gov.tr/Sayfalar/KGM/SiteTr/Trafik/HizSinirlari.aspx>
(Erişim Tarihi: 27.02.2015)

EK 1

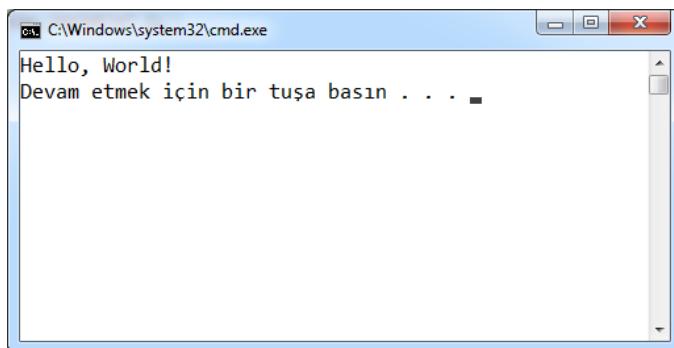
CUDA İLE İLGİLİ ÖRNEKLER

Bu bölüm projede yapılan uygulamaların en temel halinin anlaşılabilmesi ve Türkçe kaynak sıkıntısının olduğu CUDA konusuna örnekler penceresinden bakılabilmesi amacıyla hazırlanmıştır. Örneklerin kod kısımları NVIDIA desteği ile piyasaya sürülen Jason Sanders ve Edward Kandrot isimli yazarların “CUDA by Example: An Introduction to General-Purpose GPU Programming(2010)” kitabından alınmıştır. Bu kodlar üzerinde bazı değişiklikler yapılmış ve daha iyi anlaşılabilmesi amacıyla görseller ile desteklenmiştir.

İLK PROGRAM: HELLO, WORLD!

MİB KODU:

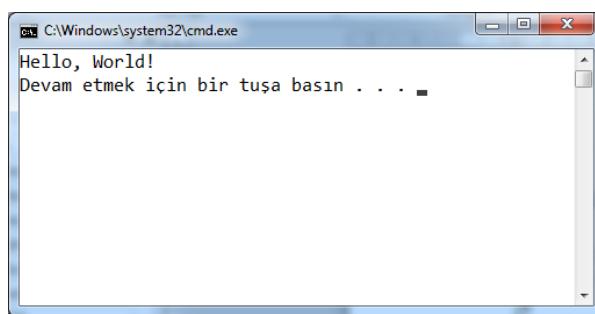
```
#include "../common/book.h"
int main( void )
{
    printf( "Hello, World!\n" );
    return 0;
}
```



Şekil Ek1- 1:Seri "Hello, World!" örneği ekran çıktısı

GİB KODU:

```
#include "../common/book.h"
__global__ void kernel( void )
{
}
int main( void )
{
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```



Şekil Ek1- 2:Paralel "Hello,World!" örneği ekran çıktısı

NOT: “ .../common/book.h” adlı kütüphane dosyasına , ilerleyen kısımlardaki örnekler için kullanılan kütüphane dosyalarına ve kitap bilgisine <https://developer.nvidia.com/content/cuda-example-introduction-general-purpose-gpu-programming-0> adresinden ulaşılabilir

PARAMETRE GEÇİRME:

```
//simple_kernel_params.cu
#include "../common/book.h"
__global__ void add( int a, int b, int *c )
{
    *c = a + b;
}
int main( void )
{
    int c; //host değişkeni
    int *dev_c; //device değişkeni

    //dev_c değişkeni için GPU üzerinde yer ayrıılıyor
```

```

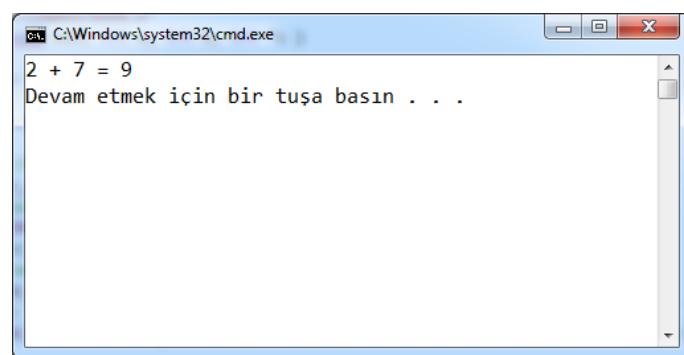
HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );

add<<<1,1>>>( 2, 7, dev_c ); //main içerisinde kernel çağrısı yapılıyor

//kernel tarafından yapılan toplama işlemi sonucu Device'tan Host'a kopyalanıyor
HANDLE_ERROR( cudaMemcpy( &c, dev_c, sizeof(int),
cudaMemcpyDeviceToHost ) );
printf( "2 + 7 = %d\n", c );

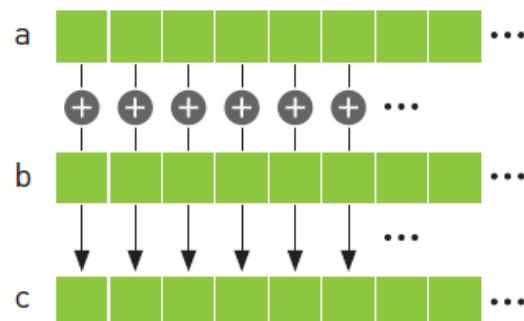
//device üzerinde ayrılan yer serbest bırakılıyor
HANDLE_ERROR( cudaFree( dev_c ) );
return 0;
}

```



Şekil Ek1- 3:simple_kernel_params.cu örneği ekran çıktısı

VEKTÖR TOPLAMI



Şekil Ek1- 4: Vektör Toplamı Bellek Modeli

MİB ÜZERİNDE VEKTÖR TOPLAMI:

```
#include "../common/book.h"
```

```

#define N 10
void add( int *a, int *b, int *c )
{
    int tid = 0;
    while (tid < N)
    {
        c[tid] = a[tid] + b[tid];
        tid += 1;
    }

}

int main( void )
{
    int a[N], b[N], c[N];
    for (int i=0; i<N; i++)
    {
        a[i] = -i;
        b[i] = i * i;
    }
    add( a, b, c );
    for (int i=0; i<N; i++)
    {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }
    return 0;
}

```

Şekil Ek1- 5: MİB üzerinde vektör toplamı ekran çıktısı

void add(int *a, int *b,int *c){...}

Bu fonksiyon ile diziler toplanmaktadır. main() içerisindeki add(a,b,c); ile a ve b dizilerinin toplanıp c dizisine aktarılmasının çağrısı yapıldı. Ancak fonksiyonun parametresi dizinin kendisi değil, başlangıç adresinin tutulduğu işaretçilerdir.

GİB ÜZERİNDE VEKTÖR TOPLAMI:

```
#include "../common/book.h"
#define N 10
__global__ void add( int *a, int *b, int *c )
{
    int tid = blockIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
int main( void )
{
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    for (int i=0; i<N; i++)
    {
        a[i] = -i;
        b[i] = i * i;
    }
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice ) );

    add<<<N,1>>>( dev_a, dev_b, dev_c );

    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost ) );

    for (int i=0; i<N; i++)
    {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaFree( dev_b ) );
    HANDLE_ERROR( cudaFree( dev_c ) );
    return 0;
}
```

```

0 + 0 = 0
-1 + 1 = 0
-2 + 4 = 2
-3 + 9 = 6
-4 + 16 = 12
-5 + 25 = 20
-6 + 36 = 30
-7 + 49 = 42
-8 + 64 = 56
-9 + 81 = 72
Devam etmek için bir tuşa basın . . .

```

Şekil Ek1- 6: GİB üzerinde vektör toplamı ekran çıktısı

Örnek adım adım işletilsin:

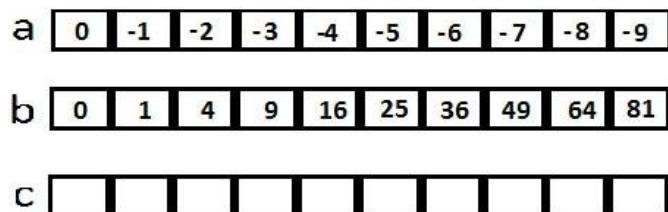
I-)

```
int a[N], b[N], c[N];
```

ve

```
for (int i=0; i<N; i++)
{
    a[i] = -i;
    b[i] = i * i;
}
```

satırları yürütüldüğünde Şekil Ek1-7'de görüldüğü gibi bir bellek yerleşimi düşünülebilir.



Şekil Ek1- 7: a[i] ve b[i] dizilerinin bellek yerleşimi

II-)

```
int *dev_a, *dev_b, *dev_c;
```

kodları ile a,b ve c dizilerimiz için device üzerinde tutulduğu değişkenlerin başlangıç adresleri işaretçilere atandı.

III-)

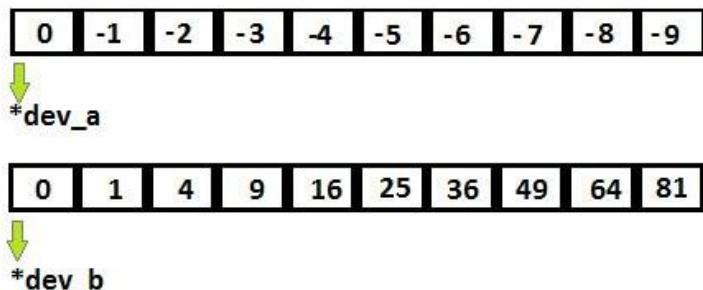
```
HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );
```

satırları ile device üzerinde işaretçi değişkenler için yer ayrıldı.

IV-)

```
HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice ) );
```

satırları sonucu a ve b dizileri GİB üzerine kopyalanır ve Şekil Ek1-8'deki bellek yapısı oluşur.



Şekil Ek1- 8: GPU üzerindeki dizilerin başlangıç adresleri ve içerikleri

V-) **add<<<N,1>>>(dev_a, dev_b, dev_c);**

satırı ile main() içerisinde kernel çağrısı yapılır. Bu çağrı sonucu GİB tarafına geçilir ve hesaplamalar GİB üzerinde yapılır.

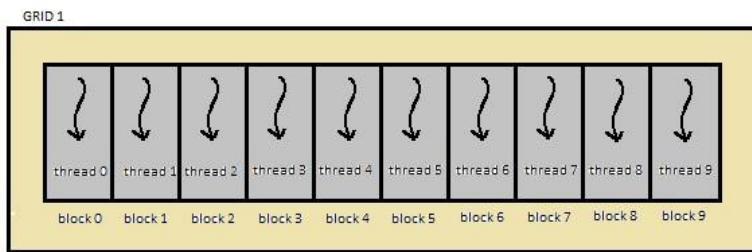
add: kernel fonksiyonunun adı

```
N=blocksPerGrid=gridDim.x=10
1=threadsPerBlock=blockDim.x
```

Örneğin, kernel<<<2,1>>>(...); çağrısı yapıldığını düşünülsün:

Kernel kodunun 2 adet kopyası oluşturulur ve paralel olarak yürütülür. Her grid'de 2 adet block vardır ve bu block'ların herbirinde 1 adet thread işlenmektedir. Toplamda $2 \times 1 = 2$ adet thread işleme girer.

Bizim örneğimizde add<<<N,1>>>(...) olduğu için thread hiyerarşisi Şekil Ek1-9'daki gibi olacaktır.



Şekil Ek1- 9:add<<<N,1>>>(...) kernel çağrısı için thread modeli

```
__global__ void add( int *a, int *b, int *c )
{
    int tid = blockIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

int tid = blockIdx.x;
tid=0 < N --> c[0]=a[0]+b[0]      c[0]=0
tid=1 < N --> c[1]=a[1]+b[1]      c[1]=0
tid=2 < N --> c[2]=a[2]+b[2]      c[2]=2
tid=3 < N --> c[3]=a[3]+b[3]      c[3]=6
tid=4 < N --> c[4]=a[4]+b[4]      c[4]=12
tid=5 < N --> c[5]=a[5]+b[5]      c[5]=20
tid=6 < N --> c[6]=a[6]+b[6]      c[6]=30
tid=7 < N --> c[7]=a[7]+b[7]      c[7]=42
tid=8 < N --> c[8]=a[8]+b[8]      c[8]=56
tid=9 < N --> c[9]=a[9]+b[9]      c[9]=72
```

Şekil Ek1- 10:kernel kodunun adım adım işletilmesi

Heterojen (Paralel +Seri) kodun kernel kısmı, değişken dizi eleman sayısı kadar($N=10$) kopya oluşturulur ve her bir kopyası farklı bir thread tarafından işlenir. Bu örnekte thread sayısı, dizi eleman sayısına eşit olduğu için her bir thread yalnızca

bir kopyayı işlemektedir.(İlerleyen kısımlarda bir thread'in birden fazla kopyayı işlediği örnekler incelenmiştir.)

VI-)

```
HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost ) );
```

kodları ile, kernel'dan dönen c dizisi device'tan host'a taşınır.

VII-)

```
for (int i=0; i<N; i++)
{
    printf( "%d + %d = %d\n", a[i], b[i], c[i] );
}

HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_c ) );
```

Hesaplanan değerler ekranda görüntülenir ve ardından ayrılan bellek bölgeleri serbest bırakılır.

İNDİRİS ve BOYUT HESAPLAMALARI TÜRETİLMİŞ ÖRNEKLER TEMEL KOD:

```
#include <stdio.h>
__global__ void kernel( int* a)
{
    int idx = blockIdx.x*blockDim.x+ threadIdx.x;
    a[idx]=7;
}

int main()
{
    int dimx= 16;
    int num_bytes= dimx*sizeof(int);
    int*d_a=0, *h_a=0;
    h_a= (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes);
    if( 0==h_a|| 0==d_a)
    {
        printf("couldn't allocate memory\n");
        return 1;
```

```

}

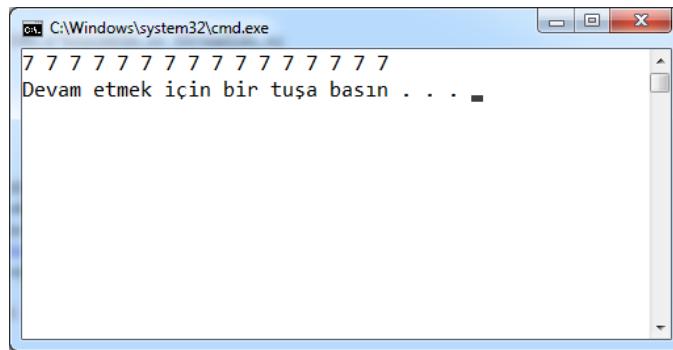
cudaMemset( d_a, 0, num_bytes);
dim3 grid, block;
block.x= 4;
grid.x= dimx/ block.x;

kernel<<<grid, block>>>( d_a);

cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
for(int i=0;i<dimx;i++)
    printf("%d ", h_a[i]);
printf("\n");

free( h_a);
cudaFree( d_a);
return 0;
}

```



Şekil Ek1- 11:Temel Kod Ekran Çıktısı

I-)

```

int dimx= 16;
block.x= 4;
grid.x= dimx/ block.x;
kernel<<<grid, block>>>( d_a);

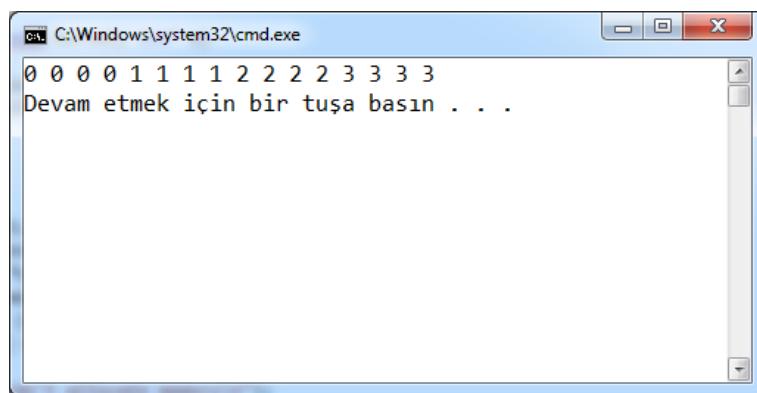
```

olarak tanımlandığına göre, grid=4 ve block=4 olur. Yani kernel<<<4,4>>>(d_a); bulunur. Bu ifade her bir grid'deki block sayısının 4, her bir block'taki thread sayısının 4 olduğunu gösterir. İşlenecek dizinin eleman sayısı 16, işlenecek toplam thread sayısı grid*block=4*4=16 olduğuna göre her bir thread yalnızca bir kez işleyecektir.

Kernel üzerinde yapılan değişiklikler ve ekran çıktıları aşağıda maddeler halinde ifade edilmiştir.

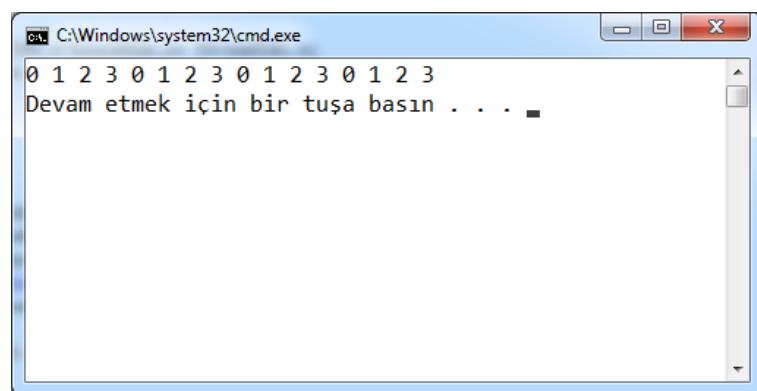
II-)

```
_global_ void kernel( int* a)
{
    int idx = blockIdx.x*blockDim.x+ threadIdx.x;
    a[idx]=blockIdx.x;
}
```



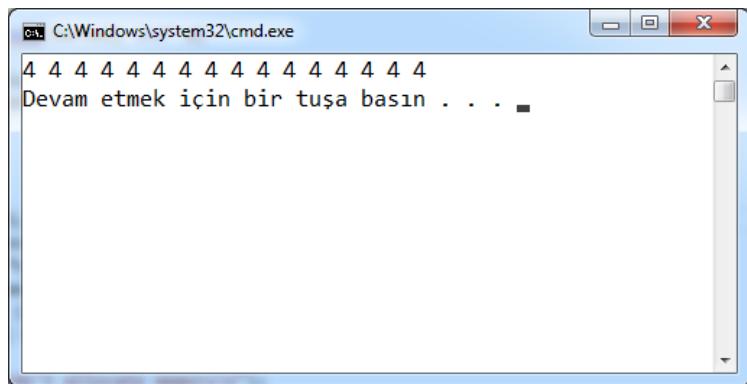
III-)

```
_global_ void kernel( int* a)
{
    int idx = blockIdx.x*blockDim.x+ threadIdx.x;
    a[idx]=threadIdx.x;
}
```



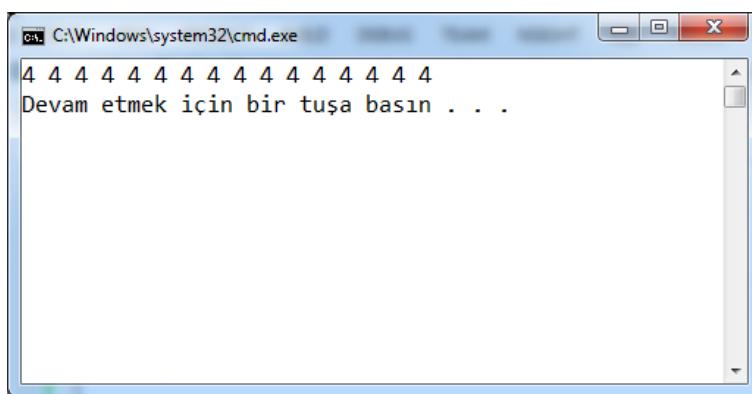
IV-)

```
_global_ void kernel( int* a)
{
    int idx = blockIdx.x*blockDim.x+ threadIdx.x;
    a[idx]=blockDim.x;
}
```



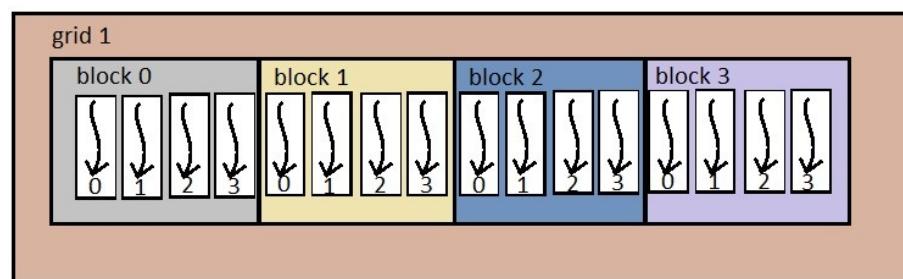
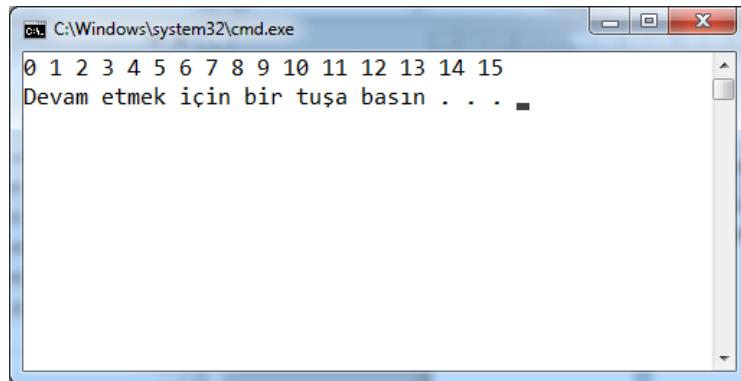
V-)

```
__global__ void kernel( int* a)
{
    int idx = blockIdx.x*blockDim.x+ threadIdx.x;
    a[idx]=gridDim.x;
}
```



VI-)

```
__global__ void kernel( int* a)
{
    int idx = blockIdx.x*blockDim.x+ threadIdx.x;
    a[idx]=idx;
}
```



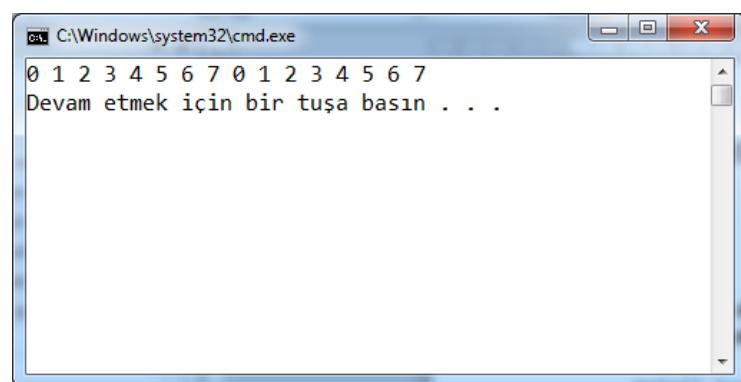
Şekil Ek1- 12:kernel<<<4,4>>> ve dizi boyutu=16 için Grid yapısı

Bu aşamada Temel Kod'un hem main() kısmında hem de kernel kısmında bazı değişiklikler gösterilmiştir.

main() içerisindeki kernel çağrısını *kernel<<<2,8>>>(...)*; şeklinde düzenlenmiş ve kernel'da adım adım değişiklikler yapılmıştır.

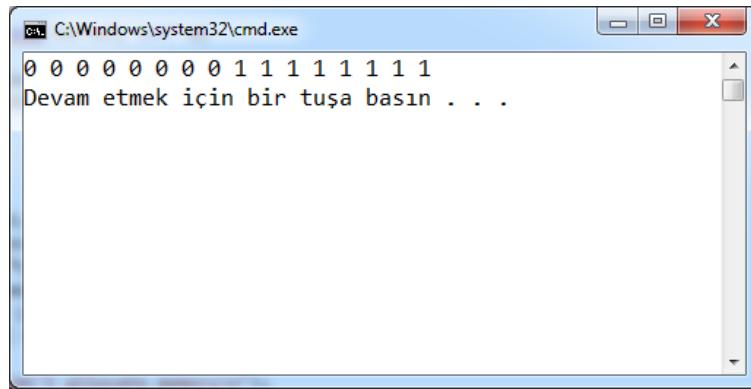
I-)

```
__global__ void kernel( int* a)
{
    int idx = blockIdx.x*blockDim.x+ threadIdx.x;
    a[idx]=threadIdx.x;
}
```



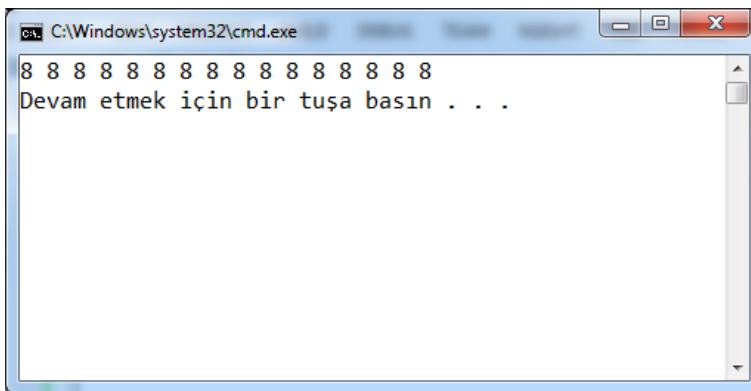
II-)

```
__global__ void kernel( int* a)
{
    int idx = blockIdx.x*blockDim.x+ threadIdx.x;
    a[idx]=blockIdx.x;
}
```



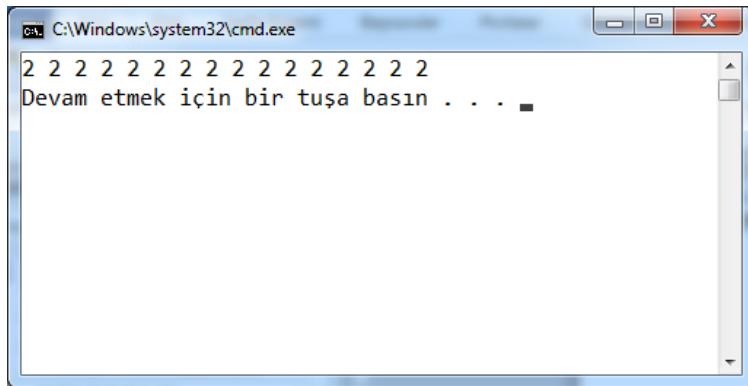
III-)

```
__global__ void kernel( int* a)
{
    int idx = blockIdx.x*blockDim.x+ threadIdx.x;
    a[idx]=blockDim.x;
}
```



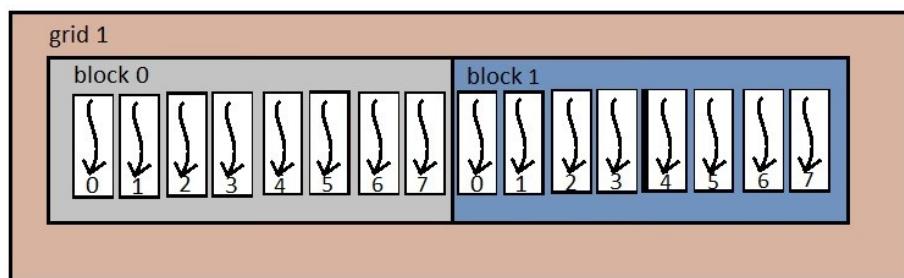
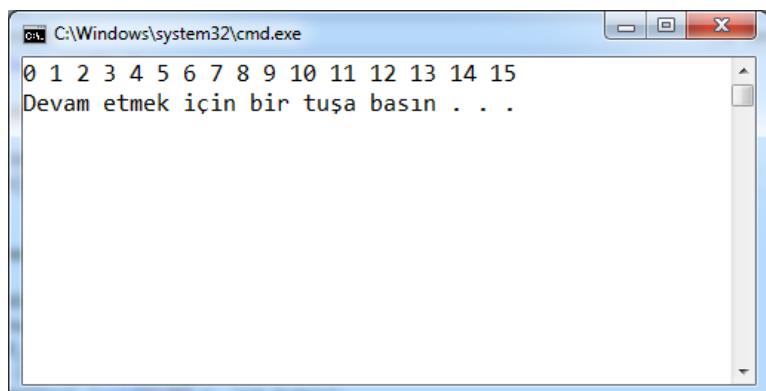
IV-)

```
__global__ void kernel( int* a)
{
    int idx = blockIdx.x*blockDim.x+ threadIdx.x;
    a[idx]=gridDim.x;
}
```



V-)

```
__global__ void kernel( int* a)
{
    int idx = blockIdx.x*blockDim.x+ threadIdx.x;
    a[idx]=idx;
}
```



Şekil Ek1- 13:kernel<<<2,8>>> olan ve 16 elemanlı dizinin grid modeli

NOT: Örnek kodda grid ve block değişkenleri dim3 tipinde yani 3 boyutlu olarak tanımlandı. Bu tanımlama CUDA'ya özel değişken tipleridir. Ancak önceki bölümlerde grid'in en fazla 2 boyutlu olabileceği söyle已被emiştir. CUDA dim3 tipinde

bir değişken tanımlaması gördüğünde, boş bırakılan boyutlara otomatik olarak “1” atar. Yani “**dim3 grid(dim,dim);**” kuralı bozmaz, “**dim3 grid(dim,dim,1);**” ile aynı şeydir.

33*1024 BOYUTLU VEKTÖR TOPLAMI

Bazı durumlarda GİB ,bu sayıdaki thread’i aynı anda işleyebilecek core(fiziksel çekirdek) sayısına sahip değildir. Bu durumda bazı thread’ler birden fazla kez çalışacaktır.

```
#include "../common/book.h"
#define N  (33 * 1024)
__global__ void add( int *a, int *b, int *c )
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}

int main( void ) {
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;

    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
    c = (int*)malloc( N * sizeof(int) );

    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }

    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice ) );
```

```

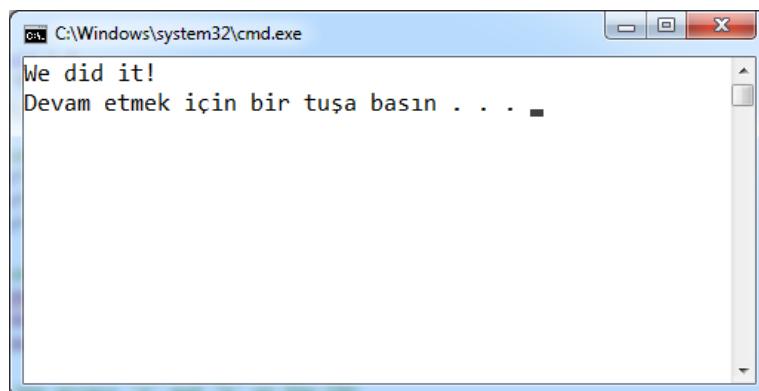
add<<<128,128>>>( dev_a, dev_b, dev_c );

HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost ) );

bool success = true;
for (int i=0; i<N; i++)
{
    if ((a[i] + b[i]) != c[i])
    {
        printf( "Error: %d + %d != %d\n", a[i], b[i], c[i] );
        success = false;
    }
}
if (success)
printf( "We did it!\n" );
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_c ) );

free( a );
free( b );
free( c );
return 0;
}

```



Şekil Ek1- 14:33*1024 boyutlu vektör toplamı

Kernel tanımlaması: `add<<<128,128>>>(dev_a, dev_b, dev_c);`
şeklindedir. İşleme girecek thread sayısı $128 \times 128 = 16384$ 'dir. Toplama işlemi uygulladığımız dizi boyutu(a vektörü ve b vektörü) ise 33792 dir. Bu durumda bazı thread'ler 2 bazıları ise 3 kez çalışacaktır.

Bu mekanizma için kernel kodu:

```

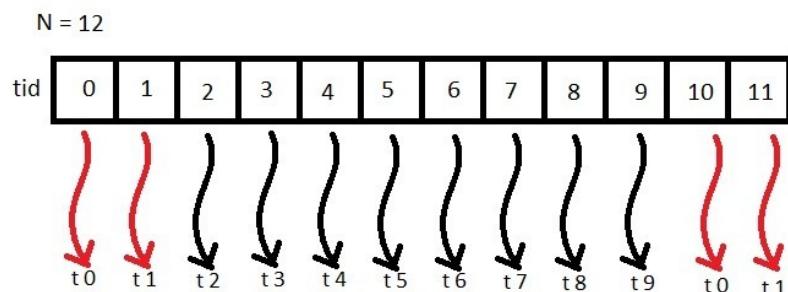
__global__ void add( int *a, int *b, int *c )
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N)
    {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}

```

şeklinde ayarlanmalıdır. Burada dikkat edilmesi gereken yapı:

```
tid += blockDim.x * gridDim.x;
```

satırıdır. Bu kod satırı ile bir thread'in birden fazla kez çalışması sağlanır. Bu yapı, N değişkeninin değerini ve thread sayısını daha küçük sayılar seçerek incelenmiştir. Örneğin N=12 ve mevcut thread sayısının 10 olduğu(kernel<<<10,1>>> olarak düşünülebilir) durumda Şekil Ek1-15'teki yapı ortaya olacaktır.



Şekil Ek1- 15:thread sayısı 10 ve dizi boyutu 12 olduğu durumdaki thread modeli

Şekil Ek1-15'te de görülebileceği gibi thread 0 ve thread 1 iki kez, diğer thread'ler yalnızca bir kez çalışacaktır.

```
tid += blockDim.x * gridDim.x;
```

satırı sayesinde "thread 0" hem tid=0 indisli dizi elemanını hem de tid=10 indisli dizi elemanını işleyecektir. Benzer şekilde "thread 1" hem tid=1 indisli hem de tid=11 indisli dizi elemanını işleyecektir. Böylece thread 0 ve thread 1 iki kez çalışmış oldu.

Büyük boyutlu vektör hesaplamalarında da durum değişmeyecektir. $N=12$ olması ile $N=33*1024$ olması arasında mantıksal bir fark yoktur.

GİB ÜZERİNDE İKİ BOYUTLU DİZİ HESAPLAMASI

```
#include <stdio.h>
__global__ void kernel( int* a, int dimx, int dimy)
{
    int ix = blockIdx.x*blockDim.x+ threadIdx.x;
    int iy= blockIdx.y*blockDim.y+ threadIdx.y;
    int idx= iy*dimx+ ix;
    a[idx] = a[idx]+1;
}

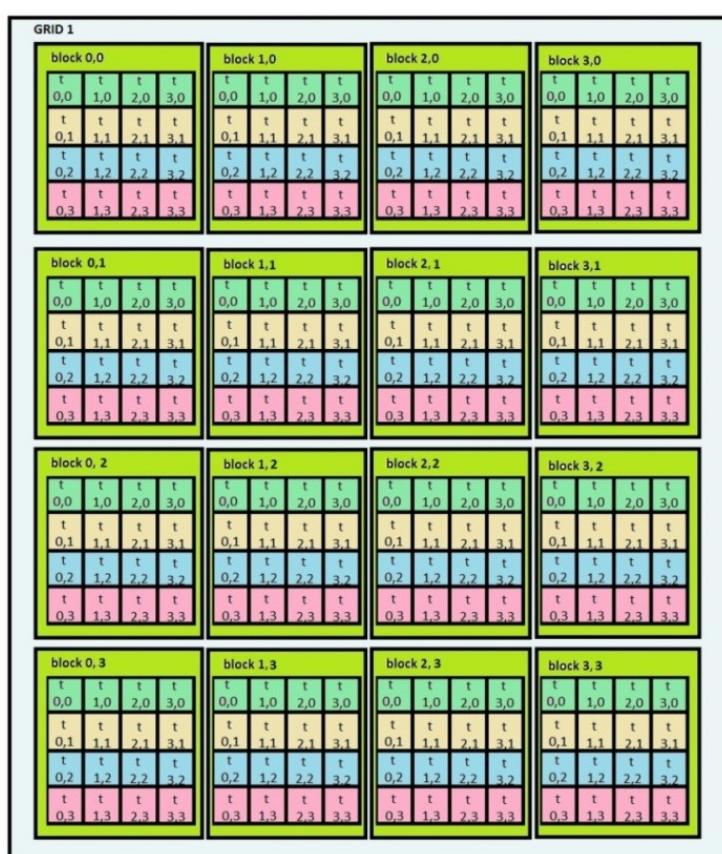
int main()
{
    int dimx= 16;
    int dimy= 16;
    int num_bytes= dimx*dimy*sizeof(int);
    int*d_a=0, *h_a=0;
    h_a= (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes);
    if( 0==h_a|| 0==d_a)
    {
        printf("couldn't allocate memory\n");
        return 1;
    }
    cudaMemset( d_a, 0, num_bytes);
    dim3 grid, block;
    block.x= 4;
    block.y= 4;
    grid.x= dimx/ block.x;
    grid.y= dimy/ block.y;
    kernel<<<grid, block>>>( d_a, dimx, dimy );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
    for(int row=0; row<dimy; row++)
    {
        for(int col=0; col<dimx; col++)
            printf("%d ", h_a[row*dimx+col] );
        printf("\n");
    }
    free( h_a);
    cudaFree( d_a);
    return 0;
}
```

```

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

```

Şekil Ek1- 16: GİB üzerinde iki boyutlu dizi hesaplaması ekran çıktısı



Şekil Ek1- 17:kernel<<<(4x4),(4x4)>>> için 16x16 boyutlu matrisin Grid modeli

kernel<<<grid, block>>>(d_a, dimx, dimy);

grid=4(her bir grid'teki block sayısı)

block=4(her bir block'taki thread sayısı)

Bu örnekte $16 \times 16 = 256$ adet thread, 16 adet block ve 1 adet grid bulunmaktadır.

THREAD SENKRONİZASYONU

```
//dot.cu
#include "../common/book.h"
#define imin(a,b) (a<b?a:b)
const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid = imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
__global__ void dot( float *a, float *b, float *c )
{
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float temp = 0;
    while (tid < N)
    {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    cache[cacheIndex] = temp;
    __syncthreads();

    int i = blockDim.x/2;
    while (i != 0)
    {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}

int main( void ) {
    float *a, *b, c, *partial_c;
    float *dev_a, *dev_b, *dev_partial_c;

    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );

    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b,N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,blocksPerGrid*sizeof(float) ) );
}
```

```

HANDLE_ERROR( cudaMalloc( (void**)&dev_b,N*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,blocksPerGrid*sizeof(float) ) );

for (int i=0; i<N; i++)
{
    a[i] = i;
    b[i] = i*2;
}

HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),cudaMemcpyHostToDevice ) );

dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,dev_partial_c );

HANDLE_ERROR(cudaMemcpy(partial_c,dev_partial_c,blocksPerGrid*sizeof(float),
                      cudaMemcpyDeviceToHost ) );

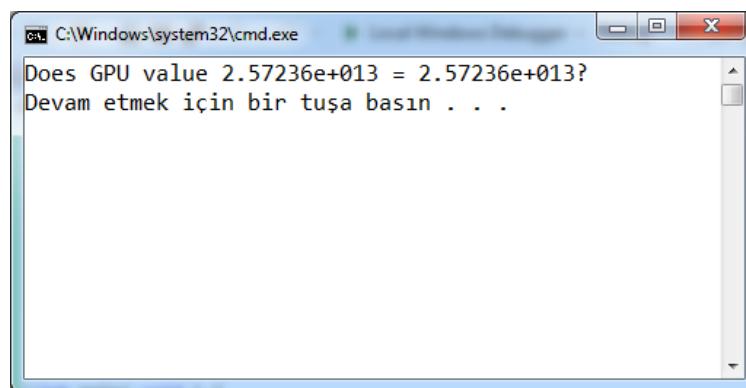
c = 0;
for (int i=0; i<blocksPerGrid; i++)
{
    c += partial_c[i];
}

#define sum_squares(x) (x*(x+1)*(2*x+1)/6)
printf( "Does GPU value %.6g = %.6g?\n", c,2 * sum_squares( (float)(N - 1) ) );

HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_partial_c ) );

free( a );
free( b );
free( partial_c );
}

```



Bu örnekte daha önce karşılaşılmayan bir çok yapı vardır. Maddeler hâlinde incelenec olursa:

i-)

```
N = 33 * 1024=33792  
threadsPerBlock = 256;  
blocksPerGrid =imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
```

ve

```
dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b, dev_partial_c );
```

olduğuna göre, `dot<<<32,256>>>` şeklinde hesaplanır. İşleyecek thread sayısı toplamda $32 \times 256 = 8192$ 'dir. Çarpımı yapılacak dizi eleman sayısı ise $33 \times 1024 = 33192$ 'dir. Bu durumda bazı thread'ler birden fazla kez çalışacaktır. Buraya kadar anlatılan ifadeler daha önce de incelenmiştir. Farklı olan yapılardan biri $(N+threadsPerBlock-1) / threadsPerBlock$ yapısıdır. Bu kod parçasının kullanılma amacı bir örnek verilerek incelenec olursa:

27 piksellik bir görüntü mevcut olsun. 5'lik thread block'ları şeklinde gruplanmak isteniyor. 27 sayısında direkt 5'e bölünürse kalan 2 piksellik görüntü kaybolacaktır. Eğer $(5-1)=4 \nmid (27+4)/5$ işlemi uygulanırsa 6 parça elde edilmiş olur. Bu durumda piksel kaybı olmayacağındır. Parça sayısı =6 ve her bir parçanın boyu 5 ise $5 \times 6 = 30 > 27$.

ii-) `dot.cu` örneğinin kernel kısmı incelenec olursa:

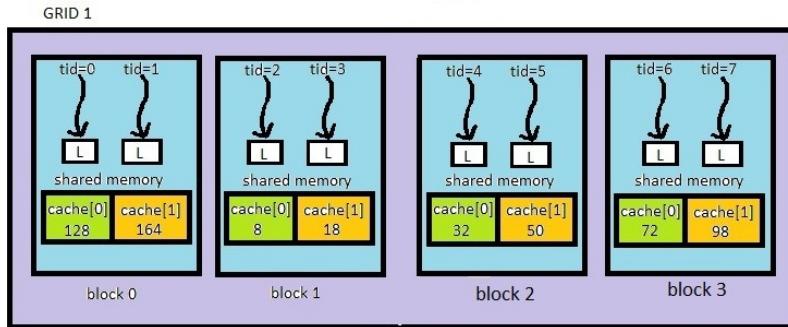
Anlaşılmasının kolay olması açısından kernel çağrısı `dot<<<4,2>>>` şeklinde ve $N=10$ olarak belirlensin. Bu durumda `blocksPerGrid=4`, `threadsPerBlock=2` olacaktır. “`_shared_`” niteleyicisi , “`_global_`” niteleyicisi gibi düşünülebilir. Cache dizisinin shared memory'de olduğunu belirtir.

```
_shared_ float cache[threadsPerBlock];  
int tid = threadIdx.x + blockIdx.x * blockDim.x;  
int cacheIndex = threadIdx.x;  
  
float temp = 0;  
while (tid < N) {  
    temp += a[tid] * b[tid];
```

```

    tid += blockDim.x * gridDim.x;
}
cache[cacheIndex] = temp;

```



Şekil Ek1- 18: dot<<<4,2>>> olan 10 elemanlı bir dizinin Grid yapısı

iii-)

Thread senkronizasyonu

Amaç: Örnekte de görülebileceği gibi 8 thread 10 elemanlık bir diziyi işlemektedir. “Thread 0” ve “thread 1” 2 kez çalışırken, diğer threadler 1 kez çalışırlar. Kodun kalan kısmında tüm thread’ler yine kullanılacaktır ve işlemlerine aynı anda başlamaları istenmektedir. Bunun için `_syncthreads()`; fonksiyonu kullanılmaktadır.

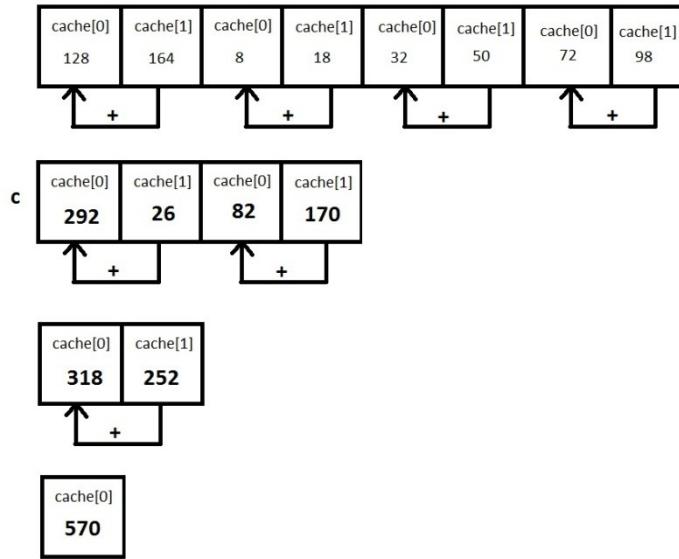
iv-)

```

int i = blockDim.x/2;
while (i != 0)
{
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}

if (cacheIndex == 0)
    c[blockIdx.x] = cache[0];

```



Şekil Ek1- 19: dot.cu örneği bellek modeli

Event API Kullanan Örnek Kod:

```
#include <stdio.h>
#include <cuda_runtime.h>
#include <helper_cuda.h>
#include <helper_functions.h>

__global__ void increment_kernel(int *g_data, int inc_value)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    g_data[idx] = g_data[idx] + inc_value;
}

int correct_output(int *data, const int n, const int x)
{
    for (int i = 0; i < n; i++)
        if (data[i] != x)
    {
        printf("Error! data[%d] = %d, ref = %d\n", i, data[i], x);
        return 0;
    }

    return 1;
}

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

```

int devID;
cudaDeviceProp deviceProps;
printf("[%s] - Starting...\n", argv[0]);
devID = findCudaDevice(argc, (const char **)argv);

checkCudaErrors(cudaGetDeviceProperties(&deviceProps, devID));
printf("CUDA device [%s]\n", deviceProps.name);

int n = 16 * 1024 * 1024;
int nbytes = n * sizeof(int);
int value = 26;

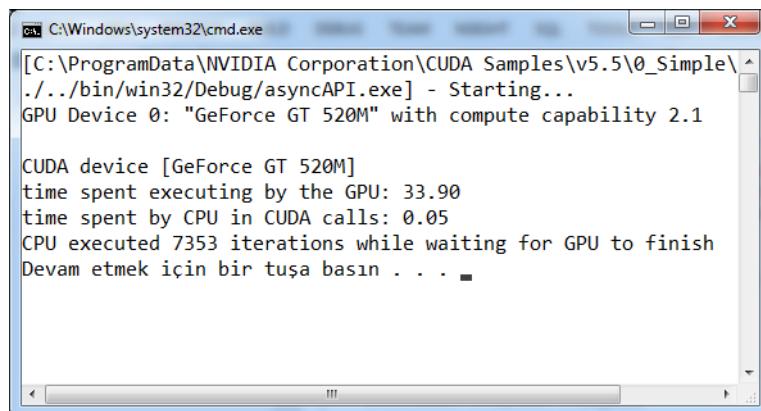
int *a = 0;
checkCudaErrors(cudaMallocHost((void **)&a, nbytes));
memset(a, 0, nbytes);

int *d_a=0;
checkCudaErrors(cudaMalloc((void **)&d_a, nbytes));
checkCudaErrors(cudaMemset(d_a, 255, nbytes));

dim3 threads = dim3(512, 1);
dim3 blocks = dim3(n / threads.x, 1);
cudaEvent_t start, stop;
checkCudaErrors(cudaEventCreate(&start));
checkCudaErrors(cudaEventCreate(&stop));
StopWatchInterface *timer = NULL;
sdkCreateTimer(&timer);
sdkResetTimer(&timer);
checkCudaErrors(cudaDeviceSynchronize());
float gpu_time = 0.0f;
sdkStartTimer(&timer);
cudaEventRecord(start, 0);
cudaMemcpyAsync(d_a, a, nbytes, cudaMemcpyHostToDevice, 0);
increment_kernel<<<blocks, threads, 0, 0>>>(d_a, value);
cudaMemcpyAsync(a, d_a, nbytes, cudaMemcpyDeviceToHost, 0);
cudaEventRecord(stop, 0);
sdkStopTimer(&timer);
unsigned long int counter=0;
while (cudaEventQuery(stop) == cudaErrorNotReady)
{
    counter++;
}
checkCudaErrors(cudaEventElapsedTime(&gpu_time, start, stop));
printf("time spent executing by the GPU: %.2f\n", gpu_time);
printf("time spent by CPU in CUDA calls: %.2f\n", sdkGetTimerValue(&timer));
printf("CPU executed %lu iterations while waiting for GPU to finish\n", counter);

```

```
    bool bFinalResults = (bool)correct_output(a, n, value);
    checkCudaErrors(cudaEventDestroy(start));
    checkCudaErrors(cudaEventDestroy(stop));
    checkCudaErrors(cudaFreeHost(a));
    checkCudaErrors(cudaFree(d_a));
    cudaDeviceReset();
    exit(bFinalResults ? EXIT_SUCCESS : EXIT_FAILURE);
}
```



Şekil Ek1- 20: Event API örneği ekran çıktısı

ÖZGEÇMİŞ

Nezihe SÖZEN 1991 yılında İstanbul'da doğdu; ilk ve orta öğrenimini aynı şehirde tamamladı; Kemal Hasoğlu Lisesi'nden mezun olduktan sonra 2010 yılında Karabük Üniversitesi Bilgisayar Mühendisliği bölümüne girdi; halen lisans eğitimine bu kurumda devam etmektedir.

İLETİŞİM BİLGİLERİ

Tel: (536) 981 02 16

E-posta: nezihe.sozen@gmail.com

